Anais do IX Congresso Brasileiro de Redes Neurais / Inteligência Computacional (IX CBRN)
Ouro Preto 25-28 de Outubro de 2009

MAPI: UM *FRAMEWORK* PARA PARALELIZAÇÃO DE ALGORITMOS DE OTIMIZAÇÃO

Sabir Ribas*, Mário Henrique de Paiva Perché*, Igor Machado Coelho*, Pablo Luiz Araújo Munhoz*, Marcone Jamilson Freitas Souza*, Andre Luiz Lins Aquino*

* *Universidade Federal de Ouro Preto*
Ouro Preto, Minas Gerais, Brasil

Emails: {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 é seqüencial. Para ilustrar o funcionamento do *framework*, este foi aplicado a um problema clássico da 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— Algoritmos Paralelos, *Framework*, Otimização, Heurísticas.

1 Introdução

A computação paralela tem provocado grande impacto nas mais diversas áreas do conhecimento. Tais impactos vão desde simulações computacionais para o meio científico, até aplicações de engenharia, aplicações comerciais e linhas de produção. A computação paralela opera sob o princípio de que problemas de grande porte geralmente podem ser divididos em problemas menores, que são resolvidos concorrentemente, em paralelo. Apesar das vantagens da computação paralela sua principal desvantagem é a dificuldade de implementação. Se comparados com sistemas seqüenciais, 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.

Considerando as aplicações que envolvem linhas produtivas, muitos problemas encontrados são de natureza combinatória e, como tal, o desafio é encontrar boas soluções, muitas vezes paralelas. Para resolver essa classe de problemas destacam-se duas abordagens: a heurística e a exata. A vantagem de se adotar metodologias exatas é a garantia da obtenção das soluções ótimas para o problema, o que não acontece no caso de heurísticas. Porém, o uso de heurísticas é muito atraente quando a prioridade é o tempo de resposta. Nesse contexto, apresentamos o MaPI, uma implementação em C++ da abstração *MapReduce*, aplicado para a implementação de algoritmos paralelos de otimização, mais especificamente problemas de natureza combinatória. Tal especificidade não restringe a aplicabilidade do MaPI que pode ser aplicado a diversas classes de problemas.

De forma geral, o objetivo do MaPI é fornecer ao usuário todo o aparato necessário para a implementação rápida de procedimentos paralelos. O sucesso do *framework* se dará quando um usuário inexperiente quanto a programação distribuída implementar uma aplicação paralela sem se preocupar com a forma de comunicação entre os processos ou como o sistema fará a paralelização. Assim, toda a implementação feita pelo usuário é seqüencial.

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. Na quarta seção são apresentados os resultados obtidos. Por fim, a quinta seção apresenta as conclusões.

2 Trabalhos Relacionados

MapReduce é uma abstração simples e poderosa, geralmente aplicada ao processamento ou geração de grandes massas de dados. Dean and Ghemawat (2004) apresentam uma visão geral sobre a implementação do *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 (Lämmel, 2007): (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 pro-

duz 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 aquela chave. Essa função junta esses valores para formar um conjunto possivelmente menor de valores. Tipicamente, zero ou apenas um valor é produzido pela função *reduce*.

Especificamente para os problemas de otimização tratados aqui o modelo 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.

Considerando a paralelização de heurísticas, 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 (Umemoto et al., 2000), (Garcia et al., 2001) e (Standerski, 2003).

Chu et al. (2009) apresenta versões paralelas baseadas em *MapReduce* de uma variedade de algoritmos de aprendizagem de máquina incluindo regressão logística, *Naive Bayes* e *backpropagation*. Tal trabalho explora ambientes *multicore*.

Cardona et al. (2007) usa 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 *MapReduce*.

3 Solução Implementada

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.

Em Haskell, uma linguagem funcional, *map* é uma função de ordem superior cujo tipo é apresentado a seguir, de acordo com a notação de Haskell:

```
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 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.

3.1 MaPI: Primeiros Passos

A idéia básica por trás do MaPI é mapear e reduzir, distribuindo o processamento e capturando os resutados. 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, onde ($[b] \rightarrow [c]$) 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*.

A seguir são apresentados os passos para se implementar um sistema paralelo usando o MaPI. Tal sistema visa a aplicação da abstração *MapReduce* a um conjunto de *strings*. A função de mapeamento recebe uma *string* e a ela adiciona o trecho "(mapeada)". A função de redução retorna a primeira *string* do conjunto mapeado.

Para seguir este exemplo deve-se ter instalada alguma implementação do MPI. Em nossos testes foi utilizada a implementação OpenMPI, que pode ser obtida pelo 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".

- Primeiramente, crie um arquivo ".cpp", doravante "ex01.cpp", e inclua o cabeçalho da biblioteca MaPI.

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

- No 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.h"
int main(int argc, char** argv)
{
    vector<string> input;
    input.push_back(string("Entrada 1 "));
    input.push_back(string("Entrada 2 "));
    printv(input);
}
```

- Se estiver tudo certo, ao compilar e executar o programa pela linha de comando abaixo:

```
mpiCC ex01.cpp -o ex01 && mpirun -np 3 ex01
```

onde "-np 3" significa que o mpirun criará três processos.

- 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 st)
      { return st + "(mapeada) "; }
```

- A aplicação da função *fmap* a cada um dos elementos da entrada 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.

```
vector<string> * freduce
            ( vector<string> * mapped )
{
    vector<string> * reduced;
    reduced = new vector<string>;
    reduced->push_back(mapped->at(0));
    return reduced;
}
```

- Os próximos passos são a inicialização e a finalização do *framework* pelas funções *MRInit* e *MRFinalize*, respectivamente. Por fim, tem-se a chamada do *MapReduce*. A seção 3.2 detalha a estrutura do *framework* e justifica sua inicialização e a finalização. O código completo do exemplo é apresentado a seguir:

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

string fmap(string st)
      { return st + "(mapeada) "; }

vector<string> * freduce
            ( vector<string> * mapped )
{
    vector<string> * reduced;
    reduced = new vector<string>;
    reduced->push_back(mapped->at(0));
    return reduced;
}

int main(int argc, char** argv)
{
  // Inicialização do Framework
  MRInit(argc,argv,fmap);

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


  // Chamada do MapReduce
  vector<string>*output
       = mapreduce(fmap,freduce,&input);

  MRFinalize(); // Finalização do Framework
}
```

- Executando o comando de compilação anteriormente apresentado, o programa supra produzirá a seguinte saída:

```
Entrada 1
Entrada 2
== Mapper ==
Entrada 1 (mapeada)
Entrada 2 (mapeada)
== Reducer ==
Entrada 1 (mapeada)
```

O resultado apresentado pelo programa consiste na exibição dos dados de entrada, do vetor mapeado e do resultado da redução.

3.2 Estrutura do MaPI

Como o MPI inicializa vários processos idênticos, a não ser por seus identificadores, o usuário é responsável por definir qual será o comportamento do sistema de acordo com os identificadores. Caso os identificadores não sejam utilizados o MPI simplesmente executará processos idênticos. Um nome muito utilizado para se referir aos identificadores é "rank". Assim, o estilo dos programas MPI geralmente é da forma:

Inicialização: Nesta etapa é definido o identificador único do processo.

Execução: Se sou o processo 0 faço isso, se sou o processo 1 faço aquilo, etc.

Finalização: Finaliza os processo 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 MaPI, o gerenciamento é feito de

forma transparente para o usuário. 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 *MRInit*, 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 executarão no servidor. Sem este registro, não seria possível obter o grau de abstração do *MapReduce* pois o usuário teria que especializar o *framework*, gerando uma implementação do *MapReduce* para cada um de seus problemas.

4 Resultados

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. O melhor vizinho de uma solução s é gerado pelo movimento m que retorna o valor mais favorável da função de avaliação na vizinhança de s. Ao se paralelizar tal procedimento, quaisquer heurísticas ou metaheurísticas que os utilizem serão automaticamente paralelizadas. Como um vizinho pode ser gerado pela aplicação direta de um movimento, e a manipulação de um movimento é computacionalmente mais barata que a de um vizinho, optou-se por paralelizar o método $melhorMovimento()$, em vez de $melhorVizinho()$. Seu pseudocódigo é apresentado a seguir.

Procedimento `Melhor Movimento com MapReduce(s, N)`

$M \leftarrow$ Movimentos derivados de $N(s)$;
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)`

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

Procedimento `reducemm(B)`

Seja $B = \{(m_1, c_1), ..., (m_{|B|}, c_{|B|})\}$ o conjunto de elementos mapeados;
$m^* \leftarrow$ movimento m_k tal que c_k seja mínimo em B, $\forall k = 1, ..., |B|$;
return m^*;

Para a validação do uso do MaPI em procedimentos de otimização, foi implementado o método de descida usando o gerador de melhor vizinho paralelo supra apresentado. O pseudocódigo desse método é apresentado a seguir.

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;

Tal procedimento foi aplicado em um problema clássico da otimização, o Problema do Caixeiro Viajante. Foram utilizados como problemas-teste as 50, 100 e 150 primeiras cidades da instância Mona Lisa, disponível no endereço www.tsp.gatech.edu/data/ml/monalisa.html. A Tabela 1 apresenta os resultados obtidos pela aplicação da descida nos problemas-teste denominados ML1, ML2 e ML3. Os valores da coluna "Processos" são da forma $1 + x$, onde x é o número de servidores de mapeamento. A Figura 1 apresenta o comportamento do tempo gasto pelo método diante do aumento no número de processos.

Tabela 1: Resultados obtidos em 1, 2 e 3 núcleos

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

Pela Tabela 1 e Figura 1, verifica-se a diminuição do tempo gasto pelo método ao se aumentar o número de processos utilizados. A medida que aumentamos os nós de processamento, o tempo de resolução do problema diminui em até 58% na segunda e terceira instâncias, e em 53% na primeira. Isso demonstra o ganho que a paralelização do algo-

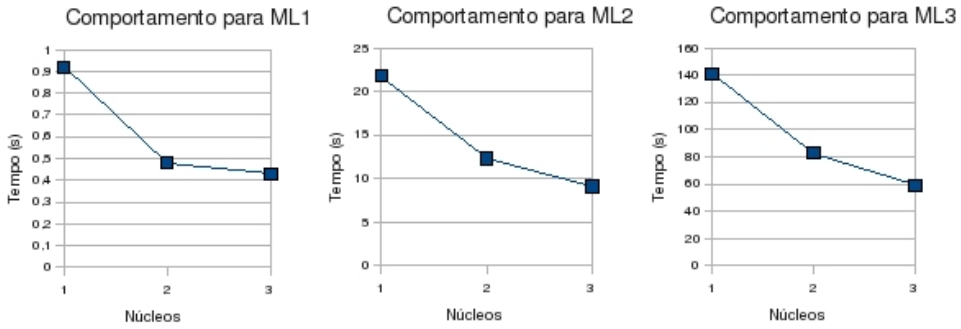

Figura 1: Comportamento do método em relação ao aumento de processos

ritmo fornece. Observa-se também que, para cada problema-teste, o custo relativo à rota do caixeiro foi o mesmo. Isso ocorre pois o algoritmo aplicado é deterministico.

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, o foco é 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.

Referências

Cardona, K., Secretan, J., Georgiopoulos, M. and Anagnostopoulos, G. (2007). A grid based system for data mining using mapreduce. technical report tr-2007-02, *Technical report*, AMALTHEA.

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A. Y. and Olukotun, K. (2009). Map-reduce for machine learning on multicore. disponível em <highscalability.com/map-reduce-machine-learning-multicore>.

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters, *In OSDI'04, 6th Symposium on Operating Systems Design and Implementation, Sponsored by USENIX, in cooperation with ACM SIGOPS* pp. 137–150.

Garcia, V. J., Mendes, A. S. and França, P. M. (2001). Algoritmo memetico paralelo aplicado a problemas de sequenciamento em maquina simples, *Anais do XXXIII Simposio Brasileiro de Pesquisa Operacional*, Campos do Jordão, SP, pp. 971–981.

Lämmel, R. (2007). Google's mapreduce programming model — revisited, *Sci. Comput. Program.* **68**(3): 208–237.

Standerski, N. B. (2003). Aplicação de algoritmos genéticos paralelos a problemas de grande escala de vmi - vendor managed inventory, *Anais do XXXIV SBPO*, Fortaleza, Brasil, pp. 1183–1195.

Umemoto, J., Iwama, K., Kawai, D., Miyazaki, S. and Okabe, Y. (2000). Parallelizing local search for cnf satisfiability using pvm, *Proc. of the AAAI-2000 workshop on parallel and distributed search for reasoning.*, Austin, USA.