Learning and Nonlinear Models (L&NLM) – Journal of the Brazilian Neural Network Society, Vol. 8, Iss. 3, pp. 163–173, 2010

MaPI: UM *FRAMEWORK* PARA PARALELIZAÇÃO DE ALGORITMOS

S. Ribas, M. H. P. Perché, I. M. Coelho, P. L. A. Munhoz, M. J. F. Souza, A. L. L. Aquino
Universidade Federal de Ouro Preto – Ouro Preto, Minas Gerais, Brasil
{sabir, mariohpp, imcoelho, pablomunhoz, marcone, alla}@iceb.ufop.br

Resumo – Este trabalho apresenta o MaPI, um *framework* que implementa a abstração *MapReduce* na linguagem C++. Ao utilizar o MaPI, o usuário é capaz de implementar uma aplicação paralela sem se preocupar com a forma de comunicação entre os processos ou como o sistema fará a paralelização. Além disso, toda a implementação feita pelo usuário pode ser sequencial. Para ilustrar o funcionamento do *framework*, este foi usado na paralelização de um algoritmo heurístico de otimização aplicado a um problema clássico de otimização, o Problema do Caixeiro Viajante. Os resultados obtidos comprovam a eficiência do *framework* como ferramenta de auxílio ao desenvolvimento de procedimentos paralelos de otimização.

Palavras-chave – MaPI, Algoritmos Paralelos, *MapReduce*, *Framework*, Otimização, Heurísticas.

Abstract – This paper introduces MaPI, a framework that implements the MapReduce abstraction. Using MaPI, the user is able to implement parallel applications without worrying about the messages transmission between the processes, or how the system will do the parallelization. Furthermore, all the implementation made by the user can be sequential. In order to demonstrate how the framework works, it is used to parallelize an optimization heuristic algorithm applied to a classical optimization problem, the Traveling Salesman Problem. The results show the efficiency of the framework as a tool to help the development of parallel optimization procedures.

Keywords – MaPI, Parallel Algorithms, MapReduce, Framework, Optimization, Heuristics.

1. INTRODUÇÃO

A computação paralela tem provocado um grande impacto nas mais diversas áreas, desde simulações computacionais para o meio científico, até aplicações comerciais e industriais. Essa forma de computação opera sob o princípio de que problemas de grande porte geralmente podem ser divididos em problemas menores, que são resolvidos em paralelo. Quando comparados com a computação sequencial, os sistemas paralelos são geralmente mais complexos, o que dificulta a programação para esses sistemas.

Com uma abordagem paralela podemos resolver vários problemas que necessitam de muitos recursos computacionais, principalmente processamento e memória. Essas restrições podem ser resolvidas com a utilização de computadores pessoais ligados em redes compartilhando a mesma tarefa, caracterizando um ambiente de programação paralela. Alguns problemas considerados inviáveis para serem resolvidos por um único computador podem ser resolvidos em tempo hábil por meio de paralelismo. Como exemplo, citamos: previsão do tempo, modelo de movimentação de corpos celestes e problemas de otimização combinatória.

Apesar das vantagens da computação paralela, sua principal desvantagem é a dificuldade de implementação. Se comparados com sistemas sequenciais, sistemas paralelos são geralmente mais complexos. Isso ocorre pois a concorrência introduz diversos problemas, entre eles o projeto de um algoritmo paralelo, a forma de comunicação entre os processos e a sincronização dos dados.

Para simplificar a implementação de algoritmos paralelos, apresentamos o MaPI[1], uma implementação em C++ da abstração *MapReduce*. O objetivo é fornecer ao usuário todo o aparato necessário para a implementação rápida de procedimentos paralelos. A vantagem desse *framework* é que um usuário inexperiente quanto à programação distribuída é capaz de implementar uma aplicação paralela sem se preocupar com a forma de comunicação entre os processos ou mesmo como o sistema fará a paralelização. Desta forma, toda a implementação feita pelo usuário pode ser sequencial.

Com o intuito de ilustrar o funcionamento do MaPI e sua aplicabilidade a problemas de otimização, desenvolvemos neste trabalho uma versão paralela de um dos procedimentos mais utilizados na otimização por métodos heurísticos, o gerador de melhor vizinho. Para testá-lo, foi implementado o método de Descida [1] usando o gerador de melhor vizinho paralelo e este, por sua vez, foi aplicado a um problema clássico de otimização, o Problema do Caixeiro Viajante (PCV) [2, 3].

A motivação para utilizarmos, como estudo de caso, problemas de otimização, parte do fato de que recentemente tem-se observado na literatura propostas de paralelização para melhorar o desempenho de algumas heurísticas, como por exemplo algoritmos evolutivos e métodos de busca local [4–6]. Isso se deve ao crescimento das facilidades em se montar um *cluster* e também pelo crescente investimento em multiprocessadores por parte dos fabricantes. Com isso, o tempo de resolução de

[1]MaPI integra o projeto MapReduce++, disponível sob licença LGPL em *http://sourceforge.net/projects/mapreducepp*. Tal projeto oferece implementações *MapReduce* tanto para multiprocessadores (a biblioteca MapMP) quanto para multicomputadores (o *framework* MaPI).

problemas de otimização poderia ser reduzido se os algoritmos utilizassem os recursos *multicore* desses processadores, assim como versões de sistemas operacionais especializados em ambientes distribuídos e construção de *clusters*.

O restante deste trabalho está organizado como segue. A segunda Seção apresenta trabalhos relacionados à computação paralela de alto desempenho e à abstração *MapReduce*. A terceira Seção apresenta o MaPI, a solução desenvolvida neste trabalho para o desenvolvimento rápido e transparente de sistemas de computação paralela de alto desempenho. A quarta Seção apresenta a aplicação do MaPI à paralelização de um algoritmo de otimização e apresentada os resultados obtidos. Por fim, a quinta Seção apresenta as conclusões e trabalhos futuros.

2 TRABALHOS RELACIONADOS

MapReduce é uma abstração simples e poderosa, geralmente aplicada ao processamento ou geração de grandes massas de dados. Em [7], é apresentada uma visão geral sobre a implementação *MapReduce* da Google, a qual facilita muito o trabalho de seus programadores. Esse modelo foi feito para processar grandes conjuntos de dados de uma maneira massivamente paralela e é baseado nos seguintes fatores [8]: (i) iteração sobre a entrada; (ii) computação sobre cada um dos pares (*chave,valor*) da entrada; (iii) agrupamento de todos os valores intermediários por chaves; (iv) iteração sobre os grupos resultantes; (v) redução de cada grupo.

O usuário da biblioteca *MapReduce* expressa a computação como duas funções: *map* e *reduce*. A função *map* recebe um par como entrada e produz um conjunto de pares intermediários também na forma (*chave,valor*). A biblioteca *MapReduce* agrupa todos os valores intermediários associados à mesma chave intermediária e os passa à função *reduce*. A função *reduce* aceita uma chave intermediária e o conjunto de valores relacionados àquela chave. Essa função junta esses valores para formar um conjunto, possivelmente menor, de valores. O esquema de comunicação *MapReduce* é apresentado na Figura 1, onde k, v e *EP* representam chaves, valores e processos, respectivamente.

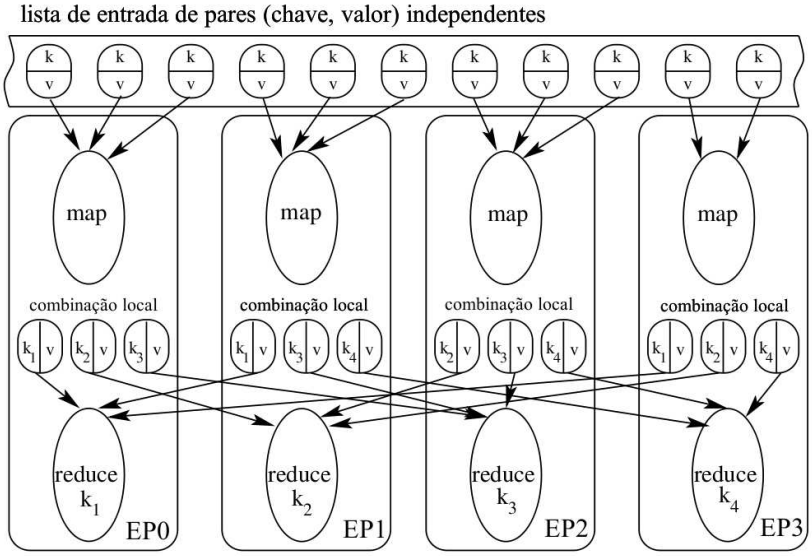


Figura 1: Esquema de comunicação *MapReduce*. Adaptado de [9]

A simplicidade da abstração permite a introdução de diversas otimizações. A mais óbvia é a execução concorrente do mapeamento e da redução. Esta paralelização automática foi proposta em [7]. Outra característica muito útil dessa abstração em sistemas de larga escala é a inerente flexibilidade a falhas, pois as tarefas, de mapeamento ou de redução, lentas ou em falha, podem simplesmente ser reiniciadas em outros nós. Assim, *MapReduce* permite ao desenvolvedor da aplicação focar em aspectos importantes do algoritmo para resolver o problema, permitindo que este ignore questões referentes à distribuição de dados, sincronização, execução paralela, tolerância a falhas e monitoramento.

Observa-se que embora o paradigma funcional seja a base da abstração *MapReduce*, funcionalidades semelhantes aos conceitos chave dessa abstração (o mapeamento e a redução) também estão presentes em linguagens que seguem outros paradigmas. Por exemplo, a biblioteca padrão de C++ (*C++ Standard Library*) [10] atualmente oferece as funções `std::transform()` e `std::accumulate()`, que são respectivamente similares ao mapeamento e à redução.

Vários algoritmos paralelos podem ser expressos em *MapReduce* [9]. Alguns exemplos são: ordenação; contagem de elementos em listas, como palavras em documentos; busca distribuída; ou transposição de grafos ou listas [7]. Outros algoritmos, tais como o caminho mínimo de *Bellman-Ford* ou *PageRank* [11], podem ser modelados de forma iterativa invocando *MapReduce*. Além disso, *MapReduce* pode ser aplicado de forma eficaz no contexto de pesquisa em aprendizagem de máquina. Em [12], os autores exploram ambientes *multicore* e apresentam versões paralelas baseadas em *MapReduce* de uma variedade de algoritmos de aprendizagem de máquina, incluindo regressão logística, *Naive Bayes* e *backpropagation*. Em [13], os autores usam o potencial de *MapReduce* para mineração de dados em larga escala em arquiteturas com vários computadores. Tal trabalho também

apresenta um exemplo de implementação de uma máquina de aprendizado na forma de *MapReduce*. *MapReduce* também vem ganhando popularidade em educação [14].

Existem diversas implementações da abstração *MapReduce*, em várias linguagens. Dentre as implementações que suportam C++ destacam-se Google MapReduce[2], nCluster[3], Qt Concurrent[4] e Sector/Sphere[5]. A Google, em 2003, popularizou o modelo de programação *MapReduce* para o processamento de bases de dados distribuídos através de milhares de nós. A implementação MapReduce dessa empresa foi desenvolvida em C++, com interfaces para Java e Python. O trabalho utiliza esse mesmo conceito de *MapReduce*, porém a implementação da Google não é de código aberto e não se encontram disponíveis os detalhes de sua implementação.

O software nCluster utiliza o *framework* SQL-MapReduce, que permite ao desenvolvedor escrever funções SQL-MapReduce em linguagens como Java, C#, Python, C++ e R, sendo este capaz de utilizá-las em um banco de dados. Os usuários do banco de dados podem, então, invocar essas funções usando SQL padrão por meio do nCluster. Segundo a Aster Data[6], proprietária do nCluster, as funções SQL-MapReduce são de simples implementação e podem ser facilmente integradas as declarações SQL.

Qt Concurrent é uma biblioteca em C++ que auxilia o desenvolvimento de aplicações *multi-thread*. Esta biblioteca utiliza o conceito de *templates* que possibilita a escrita de programas abstraindo funções de baixo nível para a criação de *threads*, como seções críticas, exclusão múltipla e sincronização. A biblioteca inclui uma interface em estilo de programação funcional para o processamento paralelo de listas e uma implementação *MapReduce* para memória compartilhada.

Sector/Sphere é muito semelhante ao Google File System/MapReduce, suportando armazenamento distribuído de dados e processamento sobre *clusters* formados por computadores convencionais. Segundo os desenvolvedores do projeto, Sector é um sistema de arquivos distribuído de alta performance, escalável e seguro. Sphere é um mecanismo de processamento paralelo de dados de alta performance capaz de processar arquivos do sistema de arquivos Sector por uma interface de programação muito simples, e esse mecanismo corresponde a uma generalização do *MapReduce*.

A implementação apresentada neste trabalho foi desenvolvida em C++ usando a interface de troca de mensagens MPI[7]. Dessa forma, o MaPI é voltado para programação paralela em ambientes de memória distribuída, diferentemente do Qt Concurrent, mas podendo também ser utilizado em memória compartilhada. MaPI é uma implementação de propósito geral, não restrita a processamento de bases de dados ou de arquivos, como o nCluster ou Sector/Sphere, podendo ser facilmente utilizado na resolução de várias classes de problemas.

No contexto da otimização, existem diversas abordagens que tratam desde problemas relacionados a algoritmos paralelos de busca local para o problema da satisfabilidade, passando por algoritmos meméticos paralelos aplicado à resolução do problema de sequenciamento em máquina simples, até problemas de logística de grande escala [15–17]. Em [4], são descritas estratégias simples para a implementação de heurísticas paralelas baseadas em GRASP. Em [5], é apresentado um algoritmo paralelo que utiliza uma técnica de busca local clássica para computar uma árvore *Steiner* no plano bidimensional. Em [6], é proposta a utilização de metaheurísticas e computação paralela para a resolução de um problema real de roteamento de veículos com frota heterogênea, janelas de tempo e entregas fracionadas, no qual a demanda dos clientes pode ser maior que a capacidade dos veículos.

Embora haja na literatura inúmeros trabalhos sobre algoritmo paralelos na resolução de problemas de otimização, são poucos os que utilizam a abstração *MapReduce*. Da mesma forma, existem vários algoritmos baseados em *MapReduce*, porém poucos são ligados à otimização. Esses foram os principais motivadores para a escolha de problemas de otimização como estudo de caso da aplicação do MaPI.

3 SOLUÇÃO IMPLEMENTADA

3.1 MOTIVAÇÃO FUNCIONAL

Embora MaPI seja um *framework* para uma linguagem procedural e orientada a objetos, o paradigma funcional é um grande motivador quanto à implementação paralela da abstração *MapReduce*. O paradigma funcional utiliza conceitos muito poderosos, como é o caso das funções de ordem superior. Define-se uma função de ordem superior aquela que possui, como parâmetro, uma outra função e é capaz de aplicá-la a um conjunto de valores.

Na linguagem funcional Haskell[8], *map* é uma função de ordem superior cujo tipo é apresentado a seguir:

```
map :: (a->b) -> [a] -> [b]
```

Mais especificamente, *map* é uma função que aplica uma função a uma lista de valores e retorna a lista resultante. A função transforma um elemento do tipo *a* em um do tipo *b* e, por meio desta função, uma lista de elementos do tipo *a* é transformada em uma lista de elementos do tipo *b*. Note que *map* pode ser claramente paralelizada, pois tomando-se os devidos cuidados para

[2]http://labs.google.com/papers/mapreduce.html
[3]http://www.asterdata.com/product/embedded-app.php
[4]http://labs.trolltech.com/page/Projects/Threads/QtConcurrent
[5]http://sector.sourceforge.net/
[6]http://www.asterdata.com/
[7]Saiba mais em *http://www.mcs.anl.gov/research/projects/mpi/*
[8]Haskell é uma linguagem de programação puramente funcional, de propósito geral. Suas características incluem: suporte a funções recursivas e tipos de dados, casamento de padrões, *list comprehensions*, *guard statements* e avaliação preguiçosa. Saiba mais em *http://www.haskell.org/*

que a ordenação final dos elementos seja mantida, a função de transformação $(a \rightarrow b)$ pode ser aplicada paralelamente a cada elemento da lista de entrada.

A idéia básica por trás do MaPI é mapear e reduzir, distribuindo o processamento e capturando os resultados. Tendo em vista a definição de *map* na linguagem Haskell, onde o usuário precisa apenas implementar a função de transformação e fornecer a lista, a redução pode ser vista como uma função na seguinte forma, em que $([b] \rightarrow [c])$ é uma função que transforma uma lista de elementos do tipo b em uma lista do tipo c, não necessariamente do mesmo tamanho:

```
reduce :: ([b] -> [c]) -> [b] -> [c]
```

A intenção do *framework* é que o usuário, ao implementar as funções $(a \rightarrow b)$ e $([b] \rightarrow [c])$, tenha um sistema que distribua o processamento tanto nos núcleos de um computador multiprocessado quanto nos nodos de um *cluster*.

3.2 MaPI: PRIMEIROS PASSOS

Nesta Seção são apresentados os passos para se implementar um sistema paralelo usando a versão 1.1 do MaPI. O sistema exemplo[9] desenvolvido nesta seção visa a aplicação da abstração *MapReduce* a uma sequência de *strings*. A função de mapeamento recebe uma *string* e a ela adiciona o trecho "(mapeada)" e a função de redução retorna a primeira *string* da sequência mapeada. Eis os passos.

Primeiramente, crie um arquivo ".cpp", doravante "ex00.cpp", e inclua o cabeçalho da biblioteca MaPI.

```
#include "lib/MaPI.hpp"
```

Nessa versão do MaPI, a entrada de dados para o *MapReduce* é feita por meio de um vetor de *strings*. Neste caso, o exemplo apresenta um vetor de duas posições.

```
#include "lib/MaPI.hpp"
int main(int argc, char** argv)
{
    vector<string> inputs;
    inputs.push_back("Entrada 1");
    inputs.push_back("Entrada 2");
}
```

O próximo passo é definir a função que será aplicada a cada elemento do vetor de entrada. Neste caso, usa-se a função *fmap*:

```
string fmap(string input, string shared)
{
    string mapped = input + shared; printf("%s\n", mapped.c_str());
    return mapped;
}
```

A aplicação da função *fmap* a cada um dos elementos da entrada (*input*) gera um vetor de outros elementos, ditos "mapeados". A função de redução é a que resume o resultado do *MapReduce*. Esta função pode ser usada para mesclar os elementos do conjunto mapeado ou pode escolher um ou mais elementos do mesmo conjunto. Em nosso exemplo, a redução é feita pela função *freduce*, que retorna um vetor com um único elemento, a primeira *string* do vetor mapeado acrescida do texto "e reduzida".

```
vector<string> * freduce (vector<string> * mapped, string shared)
{
    vector<string> * reduced = new vector<string>;
    reduced->push_back(mapped->at(0) + " e reduzida");
    return reduced;
}
```

Os próximos passos são a inicialização e a finalização do *framework* pelas funções *init* e *finalize*, respectivamente. Por fim, tem-se a chamada do *MapReduce*. A Seção 3.3 detalha a estrutura do *framework* e justifica sua inicialização e a finalização. Observe que tanto a função de mapeamento quanto a de redução recebem também um parâmetro denominado *shared* do tipo *string*, cujo tipo é definido na declaração da classe (em forma de *template*) e o conteúdo é passado na inicialização. Este parâmetro adicional tem como objetivo compartilhar dados entre os processos. O código completo do exemplo é apresentado a seguir:

[9]Para seguir este exemplo deve-se ter instalada alguma implementação do MPI. Em nossos testes foi utilizada a implementação OpenMPI, que pode ser obtida por meio do gerenciador de pacotes Synaptic, geralmente disponível em distribuições Linux. Para isso, basta entrar no Synaptic, procurar por "OpenMPI" e instalar os pacotes "libopenmpi1", "libopenmpi-dev", "libopenmpi-bin" e "libopenmpi-commom".

```
#include "lib/MaPI.hpp"

string fmap(string input, string shared)
{
    string mapped = input + shared; printf("%s\n", mapped.c_str());
    return mapped;
}

vector<string> * freduce (vector<string> * mapped, string shared)
{
    vector<string> * reduced = new vector<string>;
    reduced->push_back(mapped->at(0) + " e reduzida");
    return reduced;
}

int main(int argc, char** argv)
{
    MaPI<string> mr;
    mr.init(argc,argv,fmap,string(" mapeada"));

    vector<string> inputs;
    inputs.push_back("Entrada 1");
    inputs.push_back("Entrada 2");

    vector<string> * output = mr.mapreduce(fmap,freduce,&inputs);
    printf("%s\n", output->at(0).c_str());

    mr.finalize();
}
```

Para compilar e executar o programa use a linha de comando abaixo, na qual "`-np 3`" significa que o `mpirun` criará três processos:

```
mpiCC ex00.cpp -o ex00 && mpirun -np 3 ex00
```

Ao executar esse comando, o programa supra produzirá a seguinte saída:

```
== Mapper ==
Entrada 1 mapeada
Entrada 2 mapeada
== Reducer ==
Entrada 1 mapeada e reduzida
```

O resultado apresentado pelo programa consiste na exibição do vetor mapeado e do resultado da redução.

3.3 ESTRUTURA DO MaPI

MaPI foi desenvolvido em C++ usando MPI, daí o nome *"MapReduce over MPI"*. *Message Passing Interface*, ou simplesmente MPI, é uma interface padrão de troca de mensagens que oferece diversas abstrações que padronizam e facilitam o desenvolvimento de aplicações paralelas. Por ser um padrão, existem diversas implementações, tanto comerciais quanto de código aberto. Entre as implementações comerciais citamos Intel MPI, Sun MPI, HP-MPI e MPI/Pro; e entre as de código aberto, citamos OpenMPI, MPICH, MPICH2 e LAM/MPI. Em MPI, uma aplicação é constituída de processos que se comunicam por meio de funções de envio e recebimento de mensagens. Normalmente, o número de processos em uma aplicação MPI é fixo. No entanto, algumas implementações oferecem suporte a sistemas com número variável de processos. Os processos podem usar mecanismos de comunicação ponto-a-ponto e também podem invocar operações coletivas de comunicação.

Ao se executar um programa MPI, são inicializados vários processos idênticos. Inicialmente, a única diferença é o identificador[10] de cada processo. Dessa forma, o usuário é responsável por definir o comportamento do sistema de acordo com estes identificadores. Observa-se que, caso estes não sejam utilizados na descrição do funcionamento do sistema, o MPI simplesmente executará processos idênticos, isto é, todos os processos seguiriam os mesmos passos. Assim, o estilo dos programas MPI é, geralmente, da forma:

[10]Em MPI, um identificador é comumente chamado de "rank".

Inicialização	Definição do identificador único do processo
Execução	Se sou o processo 0 faço isso, se sou o processo 1 faço aquilo, ...
Finalização	Finalização dos processos gerados pelo MPI

Dependendo do número de processos e da organização dos nodos, o gerenciamento das trocas de mensagens pode se tornar uma tarefa árdua para o usuário. No entanto, para no MaPI, o gerenciamento é feito de forma transparente. Tal característica é decorrente da estrutura adotada para a comunicação entre os processos.

A estrutura interna do MaPI é baseada na arquitetura cliente-servidor. O processo 0 representa o cliente e os demais processos são os servidores. O comportamento dos servidores é definido pela função de mapeamento. Dessa forma, o processo 0 (zero) executa o programa do usuário e os demais processos esperam uma chamada do *MapReduce* para executar a função de transformação. Note que tal estrutura, cliente-servidor, também é transparente para o usuário.

A função *init*, que inicializa o *framework*, é responsável por levar o curso do programa para o comportamento de servidor caso o identificador do processo seja diferente de 0 (zero). Outro fato que justifica a necessidade de inicialização do *framework* é que a comunicação entre dois processos é feita por troca de dados, não sendo possível o envio de funções. Assim, outro objetivo da inicialização é registrar as funções que serão executadas nos servidores de mapeamento.

A serialização de objetos é o processo de conversão de um objeto a uma sequência de bytes, assim como o processo que permite passar a sequência de bytes para um objeto utilizável e válido. Um dos pontos a se observar ao utilizar o *MapReduce* é que os parâmetros passados para o *map*, exceto o objeto compartilhado, devem ser do tipo *string*. Com isso, há a necessidade de serialização dos objetos que serão utilizados nos nós de processamento. Existem algumas linguagens, como o Java, que já possuem uma interface de serialização; assim, basta adicionar um comando ao código para que o objeto declarado passe a ser serializável. Porém, a linguagem C++ não oferece recursos de serialização automática de objetos, ficando esta tarefa a cargo do utilizador do *framework*.

4 ESTUDO DE CASO: MaPI APLICADO À OTIMIZAÇÃO

Especificamente para problemas de otimização, o modelo *MapReduce* pode ser simplificado, pois os itens a serem mapeados não necessitam ser ordenados ou agrupados por chaves e o que interessa é a solução e uma forma de se medir sua qualidade. Usuários especificam as funções *map* e *reduce*. A primeira é responsável por processar um item de um tipo α resultando em um item do tipo β e a segunda mescla todos os dados intermediários, representados por um conjunto de elementos do tipo β, gerado a partir da aplicação da função *map* a um conjunto de dados do tipo α.

A Figura 2(a) apresenta uma forma de se aplicar a abstração *MapReduce* a problemas de otimização, onde os tipos α e β representam soluções (s e s') e a função *map* pode ser qualquer método de refinamento, e o resultado da função *reduce*, s^*, é geralmente a melhor dentre as soluções intermediárias. Supondo que na situação apresentada nesta figura sejam feitas n aplicações da função de mapeamento (*map*), tal situação é reproduzida pelo MaPI por meio da criação de $n + 1$ processos. O processo adicional é responsável por fazer a chamada e a redução do *MapReduce*. A Figura 2(b) apresenta outra forma de se utilizar a abstração em algoritmos de otimização. Trata-se da exploração concorrente da vizinhança de uma solução. Neste caso, cada processo é responsável por analisar um sub-espaço da vizinhança de uma dada solução.

(a) Mapeamento de soluções (b) Mapeamento de sub-espaços da vizinhança

Figura 2: Abstração *MapReduce* em problemas de otimização

Para testar a aplicação da abstração *MapReduce* na paralelização de procedimentos de otimização, foi implementada uma versão paralela de um dos procedimentos mais utilizados na otimização por métodos heurísticos, o gerador de melhor vizinho.

4.1 PARALELIZAÇÃO DE HEURÍSTICAS DE REFINAMENTO

Uma das ideias mais importantes utilizadas pelos métodos heurísticos de otimização baseados em busca local é a noção de vizinhança. Todos os métodos heurísticos de busca local clássicos, bem como as metaheurísticas de busca local, exploram o espaço de busca por meio de vizinhança. Dentre as heurísticas clássicas destacam-se o Método de Descida (ou *Best Improvement Method*), o Método de Primeira Melhora (*First Improvement Method*) e o Método de Descida Randômica (*Random Descent Method*) [1]. Dentre as metaheurísticas, citamos: *Variable Neighborhood Search* [18–20], *Iterated Local Search* [21], *Tabu Search* [22], *Multi-Start* [23] e GRASP [24]. O refinamento de uma solução por qualquer desses métodos envolve a análise de seus vizinhos segundo a estrutura de vizinhança adotada. Assim, ao se paralelizar o gerador de melhor vizinho, quaisquer heurísticas ou metaheurísticas de busca local que o utilizem estarão automaticamente paralelizadas.

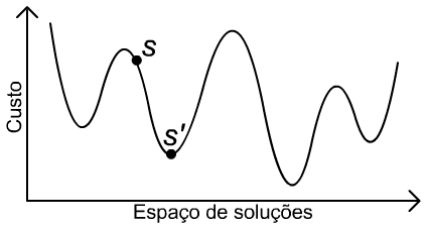
4.2 DESCIDA COM EXPLORAÇÃO PARALELA DE VIZINHANÇA

Devido à vasta utilização e aplicabilidade do Método da Descida, optamos por utilizá-lo para ilustrar o funcionamento do *framework*. A ideia desta técnica é partir de uma solução inicial qualquer e a cada passo analisar todos os seus possíveis vizinhos, movendo somente para aquele que representar a maior melhora no valor atual da função de avaliação. Pelo fato de analisar todos os vizinhos de uma dada estrutura de vizinhança N e escolher o melhor, esta técnica é comumente referenciada na literatura por *Best Improvement Method*. O pseudocódigo desse método é apresentado na Figura 3(a). Na Figura 3(b), é apresentada uma situação onde o Método da Descida parte de uma solução s e termina quando encontra um ótimo local s'.

Procedimento `Descida(`s, N`)`

$m \leftarrow melhorMovimento(s, N)$
enquanto m *melhorar a solução* s **faça**
  $s \leftarrow s \oplus m$;
  $m \leftarrow melhorMovimento(s, N)$;
fim
return s;

(a)



(b)

Figura 3: Método de Descida: (a) Pseudocódigo; (b) Representação no espaço de soluções

Como o espaço de busca de soluções é geralmente muito vasto, verificar todos os possíveis vizinhos de uma solução é uma tarefa computacionalmente árdua. Para agilizar esse processo propõe-se o uso de um gerador paralelo de melhor vizinho. Com isso, a cada passo, o espaço de soluções é dividido entre os nós de processamento, reduzindo-o em sub-espaços menores. O gerador foi desenvolvido usando-se o *framework* MaPI. Nesse caso, a entrada do *MapReduce* é uma lista de soluções, e a função de mapeamento recebe uma solução e um sub-espaço de uma vizinhança, e retorna o movimento que gerou o melhor vizinho da solução nesse sub-espaço. A saída do *MapReduce* é o melhor movimento gerado dentre todos os sub-espaços, sendo então aplicado à solução que deu início ao método.

O melhor vizinho de uma solução s é gerado pelo movimento m que possuir o valor mais favorável da função de avaliação na vizinhança de s. Como um vizinho pode ser gerado pela aplicação direta de um movimento, e a manipulação deste é computacionalmente menos custosa que a de um vizinho, optou-se por paralelizar o método `melhorMovimento`(), em vez de `melhorVizinho`(). A ideia é gerar o conjunto M de movimentos m que geram a vizinhança $N(s)$ de uma solução s e dividir esse conjunto M em $|P|$ subconjuntos disjuntos, $\{(s, M^1), \cdots, (s, M^{|P|})\}$), sendo $|P|$ a quantidade de processadores de mapeamento. O próximo passo é aplicar a função de mapeamento a cada subconjunto de forma a gerar seu melhor movimento. De posse das melhores soluções, a redução consiste simplesmente em escolher a melhor dentre essas. Além do subconjunto de movimentos possíveis, a função de mapeamento recebe também a solução s, para que essa função possa efetuar os testes necessários quanto à aplicação de cada movimento do subconjunto. O pseudocódigo do gerador paralelo de melhor movimento desenvolvido de acordo com a abstração *MapReduce* é apresentado a seguir. É importante observar que $N(s)$ deve ser enumerável. No procedimento `mapmm`, *critérioAceitação*(m^{*i}, m^i_k) retorna o melhor entre dois movimentos, sendo m^{*i} o melhor até o momento da i-ésima parte da vizinhança de s e m^i_k o movimento corrente nesta parte da vizinhança.

Procedimento `melhorMovimento(`s, N`)`

$M \leftarrow$ Conjunto de movimentos possíveis em $N(s)$;
Divida M em $|P|$ partes e seja M^i a i-ésima parte de M;
return $mapReduce(mapmm, reducemm, \{(s, M^1), ..., (s, M^{|P|})\})$;

Procedimento `mapmm(`s, M^i`)`

para *cada* $m^i_k \in M^i$ *com* $k = 1, ..., |M^i|$ **faça**
  Avalie a aplicação de m^i_k à solução s;
  $m^{*i} \leftarrow$ *critérioAceitação*(m^{*i}, m^i_k);
fim
$c^{*i} \leftarrow$ custo da aplicação de m^{*i} à solução s;
return $\big(m^{*i}, melhorCusto\big)$;

Procedimento `reducemm(`B`)`

Seja $B = \{(m^{*1}, c^{*1}), ..., (m^{*|P|}, c^{*|P|})\}$ o conjunto de elementos mapeados;
$m^* \leftarrow$ movimento m^{*i} tal que c^{*i} seja mínimo em B, $\forall i = 1, ..., |P|$;
return m^*;

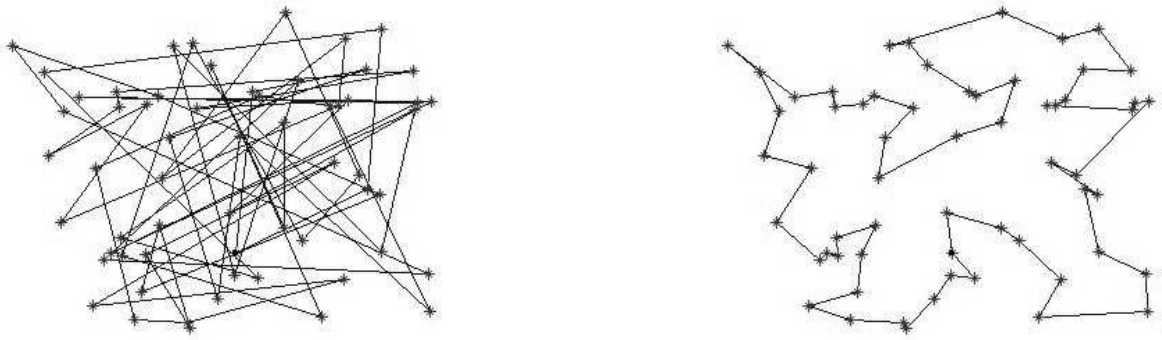
4.3 CENÁRIO DE TESTE: PROBLEMA DO CAIXEIRO VIAJANTE

O algoritmo paralelo desenvolvido na Seção 4.2 foi aplicado ao Problema do Caixeiro Viajante (PCV). O PCV é um dos problemas combinatórios mais estudados na literatura e pode ser descrito como segue. Dado um conjunto de n cidades e uma matriz de distâncias associada, um caixeiro viajante deve partir de alguma cidade, dita origem, passar por todas as outras restantes apenas uma vez, e retornar à cidade origem. O objetivo é encontrar a melhor rota para percorrer todas as cidades, ou seja, a rota de menor distância. Este problema vem sendo largamente estudado, e é utilizado como um *benchmark* para muitos dos algoritmos de otimização desenvolvidos. Por se enquadrar na classe de problemas NP-difíceis, é de difícil resolução utilizando métodos matemáticos. Portanto, na maioria das vezes, métodos heurísticos são propostos para a sua resolução.

Vários problemas da literatura podem ser modelados como um Problema do Caixeiro Viajante, como é o caso do Problema de Sequenciamento de Tarefas em Uma Máquina [25–28], em que n tarefas devem ser processadas sequencialmente em uma determinada máquina. O tempo de preparo para processar uma tarefa j imediatamente após uma tarefa j, conhecido como tempo de *setup*, é dado por s_{ij}. Esse tempo de preparo pode ser interpretado como a distância entre duas cidades do PCV, e um dos possíveis objetivos é encontrar a melhor sequência de tarefas, visando a minimização do tempo total de processamento.

Outra aplicação é o Problema de Roteamento de Veículos [29], em que clientes necessitam de uma certa quantidade de mercadoria e o fornecedor deve satisfazer todas as demandas com uma frota de caminhões. Nessa aplicação, há alguns problemas adicionais, como designar clientes para cada um dos caminhões da frota e encontrar um melhor distribuição na entrega das mercadorias para que a capacidade do caminhão não seja excedida. Cada rota que um caminhão deve fazer saindo da central de distribuição, entregando mercadorias e retornando ao estoque, pode ser resolvido como um PCV.

A Figura 4 apresenta dois exemplos de rotas para o PCV, onde o caixeiro percorre 60 cidades. A primeira rota, 4(a), foi gerada de forma aleatória. Resultados melhores podem ser obtidos ao se adicionar alguma inteligência ao método que gera a rota, como é exemplificado na Figura 4(b), na qual a distância percorrida pelo caixeiro é consideravelmente menor que na anterior.



(a) (b)

Figura 4: Exemplos de rotas para um Problema do Caixeiro Viajante

4.4 RESULTADOS COMPUTACIONAIS

Para testar o algoritmo paralelo desenvolvido, foram utilizados problemas-teste do PCV envolvendo as 50, 100 e 150 primeiras cidades da instância Mona Lisa[11], denotadas por ML1, ML2 e ML3. Todos os testes foram realizados em um PC Pentium Core 2 Quad (Q6600), com 2,4 GHz, 8 GB of RAM, sob sistema operacional Linux Ubuntu 8.10 Kernel 2.6.24-25. Vale ressaltar que os resultados aqui apresentados enfatizam apenas o sucesso da utilização do *framework*, sendo a qualidade da solução um objetivo secundário.

A Tabela 1 apresenta os resultados obtidos. As duas primeiras colunas indicam o problema-teste e o número correspondente de cidades. A terceira coluna indica o número de processos, apresentado na forma $1 + x$, sendo x o número de servidores de mapeamento. A quarta coluna mostra o tempo computacional demandado, em segundos. A última coluna apresenta a redução do tempo por problema-teste e número de processos. Para cada problema a redução r é calculada como segue: $r_p = (t_1 - t_p)/t_1$, onde t_1 é o tempo gasto utilizando-se apenas um processo de mapeamento e t_p é o tempo gasto utilizando p processos de mapeamento (neste caso são usados 1, 2 e 3 processos de mapeamento, isto é, $p = 1 \cdots 3$). A Figura 5 ilustra o desempenho do algoritmo diante do aumento do número de processos.

Pela Tabela 1 e Figura 1, verifica-se a diminuição do tempo gasto pelo algoritmo ao se aumentar o número de processos utilizados. À medida que o número de nós de processamento é aumentado, o tempo de resolução do problema diminui em até $58,1\%$ na segunda e terceira instâncias, e em $53,3\%$ na primeira. Isso mostra o ganho que a paralelização do algoritmo fornece. Observa-se também que, para cada problema-teste, o custo relativo à rota do caixeiro foi o mesmo, o que era de se esperar, uma vez que o algoritmo heurístico aplicado é determinístico.

Para implementar o algoritmo em questão, os desenvolvedores não tiveram nenhuma preocupação com a forma de comunicação entre os processos, com a sincronização dos dados ou mesmo com a topologia de rede. Toda a implementação foi feita

[11] Disponíveis no endereço *www.tsp.gatech.edu/data/ml/monalisa.html*

Tabela 1: Resultados obtidos em 1, 2 e 3 processos de mapeamento

Problema	Cidades	Processos	Custo	Tempo	Redução
ML1	50	1+1	157472	0,92	0,0%
ML1	50	1+2	157472	0,48	47,8%
ML1	50	1+3	157472	0,43	53,3%
ML2	100	1+1	251674	21,82	0,0%
ML2	100	1+2	251674	12,29	43,7%
ML2	100	1+3	251674	9,14	58,1%
ML3	150	1+1	378015	141,2	0,0%
ML3	150	1+2	378015	82,89	41,3%
ML3	150	1+3	378015	59,23	58,1%

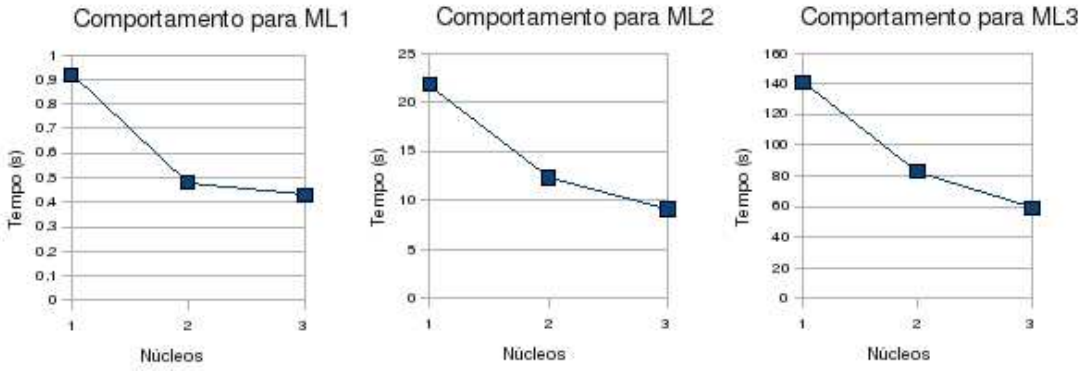


Figura 5: Desempenho do algoritmo em relação ao aumento de processos

de forma sequencial, sendo necessário aos desenvolvedores apenas conhecimentos de conceitos básicos de paralelismo para a divisão de vizinhança. Com isso, exige-se dos usuários do MaPI conhecimento e esforço de implementação muito menores do que aqueles necessários caso essa ferramenta não fosse utilizada. Isso ocorre pois o MaPI utiliza internamente as operações necessárias de trocas de mensagens entre os processos, favorecendo aos desenvolvedores concentrarem-se apenas em aspectos relevantes do algoritmo e não em detalhes da programação paralela. Além disso, MaPI implementa a abstração *MapReduce*, que é de fácil entendimento, principalmente por ter sua computação expressa por meio da implementação de apenas duas funções.

5 CONCLUSÕES E TRABALHOS FUTUROS

Apesar das vantagens de um sistema paralelo, o principal desmotivador à construção de tais sistemas é a dificuldade de implementação. Este trabalho atua no sentido de motivar o desenvolvimento de algoritmos paralelos e o uso das facilidades atuais quanto à montagem de *clusters*. Para isso, foi proposto o MaPI, um *framework* que implementa a abstração *MapReduce* na linguagem C++. Embora o MaPI possa ser aplicado a diversas classes de problemas, neste trabalho o foco foi simplificar a implementação de algoritmos paralelos de otimização e a resolução de tais problemas em um *cluster*.

Ao utilizar o MaPI, o usuário é capaz de implementar uma aplicação paralela sem se preocupar com a forma de comunicação entre os processos ou como o sistema fará a paralelização. Além disso, toda a implementação feita pelo usuário é sequencial. Portanto, o objetivo de simplificar a implementação de sistemas paralelos foi alcançada.

Para testar a aplicação da abstração *MapReduce* na paralelização de procedimentos de otimização, foi implementada uma versão paralela de um dos procedimentos mais utilizados na otimização por métodos heurísticos, o gerador de melhor vizinho. Para testá-lo, foi implementado o método de Descida usando o gerador de melhor vizinho paralelo. Tal procedimento foi aplicado a um problema clássico, o Problema do Caixeiro Viajante. Os resultados obtidos comprovam a eficiência *framework* como ferramenta de auxílio ao desenvolvimento de procedimentos paralelos de otimização.

Como trabalho futuro propõe-se a análise de implementações sequencial e distribuída de outros procedimentos de otimização, tanto baseados em busca local quanto em algoritmos evolutivos. Propõe-se, também, a utilização do MaPI na resolução de outros problemas, fora do contexto de otimização. Além disso, outro passo será o projeto de um *framework* conceitual baseado no MaPI, com o objetivo de padronizar e facilitar sua implementação tanto em outras linguagens quanto em outras arquiteturas, como GPU[12] e Cell[13]. Outro ponto é a introdução de tolerância a falhas.

[12]GPU (*Graphics Processing Unit*) é um microprocessador inicialmente projetado para processamento gráfico. Atualmete, devida à sua flexibilidade, este vem sendo usado para outros propósitos. Saiba mais em *http://www.lcg.ufrj.br/Cursos/GPUProg*

[13]Cell é uma arquitetura de microprocessador desenvolvida em conjunto pela Sony, Toshiba e IBM. Saiba mais em *http://www.ibm.com/developerworks/power/cell/*

Agradecimentos

Os autores agradecem à FAPEMIG, CAPES, CNPq e à Universidade Federal de Ouro Preto, pelo apoio ao desenvolvimento do presente trabalho.

Referências

[1] P. Hansen and N. Mladenović. "First vs. best improvement: An empirical study". *Discrete Applied Mathematics*, vol. 154, no. 5, pp. 802–817, 2006.

[2] G. Dantzig, R. Fulkerson and S. Johnson. "Solution of a large-scale traveling-salesman problem". *Operations Research*, vol. 2, pp. 393–410, 1954.

[3] T. Bektas. "The multiple traveling salesman problem: an overview of formulations and solution procedures". *Omega*, vol. 34, no. 3, pp. 209–219, 2006.

[4] M. G. C. Resende and C. C. Ribeiro. "Parallel Greedy Randomized Adaptive Search Procedures". In *Parallel Metaheuristics: A New Class of Algorithms*, edited by E. Alba. John Wiley and Sons, 2005.

[5] R. B. Muhammad. "A Parallel Local Search Algorithm for Euclidean Steiner Tree Problem". *Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06)*, pp. 157–164, 2006.

[6] G. D. Campos, H. T. Y. Yoshizaki and P. P. Belfiore. "Algoritmo Genético e computação paralela para problemas de roteirização de veículos com janelas de tempo e entregas fracionadas". *Gestão e Produção*, vol. 13, pp. 271–281, 2006.

[7] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] R. Lämmel. "Google's MapReduce programming model – Revisited". *Sci. Comput. Program.*, vol. 68, no. 3, pp. 208–237, 2007.

[9] T. Hoefler, A. Lumsdaine and J. Dongarra. "Towards Efficient MapReduce Using MPI". In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 240–249, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] B. Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991.

[11] A. N. Langville and C. D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, USA, 2006.

[12] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng and K. Olukotun. "Map-Reduce for Machine Learning on Multicore". In *NIPS*, edited by B. Schölkopf, J. C. Platt and T. Hoffman, pp. 281–288. MIT Press, 2006. Disponível em http://www.cs.stanford.edu/people/ang//papers/nips06-mapreducemulticore.pdf.

[13] K. Cardona, J. Secretan, M. Georgiopoulos and G. Anagnostopoulos. "A Grid Based System for Data Mining Using MapReduce." Technical Report TR-2007-02, AMALTHEA, 2007.

[14] A. Kimball, S. Michels-Slettvet and C. Bisciglia. "Cluster computing for web-scale data processing". In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pp. 116–120, New York, NY, USA, 2008. ACM.

[15] J. Umemoto, K. Iwama, D. Kawai, S. Miyazaki and Y. Okabe. "Parallelizing Local Search for CNF Satisfiability Using PVM". In *Proceedings of the AAAI-2000 workshop on parallel and distributed search for reasoning*, Austin, USA, 2000.

[16] V. J. Garcia, A. S. Mendes and P. M. França. "Algoritmo Memético Paralelo aplicado a problemas de Sequenciamento em Máquina Simples". In *Anais do XXXIII Simposio Brasileiro de Pesquisa Operacional*, pp. 971–981, Campos do Jordão, SP, 2001.

[17] N. B. Standerski. "Aplicação de Algoritmos Genéticos Paralelos a Problemas de Grande Escala de VMI - Vendor Managed Inventory". In *Anais do XXXIV Simpósio Brasileiro de Pesquisa Operacional – SBPO*, pp. 1183–1195, Fortaleza, Brasil, 2003.

[18] N. Mladenović and P. Hansen. "Variable neighborhood search". *Computers and Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[19] P. Hansen and N. Mladenovic. "Variable neighborhood search: Principles and applications". *European Journal of Operational Research*, vol. 130, pp. 449–467, 2001.

[20] P. Hansen, N. Mladenovic and J. A. M. Pérez. "Variable neighborhood search". *European Journal of Operational Research*, vol. 191, pp. 593–595, 2008.

[21] H. R. Lourenço, O. C. Martin and T. Stützle. "Iterated Local Search". In *Handbook of Metaheuristics*, edited by F. Glover and G. Kochenberger, chapter 11. Kluwer Academic Publishers, Boston, 2003.

[22] F. Glover and F. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[23] R. Martí. "Multi-Start Methods". In *Handbook of Metaheuristics*, edited by F. Glover and G. Kochenberger, chapter 12. Kluwer Academic Publishers, Boston, 2003.

[24] M. G. C. Resende and C. C. Ribeiro. "Greedy randomized adaptive search procedures". In *Handbook of Metaheuristics*, edited by F. Glover and G. Kochenberger, chapter 8. Kluwer Academic Publishers, Boston, 2003.

[25] A. Allahverdi, J. N. Gupta and T. Aldowaisan. "A review of scheduling research involving setup considerations". *Omega, International Journal of Management Science*, vol. 27, pp. 219–239, 1999.

[26] G. Wan and B. P.-C. Yen. "Tabu search for single machine scheduling with distinct due windows and weighted earliness/tardiness penalties". *European Journal of Operational Research*, vol. 142, pp. 271–281, 2002.

[27] A. C. Gomes Júnior, C. R. V. Carvalho, P. L. A. Munhoz and M. J. F. Souza. "Um método heurístico híbrido para a resolução do problema de sequenciamento em uma máquina com penalidades por antecipação e atraso da produção". *Anais do XXXIX Simpósio Brasileiro de Pesquisa Operacional – SBPO*, vol. 1, pp. 1649–1660, 2007.

[28] M. J. F. Souza, P. H. V. Penna and F. A. C. A. Gonçalves. "GRASP, VND, Busca Tabu e Reconexão por Caminhos para o problema de sequenciamento em uma máquina com tempos de preparação dependentes da seqüência da produção, janelas de entrega distintas e penalidades por antecipação e atraso da produção". *Anais do XL Simpósio Brasileiro de Pesquisa Operacional – SBPO*, vol. 1, pp. 1320–1331, 2008.

[29] B. L. Golden, S. Raghavan and E. A. Wasil, editors. *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *Operations Research/Computer Science Interfaces Series*. Springer, 2008.