Uma introdução ao Apache Hama

Thiago Kenji Okada[1]

[1]Instituto de Matemática e Estatística (IME) – Universidade de São Paulo (USP)
1010, Rua do Matão, Cidade Universitária
São Paulo, SP, Brasil
CEP: 05508-090
Telefone: +55-011-3091-6101

thiagoko@ime.usp.br

Abstract. *Nesse artigo, iremos apresentar uma breve introdução ao Apache Hama, um* framework *que permite criar aplicações baseadas no modelo* Bulk Synchronous Parallel *(BSP) altamente escaláveis num ambiente de nuvem computacional. Iremos explicar o que é o modelo BSP, como o modelo BSP é reproduzido internamente no Apache Hama, suas funcionalidades e aplicações.*

1. Introdução

Com o crescimento da computação em nuvem, que permitiu seus usuários acessarem uma quantidade aparentemente infinita de recursos computacionais sobre demanda, e a explosão de dados gerados no mundo contemporâneo, era necessário a criação de um novo modelo que permitisse o processamento desse enorme volume de dados de forma massivamente distribuída.

Quando o Google apresentou o primeiro artigo sobre o modelo MapReduce em 2004 [6], ele permitiu uma revolução na maneira de processar os dados. Apesar de sua implementação original ser proprietária dentro dos servidores do Google, diversas outras implementações desse modelo apareceram a partir do artigo original, sendo o Apache Hadoop, iniciado pela Yahoo!, uma das implementações mais usadas [7].

Porém, com a popularização desse modelo, alguns problemas foram encontrados, principalmente em aplicações comuns como processamento de matrizes e grafos. Para resolver esses problemas, diversos outros modelos foram propostos, como o Google Pregel [16]. Nesse artigo, vamos apresentar um desses novos modelos de computação otimizados para matrizes e grafos, conhecido como Apache Hama.

O Apache Hama é um *framework* de propósito geral, escrito em Java, baseado no modelo *Bulk Synchronous Parallel* (BSP), que roda em cima do Apache Hadoop e permite a criação de algoritmos massivamente paralelos para computação científica, em diversas áreas, como matrizes, grafos e aprendizado de máquina. Desde 2012, o Apache Hama é um dos projetos de alta prioridade da Fundação Apache [2].

Na Seção 2 iremos mostrar uma breve introdução ao modelo BSP, na Seção 3 iremos mostrar a arquitetura do Apache Hama, como um código no Apache Hama é estruturado e um exemplo de código comentado. Na Seção 4, veremos algumas aplicações do Apache Hama encontradas na literatura. Finalmente, na Seção 5, tiraremos algumas conclusões sobre o estudo desse *framework*.

2. O modelo BSP

Modelos de computação paralela são um tópico ativo e recorrente de pesquisa [8, 13, 18]. Seu principal objetivo é prover um padrão para descrever e analisar o desempenho de aplicações paralelas. Para seu sucesso como modelo de computação paralela, é necessário que o modelo considere as características do *hardware* em uso.

Um dos modelos mais bem estabelecidos para a computação paralela é o *Bulk Synchronous Paralell* (BSP), introduzido originalmente por Valiant em 1990 [20]. Seu principal objetivo é prover um modelo que represente diferentes arquiteturas paralelas bem, sem considerar os detalhes de baixo nível. O modelo BSP une características essenciais de diferentes tipos de arquituras ao combinar três atributos:

- um conjunto de processadores, cada um com sua memória local;
- um roteador, que permite transmissão de mensagens entre processadores;
- um mecanismo de sincronização entre todos os processadores.

A computação é organizada em uma sequencia de *super-passos* (do original em inglês *supersteps*), cada uma dividida em três fases. Na Figura 1 temos uma ilustração de um super-passo. Na primeira fase, todos os processos fazem computação local nos dados. Na segunda fase temos a comunicação entre processos, onde cada nó pode trocar informações um com os outros. Finalmente, a última fase, consiste em uma barreira de sincronização global, que garante que todos os nós estão prontos para o próximo super-passo. Não existem restrições de quando receber mensagens, mas todas elas tem que ser recebidas até a barreira de sincronização. No modelo original, o primeiro e segundo passo podem acontecer de forma simultânea.

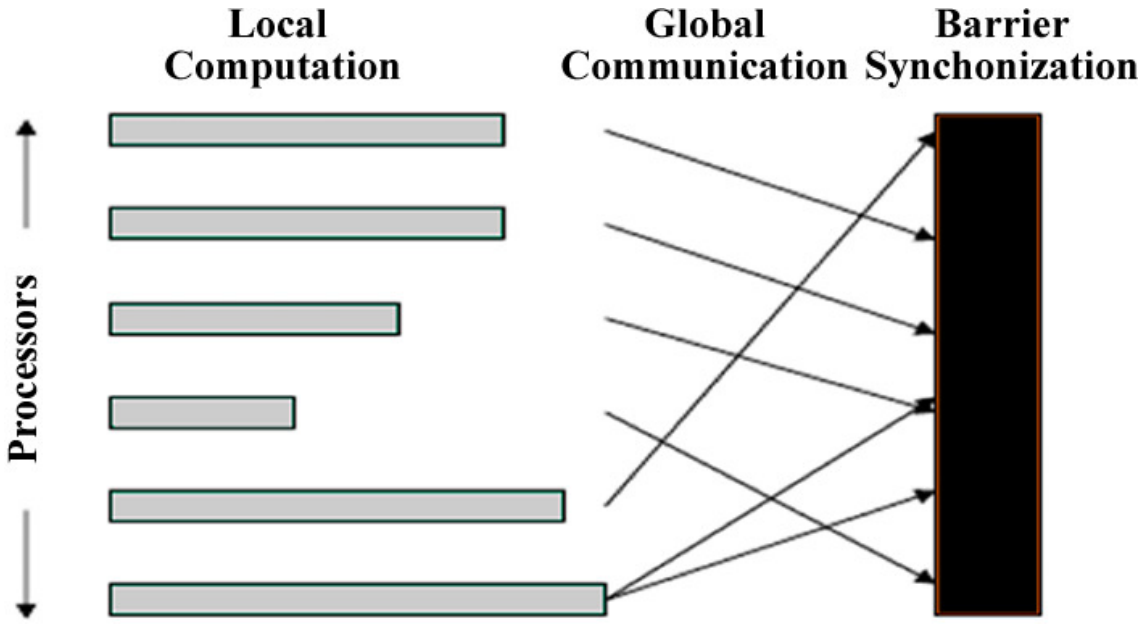

Figura 1: Superpasso no modelo BSP.

O modelo BSP é usado em diferentes contextos. O modelo foi aplicado para desenvolver algoritmos em diferentes tipos de arquiteturas, com desempenho garantido [15, 9, 5]. Modelar uma tarefa usando o modelo é útil, pois facilita a implementação e permite a previsão do seu desempenho.

Além do Apache Hama, existem diversas outras ferramentas que implementam o modelo BSP para facilitar a programação. Por exemplo, existe o BSPlib [11] para sistemas de memória tanto compartilhada como distribuída, e mais recentemente o BSGP implementou o modelo BSP em GPUs [12]. Também é possível implementar programas usando o modelo usando APIs mais genéricas como o *Message Passing Interface* (MPI) [4], basta a API oferecer as primitivas de comunicação e sincronização necessárias pelo modelo.

3. O Apache Hama

A Figura 2 ilustra a arquitetura geral do Apache Hama. O *Hama Core* provê várias primitivas de computação para matrizes e grafos e seleciona o motor de computação apropriado, *Hama Shell* provê um console interativo de usuário, enquanto o *Hama API* dá acesso a ambos de forma programática [17].

O Hama suporta três motores de computação, o *MapReduce* do Hadoop, um motor *BSP* desenvolvido para o próprio Apache Hama e o *Dryad* da Microsoft. Enquanto o MapReduce é usado para computação de matrizes, o BSP e o Dryad são usado para grafos. O Apache Hama também faz uso do Zookeeper para gerenciar metadados e operações de controle de forma atômica (por exemplo, a barreira global de sincronismo), além do Apache HDFS (*Hadoop Distributed File System*) e o Apache HBase (banco de dados não-relacional) para armazenamento de dados [17].

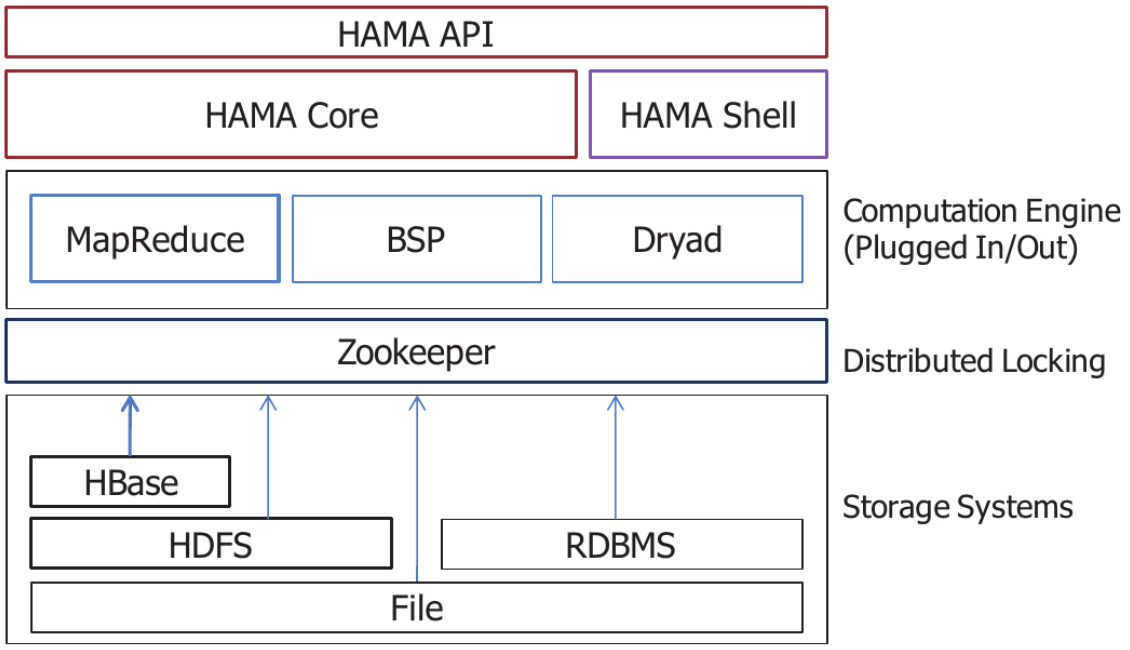

Figura 2: Arquitetura geral do Apache Hama. [17]

De acordo com Seo et al. [17], as principais contribuições do Hadoop são as seguintes:

- **Compatibilidade:** é possível usar qualquer uma das funcionalidades presentes no Hadoop, já que as interfaces entre o Hama e o Hadoop são as mesmas;

- **Escalabilidade:** já que as interfaces são as mesmas, o Hama pode rodar nas mesmas plataformas do Hadoop, como Amazon EC2, sem modificações;
- **Flexibilidade:** como mostrado no parágrafo anterior, é possível ter diversos motores de processamento, basta implementar as interfaces disponíveis;
- **Aplicabilidade:** as primitivas disponíveis do Hama podem ser aplicados em aplicações envolvendo gráficos e matrizes, como veremos mais adiante.

Nas próximas subseções, falaremos com mais detalhes da estrutura de um programa escrito no Apache Hama.

3.1. A estrutura básica de uma aplicação no Apache Hama

O Apache Hama é escrito na linguagem de programação Java. No Apache Hama você pode implementar seu próprio método BSP estendendo a classe `org.apache.hama.bsp.BSP`. Ela oferece um método chamado `bsp()` que pode ser usado para implementar um programa seguindo o modelo BSP. Também existem as funções `setup()` e `cleanup()`, chamadas antes e depois da função principal, respectivamente [3].

O método `bsp()` é definido pelo usuário e gerencia toda a parte paralela do programa. Ele contém apenas um argumento, um objeto da classe `BSPPeer`, que contém as interfaces de comunicação, contadores e entrada/saída do seu programa [3].

Como dito na Seção 2, cada programa BSP consiste em uma sequência de super-passos. Cada super-passo consiste em uma das seguintes fases:

- computação local;
- comunicação entre processos;
- barreira de sincronização.

Note que no Apache Hama, diferente do modelo original proposto por Valiant, cada fase tem que ser executado na ordem acima. A comunicação entre nós é feita durante a barreira de sincronização [3].

Existem diversas primitivas de comunicação no Apache Hama, que são derivadas de bibliotecas BSP já existentes, além de algumas primitivas extras exclusivas do Hama. Na Tabela 1, temos todas as funções de primitivas de comunicação do Apache Hama. Como veremos a seguir, atualmente, elas são métodos da classe `BSPPeer` e podem ser chamados em qualquer parte do método `bsp()` [3].

3.2. Um exemplo na prática

Em Código 1 temos um exemplo de código do método `bsp()` no Apache Hama. Apesar de um exemplo simples, vemos como o modelo BSP é aplicado dentro do Apache Hama.

Nas linhas 4-5 temos a fase de computação do modelo. Nesse caso, o programa não faz nada de útil além de incrementar uma variável `i` até o valor 100, porém poderia ser feito algo como a computação do valor local de uma matriz, por exemplo. Nas linhas 8-10, temos a comunicação entre os nós. Nesse programa são usadas várias primitivas de comunicação. O método `getAllPeerNames()` retorna todos os nós. Enviamos uma mensagem para cada nó usando o método `send()`, que recebe dois parâmetros: o nó em que a mensagem foi enviada e a mensagem de fato, que pode ser de diversos tipos

Tabela 1: Métodos para comunicação entre processos do Apache Hama.

Função	Descrição
send(String peerName, BSPMessage msg)	Envia uma mensagem para outro nó.
getCurrentMessage()	Recebe uma mensagem da fila.
getNumCurrentMessages()	Obtém o número de mensagens atuais da fila.
sync()	Inicia a barreria de sincronismo.
getPeerName()	Obtém o nome do nó da tarefa atual.
getPeerName(int index)	Obtém o nome do nó de índice n.
getNumPeers()	Obtém o número de nós.
getAllPeerNames()	Obtém o nome de todos os nós, incluindo o atual, em ordem crescente.

(nesse caso, um `Text`, que permite o envio de *strings*. Finalmente, chamando o método `getPeerName()` sem parâmetros, conseguimos o nome do nó atual.

Por último, na linha 13 temos a barreira de sincronismo. Esse programa só tem uma iteração, e por isso a barreira de sincronismo é executada uma única vez, mas esses três passos poderiam estar dentro de um laço, por exemplo. Finalmente imprimimos o resultado nas linhas 16-20. Interessante notar, como explicado no final da Seção 3.1, que a comunicação de fato só é feita após a barreira de sincronismo. Se movêssemos a chamada ao método `sync()` para antes da chamada ao `send()`, o resultado da linha 20 seria `null`.

Código 1: Um simples ”hello world”no Apache Hama.

```
1  @Override
2  public void bsp(BSPPeer<NullWritable, NullWritable, Text, Text,
       Text> peer) throws IOException, SyncException,
       InterruptedException {
3      // Computação
4      int i;
5      for (i = 0; i < 100; ++i);
6
7      // Comunicação
8      for (String peerName : peer.getAllPeerNames()) {
9          peer.send(peerName, new Text("Hello from " +
               peer.getPeerName() + ", i=" + i));
10     }
11
12     // Barreira de sincronismo
13     peer.sync();
14
15     // Imprime a mensagem de um nó qualquer
16     Random gen = new Random();
17     for (i = 0; i < gen.nextInt(peer.getNumPeers()) - 1; ++i) {
18         peer.getCurrentMessage();
19     }
20     System.out.println(peer.getCurrentMessage());
```

```
21 }
```

Em Código 2, temos um exemplo de como criar uma `main()` que execute a tarefa criada anteriormente. Para executar nosso código criado anteriormente, instanciamos dois objetos, um da classe `HamaConfiguration` e outro da classe `BSPJob`. Usamos o método `setJobName` para escolher a classe do Código 1. O uso do método `setJobName` é opcional, porém o `FileOutputFormat.setOutputPath` não, ou o nosso programa soltará uma exceção (veremos na Seção 3.3 para que ele serve).

De resto, criamos um *cluster* de máquinas (nesse caso locais) para rodar nosso exemplo. Nesse caso, temos apenas uma tarefa BSP usando todos os recursos disponíveis na máquina, mas poderíamos lançar diversas tarefas distintas. Ao executar o método `waitForCompletion(true)`, nosso programa executa até todas as tarefas BSP terminarem. A saída do nosso programa, excluindo as mensagens de depuração, pode ser visto em Código 3.

Código 2: Exemplo de main() num programa escrito para o Apache Hama.

```
1  public static void main(String[] args) throws IOException,
       ClassNotFoundException, InterruptedException {
2      HamaConfiguration conf = new HamaConfiguration();
3
4      BSPJob bsp = new BSPJob(conf, Hello.class);
5      bsp.setJobName("Hello World");
6      bsp.setBspClass(Hello.class);
7      FileOutputFormat.setOutputPath(bsp, new Path("/tmp/hello"));
8
9      BSPJobClient jobClient = new BSPJobClient(conf);
10     ClusterStatus cluster = jobClient.getClusterStatus(true);
11     bsp.setNumBspTask(cluster.getMaxTasks());
12     bsp.waitForCompletion(true);
13 }
```

Código 3: Saída do programa usado no exemplo.

```
1  Hello from local:2, i=100
2  Hello from local:0, i=100
3  Hello from local:4, i=100
4  Hello from local:2, i=100
5  Hello from local:7, i=100
6  Hello from local:1, i=100
7  Hello from local:18, i=100
8  Hello from local:14, i=100
9  Hello from local:2, i=100
10 Hello from local:16, i=100
11 Hello from local:4, i=100
12 Hello from local:7, i=100
13 Hello from local:4, i=100
14 Hello from local:3, i=100
15 Hello from local:2, i=100
```

```
Hello from local:3, i=100
Hello from local:18, i=100
Hello from local:14, i=100
Hello from local:7, i=100
Hello from local:14, i=100
```

3.3. Outras funcionalidades disponíveis pelo Apache Hama

Além das primitivas de comunicação, o Apache Hama, na sua versão atual (v0.7[1]) fornece diversas outras funcionalidades. Iremos comentar brevemente sobre cada uma delas nessa Seção. Os exemplos foram retirados de Apache [3].

É possível ler e escrever diretamente em arquivos no Apache Hama. Antes de mandar executar o método `bsp()` na `main()`, é necessário chamar os métodos `setInputFormat()` e `setOutputFormat()` com o tipo de dado que você vai passar como entrada do programa.

Como citado brevemente na Seção 3, existem mais dois métodos que podem ser implementados além do método `bsp()`, que são o `setup()` e `cleanup()`. Eles são úteis para fazer operações antes e depois, respectivamente, do código executado no método `bsp()`. Um exemplo de uso é mostrado em Código 4.

Código 4: Exemplo de uso dos métodos setup()/cleanup().

```
@Override
public void setup(BSPPeer<NullWritable, NullWritable, Text,
   DoubleWritable, DoubleWritable> peer) throws IOException {
   // Seleciona um nó como mestre
   this.masterTask = peer.getPeerName(peer.getNumPeers() / 2);
   }

@Override
public void cleanup(BSPPeer<NullWritable, NullWritable, Text,
   DoubleWritable, DoubleWritable> peer) throws IOException {
   // Imprime o resultado, somente o nó mestre
   if (peer.getPeerName == this.masterTask) {
      System.out.println(this.result);
   }
}
```

Contadores, ou *counters*, podem ser usados no código para fazer contagens diversas, como por exemplo o número de vezes que um laço foi executado. Um exemplo de uso é mostrado em Código 5.

Código 5: Exemplo de uso de contadores.

```
// Counters só funcionam com enums
enum LoopCounter{
   LOOPS
}
```

[1] http://ftp.unicamp.br/pub/apache/hama/hama-0.7.0/

```
5
6  @Override
7  public void bsp(BSPPeer<NullWritable, NullWritable, Text,
       DoubleWritable, DoubleWritable> peer) throws IOException,
       SyncException, InterruptedException {
8      for (int i = 0; i < iterations; i++) {
9          peer.getCounter(LoopCounter.LOOPS).increment(1L);
10     }
11 }
```

Finalmente, os combinadores, ou *combiners*, permitem fazer diversas operações típicas de linguagens funcionais nos dados, como por exemplo a soma ou média de todas as mensagens.

```
1  public static class SumCombiner extends Combiner {
2
3      @Override
4      public BSPMessageBundle combine(Iterable<BSPMessage>
           messages) {
5          BSPMessageBundle bundle = new BSPMessageBundle();
6          int sum = 0;
7
8          Iterator<BSPMessage> it = messages.iterator();
9          while (it.hasNext()) {
10             sum += ((IntegerMessage) it.next()).getData();
11         }
12
13         bundle.addMessage(new IntegerMessage("Sum", sum));
14         return bundle;
15     }
16
17 }
```

Note que é necessário implementar um método especial, chamado `combine()`, para isso.

4. Aplicações para o Apache Hama

Entre as diferentes aplicações possíveis para o Apache Hama, temos problemas clássicos como multiplicação de matrizes, solução de sistemas lineares [17], aprendizado de máquina [1] e processamento de grandes volumes de dados usando algoritmos de clusterização [10].

Uma aplicação mais prática para o Apache Hama seria análise de rede sociais. Podemos representar a rede social usando “atores”, que podem ser usuários, organizações, eventos ou qualquer tipo de objeto. Cada ator seria um nó, enquanto as relações entre nós seriam representadas por linhas conectadas, ou seja o clássico exemplo de um grafo [14, 19].

Nesse tipo de aplicação, a vantagem do Apache Hama sobre o MapReduce vem do fato em que, no MapReduce, a cada iteração, é necessário transmitir todos os dados

novamente. O problema é que grafos e matrizes podem facilmente chegar a bilhões de entradas [16], e (re-)enviar esses dados a cada iteração (considerando que esses dados muitas vezes são estáticos), é pouco eficiente [14].

No Apache Hama os dados podem ser divididos antes da computação, e os dados ficam lá. Caso seja necessário, somente os dados importantes para a execução do próximo super-passo são trocados entre nós. Ou seja, enquanto no modelo MapReduce cada passo é sem-estado (*stateless*), no Apache Hama cada passo mantém o estado anterior (*stateful*).

5. Conclusão

O Apache Hama é relativamente recente (o primeiro trabalho sobre ele foi publicado em 2010 [17]), o que pode ser notado pela numeração da sua versão mais recente (v0.7). Talvez por isso, não existam muitos trabalhos na literatura falando sobre o Apache Hama de forma específica.

Porém, é possível perceber pelos trabalhos publicados que o Apache Hama tem potencial de para ser a base de aplicações que rodem num conjunto massivo de dados no futuro, em especial naquelas que envolvem uso de grafos ou matrizes. O fato dele ser baseado num modelo clássico da literatura, o BSP, também é uma vantagem significativa.

A literatura mostra que existe essa necessidade de se processar conjuntos de dados imensos organizados em grafos ou matrizes. Resta saber se será o Apache Hama ou outras modificações do modelo MapReduce, como o Apache Giraph [14] (também baseado no modelo BSP) ou Apache Spark [21] (que utiliza um modelo de computação diferente, mas também contém módulos especializados para computação em grafos), que tomará o lugar do tradicional MapReduce para essa tarefa.

Referências

[1] V. S. Agneeswaran, P. Tonpay e J. Tiwary. "Paradigms for realizing machine learning algorithms". Em: *Big Data* 1.4 (2013), pp. 207–214 (ver p. 8).

[2] Apache. *Apache Hama - a general computing engine on top of Hadoop*. 2015. URL: `http://hama.apache.org/` (acesso em 19/06/2015) (ver p. 1).

[3] Apache. *FrontPage - Hama Wiki*. 2015. URL: `http://wiki.apache.org/hama/` (acesso em 19/06/2015) (ver pp. 4, 7).

[4] R. H. Bisseling. *Parallel Scientific Computation: A structured approach using BSP and MPI*. 2004 (ver p. 3).

[5] R. Y. de Camargo, A. Goldchleger, F. Kon e A. Goldman. "Checkpointing BSP parallel applications on the InteGrade Grid middleware". Em: *Concurrency and Computation: Practice and Experience* 18.6 (2006), pp. 567–579. URL: `http://dx.doi.org/10.1002/cpe.966` (ver p. 2).

[6] J. Dean e S. Ghemawat. "MapReduce: simplified data processing on large clusters". Em: *Communications of the ACM* 51.1 (2008), pp. 107–113 (ver p. 1).

[7] B. Elser e A. Montresor. "An evaluation study of bigdata frameworks for graph processing". Em: *Big Data, 2013 IEEE International Conference on*. IEEE. 2013, pp. 60–67 (ver p. 1).

[8] P. Gibbons, Y. Matias e V. Ramachandran. “The queue-read queue-write asynchronous PRAM model”. Em: *Theoretical Computer Science* 196.1–2 (1998), pp. 3–29. URL: http://www.sciencedirect.com/science/article/pii/S030439759700193X (ver p. 2).

[9] A. Goldchleger, A. Goldman, U. Hayashida e F. Kon. “The Implementation of the BSP Parallel Computing Model on the InteGrade Grid Middleware”. Em: *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*. MGC ’05. Grenoble, France: ACM, 2005, pp. 1–6. URL: http://doi.acm.org/10.1145/1101499.1101504 (ver p. 2).

[10] A. A. Golghate e S. W. Shende. “Parallel K-Means Clustering Based on Hadoop and Hama”. Em: *International Journal of Computing and Technology* 1 (2014) (ver p. 8).

[11] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas e R. H. Bisseling. “BSPlib: The BSP programming library”. Em: *Parallel Computing* 24.14 (1998), pp. 1947–1980 (ver p. 3).

[12] Q. Hou, K. Zhou e B. Guo. “BSGP: bulk-synchronous GPU programming”. Em: *ACM Transactions on Graphics (TOG)*. Vol. 27. 3. ACM. 2008, p. 19 (ver p. 3).

[13] B. H. H. Juurlink e H. A. G. Wijshoff. “A Quantitative Comparison of Parallel Computation Models”. Em: *ACM Transactions on Computer Systems* 16.3 (ago. de 1998), pp. 271–318. URL: http://doi.acm.org/10.1145/290409.290412 (ver p. 2).

[14] T. Kajdanowicz, W. Indyk, P. Kazienko e J. Kukul. “Comparison of the efficiency of mapreduce and bulk synchronous parallel approaches to large network processing”. Em: *Data Mining Workshops (ICDMW), 2012 IEEE 12th International Conference on*. IEEE. 2012, pp. 218–225 (ver pp. 8, 9).

[15] J. S. Kirtzic. “A Parallel Algorithm Design Model for the Gpu Architecture”. AAI3547670. Tese de doutorado. Richardson, TX, USA, 2012 (ver p. 2).

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser e G. Czajkowski. “Pregel: a system for large-scale graph processing”. Em: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146 (ver pp. 1, 9).

[17] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim e S. Maeng. “Hama: An efficient matrix computation with the mapreduce framework”. Em: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE. 2010, pp. 721–726 (ver pp. 3, 8, 9).

[18] D. B. Skillicorn e D. Talia. “Models and Languages for Parallel Computation”. Em: *ACM Computing Surveys* 30.2 (jun. de 1998), pp. 123–169. URL: http://doi.acm.org/10.1145/280277.280278 (ver p. 2).

[19] I. Ting, C.-H. Lin, C.-S. Wang et al. “Constructing A Cloud Computing Based Social Networks Data Warehousing and Analyzing System”. Em: *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*. IEEE. 2011, pp. 735–740 (ver p. 8).

[20] L. G. Valiant. “A Bridging Model for Parallel Computation”. Em: *Commun. ACM* 33.8 (ago. de 1990), pp. 103–111. URL: http://doi.acm.org/10.1145/79173.79181 (ver p. 2).

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker e I. Stoica. “Spark: cluster computing with working sets”. Em: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Vol. 10. 2010, p. 10 (ver p. 9).