Capítulo 6

Paralelismo de tarefas utilizando OpenMP 4.5

Calebe P. Bianchini, Fabrício G. Vilabôas, Leandro N. Castro

Abstract

This paper presents the basic concepts of task parallelism using OpenMP 4.5. Besides those concepts, it also describes its usage in natural computing to solve clustering and optimization problems.

Resumo

Este artigo apresenta uma visão geral do paralelismo de tarefas utilizando a especificação OpenMP 4.5. Ao longo do texto, além de apresentar esses conceitos, eles serão aplicados em algoritmos de computação natural para resolver problemas de agrupamento e otimização.

6.1. Introdução

Já é amplamente conhecido que as mudanças das arquiteturas dos processadores tiveram um papel fundamental na evolução das discussões sobre a computação paralela e, consequentemente, sobre o projeto de algoritmos eficientes. Inicialmente, os computadores de propósitos específicos e desenhados exclusivamente para alto desempenho foram marcados por grandes empresas mundiais, como a NEC, Cray, Fujitso, dentre outras tantas que se destacaram ao longo dos anos. Elas ainda figuram entre as maiores do mundo, principalmente quando consultado o site TOP500[1] [Hager and Wellein, 2010].

Porém, com o surgimento de arquiteturas de computadores denominados pessoais (ou PC), a construção de supercomputadores passou a utilizar estas arquiteturas que têm um melhor custo-benefício, além de serem caracterizados como "de prateleira" e, portanto, não necessariamente projetadas para computação científica. Por isso, no cenário atual - entenda-se, final da década de 2010 - todos os principais fabricantes de supercomputadores oferecem suas soluções com base nessas arquiteturas "modernas".

[1] Consulte o site `http://www.top500.org` para maiores detalhes desses fabricantes.

Um dos recursos existentes nos processadores dessas arquiteturas é o paralelismo. Esse paralelismo está presente em qualquer sistema computacional, seja em um supercomputador, em um computador pessoal, ou, até mesmo, no celular ou smartphones. Pode-se enumerar diversos recursos que essas arquiteturas apresentam com algum grau de paralelismo, como instruções vetoriais, multi-threads, multi-cores, multi-sockets, coprocessadores, dentre outros recursos existentes no hardware [McCool et al., 2012].

A partir do momento em que a computação paralela é algo "pervasivo", entende-se que é fundamental sua compreensão para que bons projetos de algoritmos sejam desenhados e, consequentemente, possam ser programados. Nesse sentido, é natural que ao longo de todos esses anos de computação paralela diversas soluções tenham sido apresentadas para diversos problemas, tanto na arquitetura do processador, quanto no projeto de algoritmos, como também no modelo de programação [Rauber and Rünger, 2013].

A título de curiosidade, umas das soluções propostas para a arquitetura é amplamente conhecida: o *pipeline*. A Figura 6.1 mostra um exemplo de execução de 4 instruções, na qual cada instrução é dividida em 4 estágios: *busca*, *decodificação*, *execução* e *escrita*. Nesse exemplo, cada estágio das instruções é executado como em uma linha de produção, de tal forma que, em cada unidade de tempo, um estágio diferente de uma instrução diferente é executada: perceba que o tempo *t4*, cada instrução está em um estágio diferente do *pipeline*.

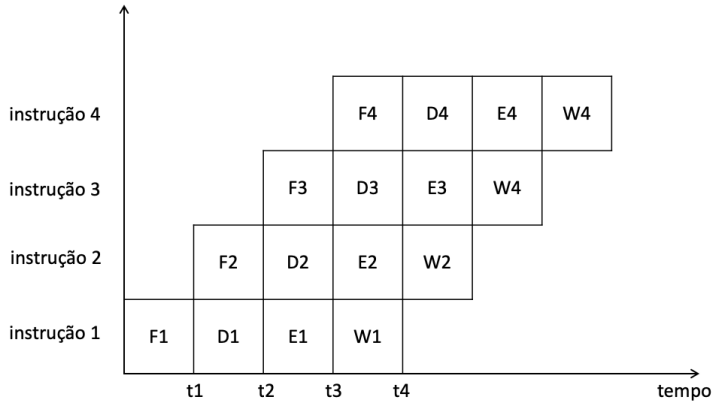
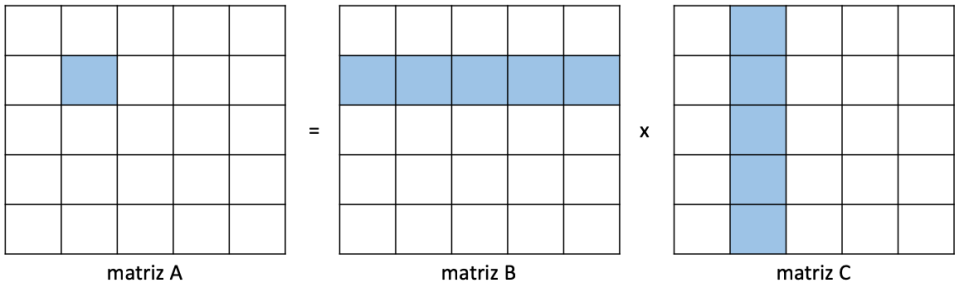


Figure 6.1: Sobreposição das execuções dos estágio de diferentes instruções: *busca* (F), *decodificação* (D), *execução* (E) e *escrita* (W).

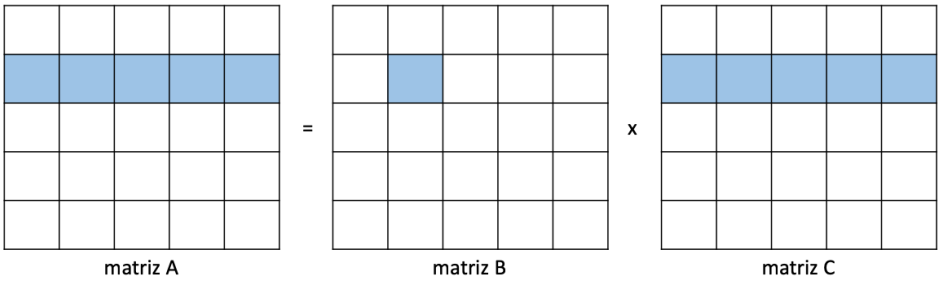
Apesar da perceptível melhoria no desempenho, outros desafios precisam ser solucionados, como o tempo de ciclo do processador para em cada estágio, a dependência dos registradores e dados utilizados nas instruções subsequentes, a execução inválida de estrutura de controle condicionais, dentre outros conflitos (*hazards*) de *pipeline*.

No contexto de projetos de algoritmos, é possível perceber que algoritmos clássicos podem ser reprojetados para aproveitar os recursos da computação paralela. Por exemplo, o algoritmo de multiplicação de matrizes convencional, observado na Figura 6.2, pode ser reprojetado para uma arquitetura paralela [Kumar, 2002].

A solução tradicional, mostrada na Figura 6.2a, dificulta o uso eficiente de instruções vetoriais existentes no processador, já que o percurso da matriz C é feito pela



(a) Projeto do algoritmo convencional: for_{ijk}.



(b) Projeto de algoritmo modificado para aproveitamento de arquitetura paralela: for_{ikj}.

Figure 6.2: Diferentes projetos para o algoritmos de multiplicação de matrizes.

coluna. Já o projeto apresentado na Figura 6.2b potencializa o uso dessas instruções vetoriais e paralelas, uma vez que, tanto a matriz A quanto a matriz C, são acessadas por meio das linhas [David, 2016]. Essa técnica ainda pode ser melhor aproveitada se um estudo mais aprofundado da hierarquia de memória for realizado [Kumar, 2002].

Em um nível mais abstrato, é possível apresentar um modelo de programação que descreve um sistema computacional paralelo por meio de um ambiente de programação. Esse modelo de programação paralelo é influenciado pela arquitetura do processador, pela linguagem de programação, pelo compilador, por bibliotecas, pelo *runtime*, dentre outros critérios que podem ser utilizados para diferenciar os modelos de programação paralela, como [Rauber and Rünger, 2013]:

- níveis de paralelismo (instrução, comando, *loop*, tarefas, etc);
- paralelismo explícito, implícito, ou personalizado;
- granularidade e modo de execução (síncrono ou assíncrono, SPMD, etc);
- modelos de comunicação e unidade para a troca da informação;
- controle de seção crítica e mecanismos de sincronização.

Este capítulo apresenta um modelo de programação paralelo definido pela especificação OpenMP chamado de tarefa (*task*), além de uma breve discussão desse modelo aplicado a área de computação natural, em algoritmos de classificação e bioinspirados.

6.2. Processamento de Alto Desempenho

Existem diversos assuntos a serem tratados quando o tema é processamento de alto desempenho. Naturalmente, seu início se dá pela discussão sobre o modelo clássico da arquitetura de *von Neumann*. Uma das características desse modelo é a presença de uma unidade de memória, responsável por armazenar um programa que, ao mesmo tempo, se torna o grande gargalo de acesso pela UCP (Unidade Central de Processamento) [Rauber and Rünger, 2013].

As soluções para este problema permitiram modificações no modelo clássico de *von Neumann*, derivando arquitetura especializadas que, consequentemente, impactaram o processamento de alto desempenho.

Dentre essas especializações estão o surgimento do *cache*, sendo este responsável por permitir um acesso mais rápido às instruções e aos dados de um programa pela UCP. O princípio fundamental que rege o funcionamento da *cache* é a localidade, ou seja, os itens que foram armazenados por ela possuem uma alta probabilidade de serem acessados em um futuro próximo [Hager and Wellein, 2010].

Uma outra especialização de arquitetura está no uso de paralelismo no nível de instruções (do inglês, ILP - *instruction-level parallelism*) por uma UCP. Assim, é possível aumentar a quantidade de instruções em execução já que elas são executadas com algum grau de paralelismo pela UCP - impactando, novamente, o processamento de alto desempenho.

A partir dessas diversas especializações da arquitetura foi possível estudar melhor o comportamento do hardware em relação ao uso de instruções e dos dados. A classificação mais conhecida é denominada de *taxonomia de Flynn*. Ela contempla a arquitetura de *von Neumann* por meio da categoria SISD (*single instruction, single data*). Também pode ser observado o paralelismo de dados, presente nas unidades de processamento vetoriais e nas unidades de processamento gráfico, categorizados como SIMD (*single instruction, multiple data*). Uma outra categoria, MIMD (*multiple instruction, multiple data*), identifica que uma arquitetura pode executar, de forma paralela, fluxos de instruções independentes, sendo que cada fluxo tem seu próprio fluxo de dados. Essa categoria permite que um conjunto de processadores sejam organizados em múltiplos computadores, tendo, entre si, uma memória distribuída; ou em um único computador, possuindo uma memória compartilhada com interconexões especializadas. [Pacheco, 2011]

Apesar de existirem diversos outros assuntos importantes para um melhor e maior entendimento de processamento de alto desempenho, conforme pode ser encontrado na literatura clássica como [Hager and Wellein, 2010], [McCool et al., 2012], [Rauber and Rünger, 2013] e [Pacheco, 2011], o foco principal deste seção é apresentar o modelo MIMD com o uso de memória compartilhada, juntamente com um modelo de programação paralela denominada OpenMP [van der Pas et al., 2017, Board, 2015].

6.2.1. *OpenMP*

O OpenMP é definido como uma API (*Application Programming Interface*) para programação paralela de memória compartilhada. Ele foi projetado para que toda thread existente na solução tenha possibilidade de acessar toda a memória disponível. No contexto

desse texto, a UCP (Unidade Central de Processamento) será utilizada como a unidade responsável pela execução de uma thread, apesar da UCP poder ser encontrada, hoje, inserida no contexto de *multi-thread*, *multi-core*, *multi-socket*. Em todo caso, a Figura 6.3 representa o sistema de threads em diversas UCP com acesso a memória compartilhada que se assume ao logo das próximas seções.

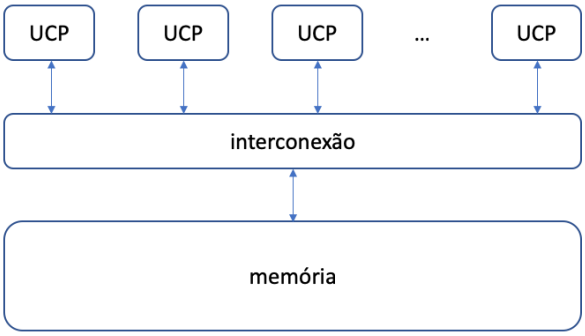


Figure 6.3: Sistema de memória compartilhada, com diversas UCPs.

A programação OpenMP é feita em C, C++ ou Fortan. Sua estrutura completa envolve a definição de diretivas de compilação que define e caracteriza o paralelismo, além das bibliotecas disponíveis em tempo de compilação e execução, bem como variáveis de ambientes para os ajustes no nível de sistema operacional. Ao longo das próximas seções, a sintaxe utilizada será compatível com C e C++.

A diretiva em C/C++ para o uso do OpenMP é:

```
#pragma omp
```

Como ela é uma diretiva, se o compilador não a reconhecer, simplesmente ignora seu significado.

Um região paralela pode ser definida com a diretiva:

```
#pragma omp parallel
```

Essa diretiva envolve um bloco estruturado de comandos em C/C++ que é executado por diversas threads. Durante a execução, até um momento antes da execução da região paralela, existe apenas uma única thread denominada mestre.

Nesse momento, fica claro que o modelo de execução do OpenMP para as regiões paralelas é o *fork/join*. Quando a thread mestre encontrar uma região paralela, diversas threads serão criadas (*fork*) para a execução da região paralela. Cada thread, ao terminar sua execução e chegando ao final da região paralela, espera pelas demais threads até que todas elas também tenham alcançado esse mesmo ponto (*join*). A Figura 6.4 apresenta esse modelo *fork/join* para o OpenMP.

Um exemplo de programa que faz uso de uma região paralela em OpenMP pode ser visto na Listagem 6.1. Neste exemplo, a diretiva da linha 8 define o bloco estruturado compreendido entre as linhas $9-18$ como uma região paralela que é executado por todas as threads disponíveis no ambiente (*fork*). Isso significa que, na linha 11, todas as threads farão a impressão da mensagem "`Ola mundo, sou a thread` ", seguida

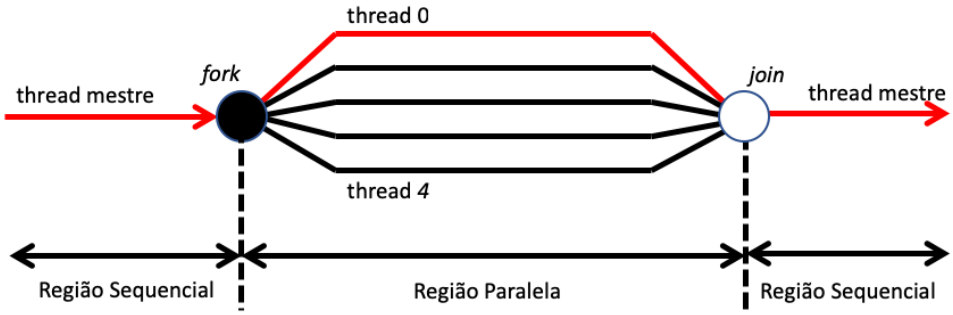


Figure 6.4: Modelo *fork/join* de execução de threads no OpenMP.

```
1  #include <omp.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main (int argc, char *argv[])
6  {
7  /* Cria diversas threads (fork) */
8  #pragma omp parallel
9  {
10   /* Recupera o numero da thread em execucao */
11   printf("Ola mundo, sou a thread %d\n", omp_get_thread_num());
12
13   /* Apenas a thread mestre */
14   if (omp_get_thread_num() == 0)
15   {
16     printf("Numero de threads em paralelo: %d\n", omp_get_num_threads()
       );
17   }
18 }  /* Espera a execucao de todas as threads (join) */
19   return 0;
20 }
```

Listagem 6.1: Exemplo com diretiva de região paralela em OpenMP.

do identificador da thread. Porém, a linha 16 será executada apenas pela thread mestre, uma vez que somente ela satisfará a condição da linha 14. Ao final da execução paralela há a sincronização entre as threads, na linha 18, quando elas são desativadas e volta existir apenas a thread mestre.

Em relação a memória compartilhada, OpenMP assume que qualquer variável declarada fora das regiões paralelas são automaticamente compartilhadas entre as threads. Isso significa que, caso seja necessário, deverá ser indicado quais variáveis terão visibilidade (ou escopo) local a cada thread nos blocos das regiões paralelas. Além disso, caso qualquer variável compartilhada seja modificada por uma thread, a propagação do novo valor para as demais threads segue um modelo de consistência de memória relaxada.

OpenMP também possui construções e diretivas para a divisão de trabalho entre threads. A diretiva mais conhecida para isso é:

`#pragma omp for.`

Essa construção permite que as iterações de um laço de repetição sejam divididos entre as threads disponíveis e, consequentemente, o trabalho sejam dividido entre todas elas.

Uma outra construção do OpenMP permite que haja threads para diferentes blocos estruturados. Essa construção, concebida inicialmente para uma granularidade maior de paralelismo, permite que diferentes funções sejam chamadas pelas threads, aumentando o encadeamento das execuções entre essas threads. A diretiva utilizada para isso é:

`#pragma omp sections.`

Para mais informações sobre essas e outras construções do OpenMP, consulte principalmente as bibliografias utilizadas neste texto, como [Pacheco, 2011], [van der Pas et al., 2017] ou [Board, 2015]; ou visite o repositório de código-fonte disponível[2].

6.2.1.1. *Parallel Task*

O paralelismo de tarefa no OpenMP foi apresentado na versão 3.0 dessa especificação em 2008 (a versão 5.0 foi publicada em 2018). Sua principal característica é permitir que algoritmos irregulares em relação a carga de processamento e o fluxo de sua execução fossem paralelizados. Até então, o paralelismo só poderia feito por meio de laços de repetições ou de seções paralelas, não atendendo todas as características de paralelismo e trazendo um grau de complexidade maior para a programação.

O paralelismo de tarefas provê uma solução elegante para o problema. Ele é gerenciado por um sistema de fila dinâmica que atribui às threads um conjunto de trabalho que pode ser realizado. Essa atribuição é feita quando uma thread solicita um "novo trabalho" para a fila, até que a fila esteja vazia.

As primeiras versões do paralelismo de tarefas era bem rudimentar. Porém, a partir da versão 4.0 do OpenMP, esse paralelismo se tornou bem mais estruturado, com mais diretivas e parâmetros, bem como mais fácil de programar.

Basicamente, uma tarefa é um bloco estruturado com uma diretiva de OpenMP e que pode ser executado por qualquer thread disponível. A sintaxe que define o bloco de uma tarefa é:

`#pragma omp task.`

A Listagem 6.3 mostra um exemplo simples de um programa que faz uso de paralelismo de tarefas. Na região paralela definida na linha 8, apenas uma thread executa o bloco indicado pela diretiva `single` da linha 11, e todas as demais threads ignora esse bloco e passam para a instrução seguinte. Essa linha seguinte é exatamente um ponto de sincronização, no qual essas outras threads esperam o término da região paralela (linha 24). Já a thread que executa a diretiva `single` cria duas tarefas (linha 14 e linha 19). A critério do ambiente de execução, a própria thread que criou a tarefa pode executar uma

[2]Visite o repositório com os códigos utilizados neste minicurso em `https://github.com/hpc-fci-mackenzie/erad-rs-2019`.

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
// Regiao paralela
#pragma omp parallel
{
  /* Apenas uma thread executa o bloco a seguir */
#pragma omp single
  {
    /* Primeira tarefa */
#pragma omp task
    {
      printf("Ola ");
    }
    /* Segunda tarefa */
#pragma omp task
    {
      printf("Mundo ");
    }
  }
} /* Fim da regiao paralela */
  return 0;
}
```

Listagem 6.2: Exemplo de criação de tarefas em OpenMP.

dessas tarefas. Independente disso, uma delas é adicionada à fila dinâmica de atribuição de tarefas. Nesse caso, sabendo que existem diversas outras threads no estado de espera, o ambiente de execução pode atribuir uma tarefa a uma destas threads disponíveis. Neste caso, não existe garantia da ordem de execução das tarefas e, consequentemente, as mensagens "`Ola Mundo` " e "`Mundo Ola` " podem ser impressas.

Um outro exemplo interessante de entender o funcionamento do paralelismo de tarefa é por meio da versão paralela do algoritmo de ordenação Quicksort, que é descrito a seguir [Kumar, 2002, van der Pas et al., 2017]:

1. escolha um elemento do arranjo de dados, chamando-o de *pivô*.
2. em seguida, coloque o *pivô* em sua posição final e ordenado em relação ao arranjo, com os menores elementos à esquerda dele, bem como os maiores elementos à direita. Essa fase é chamada de *particionamento*.
3. *recursivamente*, faça os passos 1e 2 para as partes esquerda e direita do *pivô* do arranjo.
4. a *recursão* termina quando não houver nenhum outro elemento a ser ordenado.

Sabe-se que a escolha do *pivô* é fundamental para o comportamento estável do Quicksort e, uma vez que a probabilidade de haver um desbalanceamento nessa divisão é

alta, cada divisão pode ser definida como uma tarefa e fazer um ótimo uso do paralelismo de tarefas.

Uma forma simplificada de visualizar a recursão existente no Quicksort é por meio de uma árvore. Essa árvore pode ser vista na Figura 6.5: cada nível da árvore representa uma nova subdivisão da tarefa de recursão, onde estão circulados os *pivôs* em cor vermelha e, uma vez ordenado, ele fica anotado em cor azul.

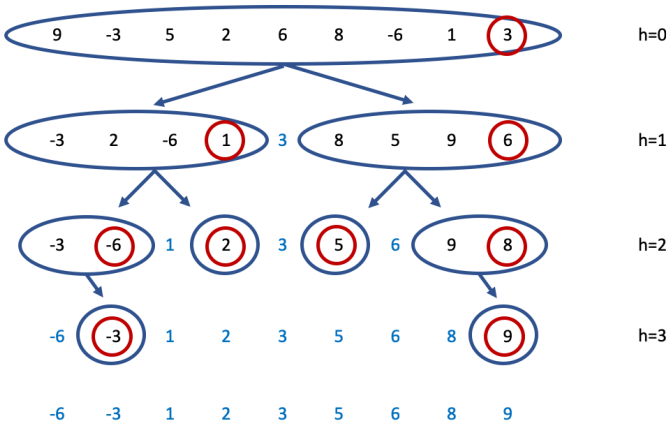


Figure 6.5: Árvore de recursão do Quicksort.

A paralelização em tarefas desse algoritmo leva em consideração que a altura máxima da árvore h é derivada da quantidade de elementos de um arranjo: $h = \log_2 n$. Na Figura 6.5, a altura é $h = 3$. Sabe-se também que, em cada nível da árvore h', existe $2^{h'}$ sub-arranjos devido às recursões. Assim, também é possível considerar que, em um determinado computador com p processadores, é possível criar recursões concorrentes com, no máximo $p = h'_p$. Dessa forma, o algoritmo Quicksort com paralelismo de tarefas em OpenMP pode ser visto na Listagem 6.3.

Nessa Listagem 6.3, as linhas 5 e 10 definem as tarefas `#pragma omp task` com algumas cláusulas modificadoras:

- *final*: evita que novas tarefas sejam criadas conforme avaliação da expressão pelo ambiente de execução do OpenMP.
- *mergeable*: modifica o ambiente de dados mantido automaticamente pelo OpenMP para que o escopo dos dados sejam compartilhados e, consequentemente, diminua o uso de memória.
- *default*: modifica o comportamento padrão de compartilhamento de variáveis, sendo obrigatória a indicação do modo de compartilhamento das variáveis utilizadas na região paralela de tarefas.
- *shared*: as variáveis indicadas têm seu escopo definido como compartilhado entre todas as tarefas/threads.
- *firstprivate*: as variáveis indicadas têm seu escopo definido como local (privadas) e são iniciadas com o mesmo valor respectivo às variáveis associadas.

```
long quicksort(long *a, long lo, long hi)
{
    if(lo<hi) {
        long p = partition(a, lo, hi);
#pragma omp task final( (p - lo) < n_proc) mergeable \
    default(none) shared(a) firstprivate(lo,p)
    {
        quicksort(a, lo, p - 1); // Ramo esquerdo da arvore
    }
#pragma omp task final( (hi - p) < n_proc) mergeable \
    default(none) shared(a) firstprivate(hi,p)
    {
        quicksort(a, p + 1, hi); // Ramo direito da arvore
    }
        return(p);
    }
}
```

Listagem 6.3: Quicksort com paralelismo de tarefas.

6.3. Computação Natural

Os mecanismos naturais ou biológicos para o processamento de dados já é conhecido é tem sido capturado e incorporados em diversos sistemas computacionais. Esse é o princípio da *Computação Natural*, que tenta aproximar da natureza as abordagens computacionais empregas na construções de diversos sistemas. Para isso, tem-se um esforço na construção de ferramentas para soluções em sistemas complexos nas diversas áreas; na projeção de dispositivos computacionais que descrevem fenômenos naturais; na sintetização de vida artificial; e na suplementação da arquitetura clássica de *Von Neumann* e suas derivações [Castro et al., 2004].

Uma das primeiras áreas de estudo da computação natural é também uma das mais antigas. Ela aproveita diversas descobertas de princípios e teorias da natureza, bem como utiliza diversos modelos derivados dessas descobertas. Nesta área, destacam-se [Castro et al., 2004, De Castro, 2006]: *redes neurais*, *algoritmos evolutivos*, *sistemas imunológicos* e *inteligência de enxames*; sendo que este último assunto será aprofundado nas seções a seguir.

6.3.1. Inteligência de enxames

O termo inteligência de enxames foi proposto no fim da década de 1980 onde se referia a sistemas robóticos compostos por uma coleção de agentes simples em um ambiente interagindo de acordo com regras locais. O termo “enxame” (ou coletivo) é utilizado de forma genérica para se referir a qualquer coleção estruturada de agentes capazes de interagir. O exemplo clássico de um enxame é um enxame de abelhas, entretanto a metáfora de um enxame pode ser estendida a outros sistemas com uma arquitetura similar. Uma colônia de formigas pode ser vista como um enxame onde os agentes são formigas, uma revoada de pássaros é um enxame onde os agentes são pássaros, um engarrafamento é um enxame onde os agentes são carros, uma multidão é um enxame de pessoas, um sistema imunológico é um enxame de células e moléculas, e uma economia é um enxame

de agentes econômicos. Embora a noção de enxame sugira um aspecto de movimento coletivo no espaço, como em um "enxame de pássaros", estamos interessados em todos os tipos de comportamentos coletivos, não apenas movimento espacial.

A inteligência de enxame inclui qualquer tentativa de projetar algoritmos ou dispositivos distribuídos de solução de problemas inspirados pelo comportamento coletivo de insetos sociais e outras sociedades animais [Bonabeau et al., 1999]. A inteligência de enxame é uma propriedade de sistemas compostos por agentes não inteligentes e com capacidade individual limitada, capazes de apresentar comportamentos coletivos inteligentes [White and Pagurek, 1999].

Algumas propriedades da inteligência coletiva:

- Proximidade: os agentes devem ser capazes de interagir;
- Qualidade: os agentes devem ser capazes de avaliar seus comportamentos;
- Diversidade: permite ao sistema reagir a situações inesperadas;
- Estabilidade: nem todas as variações ambientais devem afetar o comportamento de um agente;
- Adaptabilidade: capacidade de se adequar a variações ambientais.

Sendo assim, um sistema de enxame é aquele composto por um conjunto de agentes capazes de interagir entre si e com o meio ambiente. A inteligência de enxame é uma propriedade emergente de um sistema coletivo que resulta de seus princípios de proximidade, qualidade, diversidade, estabilidade e adaptabilidade. Duas principais linhas de pesquisa podem ser observadas em inteligência de enxame:

- Trabalhos inspirados por comportamentos sociais de insetos;
- Trabalhos inspirados na habilidade das sociedades humanas em processar conhecimento.

Embora existam diferenças entre estas abordagens, elas possuem a seguinte característica importante em comum: população de indivíduos capazes de interagir entre si e com o ambiente.

Insetos sociais são aqueles que vivem em comunidades ou colônias como por exemplos: formigas, abelhas, vespas e cupins. Uma colônia pode ser considerada como uma grande família de insetos sem hierarquia, na maioria dos casos. Dentro de uma colônia existe uma sobreposição entre gerações de pais e filhos. Cada inseto parece ter sua própria agenda, mesmo assim, uma colônia parece extremamente bem organizada. A integração de todas as atividades individuais não requer supervisão, trata-se de um fenômeno auto-organizado. Formigas do tipo *leafcutter* cortam folhas de plantas e árvores para cultivar fungos. Formigas trabalhadoras buscam por alimento a grandes distâncias do ninho, criando caminhos a partir do ninho e de volta para o ninho. Formigas do tipo *weaver* formam correntes com seus próprios corpos permitindo que elas atravessem grandes buracos e carreguem alimento para o ninho. Durante sua fase de movimentação e busca

por alimento, as formigas do tipo *army* organizam frentes de batalha impressionantes. As abelhas constroem uma série de pentes paralelos formando correntes que induzem um aumento local de temperatura. Desta forma, fica mais fácil moldar a colmeia. As fontes de alimento são exploradas de acordo com sua qualidade e distância do ninho. Podemos citar como exemplos de problemas resolvidos por insetos sociais: encontrar alimento, construir ou aumentar o ninho, dividir a mão de obra, alimentar a colônia, responder a desafios externos (clima, predadores, etc.), soar alarmes e encontrar um local apropriado para construir o ninho.

6.3.1.1. Colônia de formigas

As formigas são os insetos sociais mais amplamente estudados. Exemplos da popularidade das formigas podem ser encontrados em filmes como Formiguinha Z e Vida de Inseto. Considere o seguinte trecho de Formiguinha Z, onde uma formiga trabalhadora chamada Z entra no consultório do terapeuta reclamando de sua insignificância:

- "Eu me sinto insignificante.";
- "Ah, você teve um grande progresso.";
- "Tive?";
- "Sim . . . você é insignificante!".

Entretanto, a perspectiva que a maioria das pessoas tem da organização social dos insetos é errônea. Os filmes acima mostram isso claramente. Neste filme, por exemplo, existe uma forte hierarquia social, com, por exemplo, herdeiros de trono.

Algumas tarefas que as formigas devem desempenhar: coletar e distribuir alimento, construir o ninho, cuidar do ninho, dos ovos e das larvas, etc. Alocação de tarefas é o processo que resulta em alguns trabalhadores realizando tarefas específicas, em quantidades apropriadas à situação atual. Trata-se de soluções encontradas para problemas dinâmicos e requer, portanto, um processo contínuo de adaptação. No caso particular das formigas, este processo é auto-organizado. Nenhuma formiga é capaz de avaliar as necessidades globais do formigueiro e nem de contar a quantidade de trabalhadores envolvidos em cada tarefa de forma a decidir como realocá-los. A capacidade de cada formiga é limitada. Cada trabalhador precisa tomar apenas decisões locais.

6.4. Aplicações - Otimização e Agrupamento

Para a seção de *hands-on*, apresentamos dois casos de uso. O primeiro caso de uso é a resolução de um problema de otimização e o segundo caso de uso é a resolução de um problema de agrupamento de dados. Ambos os casos são baseados em inteligência de enxames mais especificamente em colônias de formigas.

Como visto na Seção 6.3, ao desenvolver algoritmos bio-inspirados, estamos buscando por comportamentos que emergem de ações individuais e que alteram o comportamento de todo o ambiente. As formigas são um exemplo bem didático para a explicação

deste conceito. Leve em consideração a tarefa de busca por alimento. Cada formiga segue um caminho inicialmente aleatório e, ao achar o alimento, volta deixando uma trilha de feromônio para marcar o caminho. Ao fazer isto, a probabilidade de outras formigas irem por esse mesmo caminho aumenta. A medida que novas formigas chegam ao alimento, elas voltam deixando essa trilha de feromônios. Desta forma temos um comportamento coletivo que emerge a partir de uma ação individual.

Essa seção está estruturada da seguinte forma: a Seção 6.4.1 apresenta uma breve revisão sobre grafos, a Seção 6.4.2 apresenta o problema de otimização e a Seção 6.4.3 apresenta o problema de agrupamento de dados.

6.4.1. Breve recordação sobre grafos

Um grafo pode ser definido como uma 2-upla $G = (V, E)$ onde V é um conjunto de vértices ou nós, e E é um conjunto de conexões ou pares de nós ligando estes vértices: $V = v_0, v_1, \ldots, v_N$, $E = (v_i, v_j) : i \neq j$.

Um caminho em um grafo consiste em uma sequência alternada de nós e conexões. Quando não existir ambiguidade, um caminho pode ser descrito por uma sequência de nós.

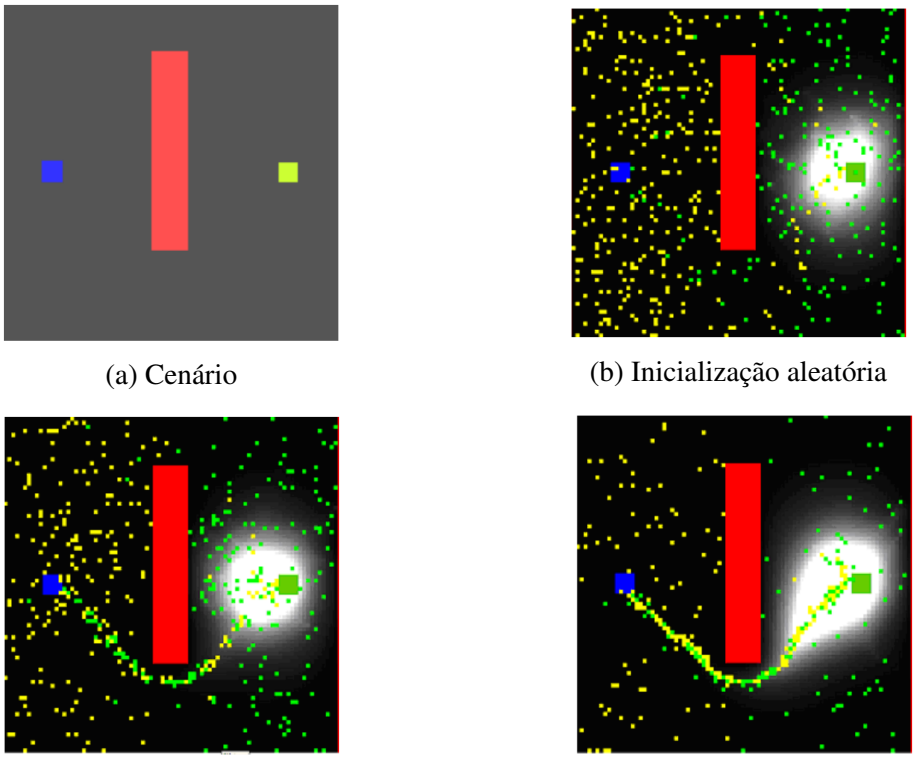
Um grafo é dito 1) conexo se existir pelo menos uma conexão ligando cada par de nós; 2) Direcionado quando existe uma direção específica de percurso; 3) Ponderado se para cada conexão $e \in G$ for especificado um número não negativo $w(e) \geq 0$ denominado peso ou comprimento de e.

6.4.2. *ACO - Ant Colony Optimization*

Alguns estudos de campo mostram que as formigas possuem uma grande capacidade de explorar fontes de alimentos sem perderem a capacidade de explorar o ambiente como um todo. Desta forma, as formigas conseguem encontrar o menor caminho entre o ninho e as fontes de alimento. O *Ant Colony Optimization*, ou ACO, considera a tarefa de busca por alimento como a metáfora para seu desenvolvimento.

Quando alguma fonte de alimento é encontrada, as formigas são recrutadas para buscarem aquele alimento e levarem para o ninho. Recrutamento é o nome dado ao mecanismo comportamental que permite que uma colônia de formigas reúna rapidamente uma grande quantidade de coletadoras em torno de uma determinada fonte de alimento. O mecanismo mais utilizado para o recrutamento é a tilha de feromônios. O feromônio possui duas funções importantes: definir a trilha a ser seguida e servir como sinal de orientação para as formigas passeando fora do ninho.

As formigas, ao definirem o menor caminho entre a fonte de alimento e o ninho, estão minimizando o tempo gasto nesta viagem. Generalizando essa visão, podemos aplicar essa metáfora à resolução do problema de roteamento similar ao problema do caixeiro viajante (TSP). Isto foi proposto por [Dorigo et al., 1996]. A abordagem proposta pelos autores está baseada em um grupo de "formigas artificiais" que liberam e seguem "trilhas de feromônio artificial". Neste caso, existe uma colônia de formigas artificiais, cada uma indo de uma cidade a outra de forma independente, favorecendo cidades próximas ou caminhando aleatoriamente. Enquanto uma formiga atravessa um determinado



(a) Cenário

(b) Inicialização aleatória

(c) Após iterações

(d) Final da execução

Figure 6.6: Experimento de busca por alimento

caminho entre cidades, ela libera certa quantidade de feromônio inversamente proporcional ao comprimento total do caminho percorrido pela formiga. Com isso temos que quanto menor o comprimento do caminho percorrido, maior a quantidade de feromônio liberada e vice-versa. Depois que todas as formigas tiverem completado suas rotas e liberado feromônio, as conexões pertencentes a maior quantidade de rotas mais curtas terão mais feromônio depositado. Como o feromônio evapora com o tempo, quanto maior o comprimento do caminho, mais rápido será o desaparecimento de uma trilha em um caminho longo. A maior parte dos algoritmos de otimização baseados em colônias de formigas é utilizada para resolver problemas de otimização combinatória representados por grafos.

6.4.2.1. Uma Simulação de Vida Artificial

As Figuras 6.6a, 6.6b, 6.6c e 6.6d ilustram o experimento de busca por alimento. Ao lado direito das figuras está o ninho, ao lado esquerdo está a fonte de alimento e no meio está um obstáculo. Este experimento foi feito com 500 formigas. É possível observar que a trilha de feromônios fica mais forte com o passar das iterações.

6.4.2.2. Algoritmo Genérico de Otimização por Colônias de Formigas

Um algoritmo de otimização por colônias de formigas alterna, por uma quantidade máxima de iterações, a aplicação de dois procedimentos básicos:

- Um procedimento paralelo de construção/modificação de trilhas no qual um conjunto de N formigas constrói/modifica N soluções paralelas para o problema;
- Uma regra de atualização de feromônio a partir da qual a quantidade de feromônio nas conexões é alterada.

O processo de construir ou modificar uma solução (caminho) é feito de forma probabilística, e a probabilidade de uma nova conexão ser adicionada à solução sendo construída é função de uma qualidade heurística η (*heuristic desirability*) e da quantidade de feromônio τ depositada por outras formigas. A qualidade heurística expressa a probabilidade de uma formiga se mover para uma determinada conexão. Por exemplo:

- Quando o caminho mínimo está sendo procurado, η pode ser tomado como sendo inversamente proporcional à distância entre um par de nós;
- A regra de atualização da quantidade de feromônio deve levar em conta a taxa de evaporação de feromônio τ e a qualidade das soluções produzidas.

6.4.2.3. Exemplo de Aplicação

Considere o problema do caixeiro viajante (TSP) representado sob a forma de um grafo. Neste problema, as formigas constroem as soluções movendo-se de um nó para outro do grafo. A cada iteração, uma formiga k, $k = 1, ...N$, constrói um caminho (rota) aplicando uma regra de transição probabilística $(\varepsilon - 1)$ vezes. A transição de uma formiga da cidade i para a cidade j na iteração t dependerá de três fatores: 1) do fato da cidade já ter sido visitada ou não; 2) do inverso da distância d_{ij} entre as cidades i e j, denominado de visibilidade $\eta_{ij} = 1/d_{ij}$; e 3) da quantidade de feromônio τ_{ij} na conexão ligando as cidades i e j.

Como no caso do TSP cada cidade não deve ser visitada mais do que uma vez, é preciso armazenar informação sobre as cidades que já foram visitadas. Isso pode ser feito empregando-se, por exemplo, uma lista tabu ou memória, que definirá o conjunto de cidades J_i^k que a formiga k ainda deve visitar enquanto na cidade i. A probabilidade de uma formiga k ir de uma cidade i para uma cidade j na iteração t é dada pela regra de transição apresentada pela Equação 1, onde $\tau_{ij}(t)$ é o nível de feromônio na conexão (i, j), e η_{ij} é a visibilidade da cidade j quando na cidade i. Os parâmetros α e β são definidos pelo usuário e controlam o peso relativo da intensidade da trilha (feromônio) e da visibilidade. Por exemplo, se $\alpha = 0$ cidades mais próximas tenderão a serem escolhidas, enquanto se $\beta = 0$, apenas amplificação na quantidade de feromônio será considerada.

$$p_{ij}^k(t) = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in J} [\tau_{il}(t)]^\alpha [\eta_{il}]^\beta}, & \text{se} j \in J_i^k \\ 0, & \text{caso contrário} \end{cases} \tag{1}$$

Quando uma formiga atravessa uma conexão ela libera um pouco de feromônio. Neste caso, a quantidade de feromônio liberada em cada conexão (i, j) pela formiga k, $\Delta\tau_{ij}^k(t)$, depende de seu desempenho que é dado pela Equação 2, onde $L^k(t)$ é o comprimento da rota $T^k(t)$ percorrida pela formiga k na iteração t, e Q é outro parâmetro definido pelo usuário.

$$\Delta\tau_{ij}^k(t) = \begin{cases} Q/L^k(t), & \text{se}(i,j) \in T^k(t) \\ 0, & \text{caso contrário} \end{cases} \tag{2}$$

A regra de atualização de feromônio é dada pela Equação 3, onde $\rho \in [0,1)$ é a taxa de decaimento de feromônio, $\Delta\tau_{ij}(t) = \sum_k \Delta\tau_{ij}^k(t)$, e $k = 1, \ldots N$é o índice das formigas..

$$\tau_{ij}(t) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}(t) \tag{3}$$

Os proponentes do algoritmo sugerem a utilização de $N = \varepsilon$, ou seja, a quantidade de formigas igual a quantidade de cidades do grafo. Os autores também introduziram o conceito de "formigas elitistas", responsáveis por reforçar a melhor rota encontrada até o momento, adicionando $b.Q/L_{best}$ ao seu valor de feromônio, onde b é a quantidade de formigas elitistas, e L_{best} é o comprimento da melhor rota encontrada até o momento. Alguns parâmetros sugeridos: $\alpha = 1$, $\beta = 5$, $\rho = 0.5$, $N = \varepsilon$, $Q = 100$, $\tau_0 = 10^{-6}$, e $b = 5$.

6.4.3. *ACA - Ant Clustering Algorithm*

Para limpar seus formigueiros, algumas espécies de formigas juntam corpos e partes de corpos de formigas mortas em regiões específicas do formigueiro. O mecanismo básico por trás deste processo é uma atração entre os itens mortos mediada pelas formigas. Pequenos amontoados se formam e vão crescendo atraindo uma maior quantidade de corpos naquela região do espaço. Este comportamento pode ser modelado utilizando-se duas regras simples:

- Regra para pegar um item: se uma formiga encontra um item morto ela o pega e passeia pelo ambiente até encontrar outro item morto. A probabilidade desta formiga pegar o item morto é inversamente proporcional a quantidade de itens mortos naquela região do espaço;
- Regra para largar um item: carregando um item a formiga eventualmente encontra mais itens no caminho. A probabilidade desta formiga deixar este item junto ao outro é proporcional à quantidade de itens mortos naquela região.

Como resultado destas simples regras comportamentais, todos os itens mortos irão, eventualmente, ser agrupados na mesma região ou em regiões vizinhas do espaço.

A análise de *cluster* ou *clusterização* de dados pode ser definida como a organização ou separação de um conjunto de dados ou padrões em grupos denominados de

clusters. Essa organização é feita baseada em algum critério de similaridade. Os dados são geralmente representados por um vetor de medidas ou atributos que corresponde a um ponto em um espaço multidimensional. Intuitivamente, dados em um mesmo *cluster* são mais semelhantes do que dados que não pertencem ao mesmo *cluster*. O problema de clusterização de dados pode ser definido como a seguir: seja um conjunto X de N amostras (dados), $X = x1, \ldots, x_N$, cada qual de dimensão L, encontre um esquema de discriminação para agrupar (*clusterizar*) os dados em c grupos denominados de *clusters*. O número de *clusters* e as características de cada *cluster* devem ser determinados.

Para desenvolver um esquema de discriminação de forma a *clusterizar* os dados é necessário definir uma métrica, geralmente uma medida de distância, que quantifica o grau de similaridade (ou dissimilaridade) entre dois dados em um determinado espaço métrico. A métrica mais comumente utilizada é a distância Euclidiana que está definida pela Equação 4.

$$D_2(x_i, x_j) = (\sum_k (\chi_{i,k} - \chi_{j,k})^2)^{1/2} = ||x_i - x_j||_2 \tag{4}$$

É importante salientar que *clusterização* envolve o agrupamento de dados não rotulados, ou seja, dados cujas classes não são pré-conhecidas.

6.4.3.1. Algoritmo Simples de Clusterização (ACA)

Neste algoritmo, uma colônia de "agentes formigas" movendo-se aleatoriamente em uma malha (matriz) bidimensional tem a capacidade de pegar itens dentro da malha e movê-los para outras posições da malha. A ideia geral é de que itens isolados devem ser pegos e movidos para locais da malha onde se encontram mais itens daquele mesmo tipo. Note, entretanto, que o grupo ao qual cada item pertence é desconhecido *a priori*. Assuma que existe um único tipo de item no ambiente e que uma determinada quantidade de agentes formiga cuja função é carregar itens de uma posição a outra da malha está disponível. A probabilidade p_p de que uma formiga "descarregada" se movendo aleatoriamente pela malha pegue um determinado item está definida pela Equação 5, onde f é a fração de itens percebidos na vizinhança da formiga, e k_1 é uma constante (*threshold*). Para $f << k_1$, $p_p \approx 1$, ou seja, a probabilidade de uma formiga pegar um item quando há poucos itens em sua vizinhança é grande.

$$p_p = (\frac{k_1}{k_1 + f})^2 \tag{5}$$

A probabilidade p_d de uma formiga "carregada" movendo-se aleatoriamente pelo ambiente deixar este item em uma determinada posição da malha é dada pela Equação 6, onde f é a fração de itens percebidos na vizinhança da formiga e k_2 é outra constante (*threshold*). Para $f << k_2$, $p_d \approx 0$, ou seja, a probabilidade de uma formiga deixar um

item quando há poucos itens em sua vizinhança é pequena.

$$p_d = (\frac{f}{k_2 + f})^2 \tag{6}$$

Para utilizar este modelo teórico como uma ferramenta de *clusterização* (engenharia), ainda é necessário definir dois aspectos importantes: 1) Qual o tipo de ambiente no qual as formigas vão se movimentar? 2) Como definir a função f?

No algoritmo padrão, as formigas movem-se em uma malha bidimensional contendo $m \times m$ células, e possuem a capacidade de perceber o ambiente em uma vizinhança de sua posição atual $V_(s_s)$. Neste caso, os padrões de entrada são projetados em regiões aleatórias da malha e devem posteriormente ser reposicionados de forma a preservar as relações de vizinhança entre itens "vizinhos" no espaço original de atributos. Note que f pode ser entendido como sendo a "visibilidade" de cada formiga. Assim como no caso da função de *fitness* em algoritmos evolutivos, f será uma função do problema a ser tratado. Por exemplo, em um contexto de sistemas robóticos, f foi definido como sendo o quociente entre a quantidade N de itens encontrados nas últimas T iterações do algoritmo e a maior quantidade possível de itens que poderia ser encontrada neste período.

6.4.3.2. Exemplos de Aplicação

[Lumer and Faieta, 1994] aplicaram o algoritmo padrão ao problema de análise exploratória de dados, onde o objetivo era encontrar *clusters* em dados não rotulados. Os dados foram tomados em um espaço Euclidiano de dimensão L, $\Re^L$, e foi utilizado uma malha bidimensional com vizinhança unitária. A função f é dada pela Equação 7, onde $f(x_i)$ é uma medida da similaridade média do item x_i em relação a outro item x_j na vizinhança de x_i, α é um fator que define a escala de dissimilaridade, e $d(x_i, x_j)$ é a distância Euclidiana entre os dados x_i e x_j em seus espaços originais.

$$f(x_i) = \begin{cases} \frac{1}{s^2} \sum_{x_j \in V_{(s \times s)}} (r)[1 - \frac{d(x_i, x_j)}{\alpha}], & \text{se } f > 0 \\ 0, & \text{outros casos} \end{cases} \tag{7}$$

As probabilidades de pegar e deixar um item são dadas pelas Equações 8 e 9 respectivamente.

$$p_p(x_i) = \frac{k_1}{k_1 + f(x_i)} \tag{8}$$

$$p_d(x_i) = \begin{cases} 2f(x_i), & \text{se } f(x_i) < k_2 \\ 1, & \text{se } f(x_i) \geq k_2 s \end{cases} \tag{9}$$

Embora o algoritmo ACA seja capaz de agrupar os dados, ele geralmente encontra uma quantidade de grupos maior do que a existente na base de dados original. Além disso, o algoritmo padrão não estabiliza em uma dada solução, ele fica construindo e

reconstruindo grupos constantemente. Para aliviar estes problemas, [Vizine et al., 2005] propuseram três modificações no algoritmo original:

- Um decaimento para o parâmetro k_1;
- Um campo de visão progressivo que permite uma visão mais abrangente para as formigas; e
- A adição de feromônio aos itens carregados pelas formigas e possibilidade de transferência de feromônio para a malha.

Decaimento de k_1:

- A cada ciclo (10.000 passos de formiga) k_1 sofre um decaimento geométrico: $k_1 \leftarrow 0.98 \times k_1$ até $k_{1min} = 0.001$;
- Quando uma formiga percebe um grupo grande ela aumenta seu campo de visão: se $f(x_i) > \theta$ e $s^2 \leq s^2_{max}$, então $s^2 \leftarrow s^2 + n_s$. A sugestão dos autores é: $s^2_{max} = 7 \times 7$ e $\theta = 0.6$;
- Inspirados pelo processo de realimentação positiva via feromônio no processo de construção de ninhos pelos cupins, os autores propuseram a adição de um nível de feromônio $\phi(i)$ à malha, onde i é o índice da célula da malha. As Equações 10 e 11 apresentam esse conceito.

$$p_p(x_i) = \frac{1}{f(i)\phi(i)}\left(\frac{k_1}{k_1 + f(x_i)}\right)^2 \tag{10}$$

$$p_d(x_i) = \frac{1}{f(i)\phi(i)}\left(\frac{k_2}{k_2 + f(x_i)}\right)^2 \tag{11}$$

Referencias

[Board, 2015] Board, O. A. R. (2015). Openmp application program interface. Technical report.

[Bonabeau et al., 1999] Bonabeau, E., Marco, D. d. R. D. F., Dorigo, M., Théraulaz, G., Theraulaz, G., et al. (1999). *Swarm intelligence: from natural to artificial systems*. Number 1. Oxford university press.

[Castro et al., 2004] Castro, L. N., Hruschka, E. R., and Rosatelli, Marta C. ; Campello, R. J. G. B. (2004). Computação natural: Uma breve visão geral. In *Workshop em nanotecnologia e Computação Inspirada na Biologia*. NanoBIO.

[David, 2016] David (2016). Putting your data and code in order: Optimization and memory – part 1. `https://software.intel.com/en-us/articles/putting-your-data-and-code-in-order-optimization-and-memory-part-1`. Acessado em 2019-01-30.

[De Castro, 2006] De Castro, L. N. (2006). *Fundamentals of natural computing: basic concepts, algorithms, and applications*. Chapman and Hall/CRC.

[Dorigo et al., 1996] Dorigo, M., Maniezzo, V., Colorni, A., et al. (1996). Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, man, and cybernetics, Part B: Cybernetics*, 26(1):29–41.

[Hager and Wellein, 2010] Hager, G. and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

[Kumar, 2002] Kumar, V. (2002). *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[Lumer and Faieta, 1994] Lumer, E. D. and Faieta, B. (1994). Diversity and adaptation in populations of clustering ants. In *Proceedings of the third international conference on Simulation of adaptive behavior: from animals to animats 3: from animals to animats 3*, pages 501–508. MIT Press.

[McCool et al., 2012] McCool, M., Reinders, J., and Robison, A. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Pacheco, 2011] Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Rauber and Rünger, 2013] Rauber, T. and Rünger, G. (2013). *Parallel Programming: For Multicore and Cluster Systems*. Springer Publishing Company, Incorporated, 2nd edition.

[van der Pas et al., 2017] van der Pas, R., Stotzer, E., and Terboven, C. (2017). *Using OpenMP—The Next Step: Affinity, Accelerators, Tasking, and SIMD*. Scientific and Engineering Computation. MIT Press.

[Vizine et al., 2005] Vizine, A. L., De Castro, L. N., Hruschka, E. R., Gudwin, R. R., et al. (2005). Towards improving clustering ants: An adaptive ant clustering algorithm. *Informatica (Ljubljana)*.

[White and Pagurek, 1999] White, T. and Pagurek, B. (1999). Emergent behavior and mobile agents. In *Proc. of the Workshop on Mobile Agents in the Context of Competition and Cooperation at Autonomous Agents*.