Projeto de uma Biblioteca Paralela de Grafos

Phillippe Samer, Afonso H. Sampaio, Anolan Milanés, Sebastián Urrutia
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627 – Pampulha – Belo Horizonte – MG CEP 31270-901
Email: {samer, afonsohs, anolan, surrutia}@dcc.ufmg.br

RESUMO

A teoria dos grafos fornece uma poderosa ferramenta (incluindo teoremas e algoritmos) para a modelagem e resolução de problemas em diversos domínios. Este trabalho apresenta *Magical*, uma nova biblioteca para C++ com implementações paralelas (utilizando OpenMP) de algoritmos em grafos. Até onde sabemos, o panorama de projetos existentes indica a necessidade de uma biblioteca especificamente projetada para explorar o desempenho oferecido pelo paralelismo em arquiteturas multicore, além de fornecer uma interface que facilite seu uso, extensão e integração em implementações existentes. Neste documento, descrevemos o projeto da biblioteca e avaliamos seu desempenho através de um estudo de caso relativo a um problema de caminhos mínimos.

PALAVRAS CHAVE. Biblioteca Paralela. Algoritmos em Grafos. Multicore.

ABSTRACT

Graph Theory provides a set of powerful tools (both theorems and algorithms) for problem modeling and solving in numerous domains. In this paper we describe *Magical*, a new OpenMP-based C++ multicore graph library. To the best of our knowledge, an overview of existing projects indicates a particular need for a library specifically designed to exploit the multicore architecture trend for high performance parallelism, and provide an interface which is easier to use, extend, and integrate into existing implementations. We describe in this document the library design and evaluate its performance by means of a case study concerning a shortest-paths problem.

KEYWORDS. Parallel Library. Graph Algorithms. Multicore.

1. Introdução

Um grafo é uma abstração matemática que representa relações entre entidades. Mais antiga que a maioria das disciplinas em ciência da computação, a teoria dos grafos inclui um grande conjunto de problemas amplamente estudados e algoritmos que fornecem ferramentas poderosas para uma ampla gama de domínios de aplicação. Assim, bibliotecas implementando essas ferramentas desempenham um papel importante em projetos de diversas áreas, tais como pesquisa operacional, redes complexas e mineração de dados. No entanto, conforme utiliza-se algoritmos em grafos para atacar instâncias mais difíceis dos problemas, o tempo de execução das aplicações pode tornar-se proibitivo.

Atualmente, é comum encontrar computadores pessoais com vários núcleos de processamento em um único chip. No entanto, melhorar o desempenho de aplicações a partir do uso de arquiteturas *multicore* não é uma tarefa simples. A fim de ser capaz de explorar o paralelismo fornecido por estes ambientes, programas e bibliotecas devem ser adaptados ou mesmo projetados novamente.

Existem várias bibliotecas implementando algoritmos em grafos, tanto sequenciais como paralelas. Entretanto, até onde sabemos, apenas uma (a MTGL, vide seção 2.2) está preparada para se beneficiar de arquiteturas multicore. Como discutimos à frente, a maior parte das implementações paralelas atualmente disponíveis assume uma arquitetura de memória distribuída, exigindo o uso de estruturas de dados serializados e introduzindo *overhead* devido à troca de mensagens. Além disso, o uso dessas bibliotecas é consideravelmente difícil. Por exemplo, a instalação e uso de bibliotecas baseadas em Boost envolve a configuração e ajustes com flags para ligação de diversos pacotes. Mesmo no caso da MTGL, a despeito de um forte histórico em computação de alto desempenho, é possível que alguns de seus princípios de projeto dificultem sua integração para uso em implementações já existentes.

Neste trabalho, propomos uma abordagem diferente: fornecer uma biblioteca paralela para grafos que é simultaneamente fácil de usar e especificamente projetada para arquiteturas multicore. De fato, estas são as principais diretrizes do projeto Magical (*Multicore Appropriate Graph Interface and Collection of Algorithms*). Primeiro, temos como arquitetura alvo um sistema de memória compartilhada, dada a atual falta de implementações capazes de explorar o potencial desempenho das modernas plataformas multicore. Além disso, nos concentramos em oferecer uma interface de programação (API) fácil de usar, habilitando implementações atuais a se beneficiar facilmente de um sistema com múltiplos núcleos. Em particular, não é exigido que o usuário tenha conhecimento de programação paralela.

A contribuição de Magical consiste em proporcionar um conjunto de implementações paralelas em teoria dos grafos e recursos que podem ser facilmente utilizados por outras aplicações (futuras ou já existentes) para explorar o desempenho de sistemas multicore. Nós descrevemos como a nossa biblioteca de código aberto oferece uma interface simples, em termos de esforço de programação necessário para integrá-la ao código existente. São apresentados também resultados computacionais preliminares relativos aos algoritmos inicialmente implementados como parte da biblioteca.

A organização deste trabalho é a seguinte. Na seção 2 revisamos trabalhos relacionados, com um panorama das bibliotecas existentes. A seção 3 descreve o projeto da biblioteca proposta, incluindo sua modelagem e arquitetura. Na seção 4 apresentamos um estudo de caso sobre algoritmos de caminhos mínimos, descrevendo

resultados computacionais. Por fim, as conclusões e trabalhos futuros são discutidos na seção 5.

2. Trabalhos Relacionados

Várias bibliotecas implementando algoritmos em grafos têm sido propostas na literatura, tanto sequenciais como paralelas. Projetos existentes abrangem diferentes linguagens de programação e plataformas, embora os mais maduros sejam implementados em C++ ou Java, conforme descreve de Heus (2011). Avalia-se a seguir tais bibliotecas em termos das seguintes características:

- Paralelização: se é puramente sequencial ou oferece implementações paralelas.
- Flexibilidade: se fornece uma interface genérica, permitindo ao usuário estender ou adaptá-la a necessidades específicas.
- Recursos: se são oferecidos tanto algoritmos quanto estruturas de dados.
- Disponibilidade: se encontra-se disponível atualmente, se oferece código livre e se tem uma licença comercial.

2.1. Boost

Boost é uma coleção aberta e gratuita de bibliotecas estendendo a funcionalidade de C++, entre as quais existe a BGL – *Boost Graph Library*, apresentada em Siek et al. (2002). Como a maioria dos recursos de Boost, a BGL é concebida com um forte investimento em programação genérica, oferecendo tanto estruturas de dados para grafos e uma coleção de algoritmos, além de sua versão paralela (PBGL – *Parallel Boost Graph Library*), descrita por Gregor e Lumsdaine (2005).

Projetada para computação distribuída, a interface de programação da PBGL envolve o uso de estruturas de dados distribuídas e grupos de processos, como esperado em um ambiente de memória distribuída. No entanto, em um ambiente multicore introduz-se com isso tanto overhead como esforço de programação indesejado na passagem da BGL sequencial.

Similarmente, as bibliotecas ParGraph (2012) e *CGM Graph Library* (ver Chan et al. (2005)) fornecem implementações paralelas para sistemas distribuídos. ParGraph também é baseada na BGL, mas torna os mecanismos de comunicação mais explícitos que a PBGL. Já a CGM é construída sobre um modelo de comunicação próprio, também usando MPI para troca de mensagens.

Até onde entendemos, os objetivos do projeto dessas bibliotecas não são apropriados para sistemas multicore, e não atendem à proposta de explorar todo o potencial de um processador moderno.

2.2. MTGL

A MTGL (*MultiThreaded Graph Library*), descrita por Barrett et al. (2009), é desenvolvida pelo *Sandia National Laboratories* desde 2008. É uma biblioteca aberta para C++, e oferece algoritmos paralelos para grafos e estruturas de dados.

Embora inspirada no projeto Boost (incluindo containers genéricos de grafos), a MTGL tem como alvo plataformas de memória compartilhada. Primeiramente, foi concebida apenas para arquiteturas de supercomputadores massivamente paralelos, como a série Cray MTA/XMT. Em seguida foi portada para processadores multicore e SMP por meio da biblioteca QThreads, também do Sandia Labs. QThreads oferece

Tabela 1: Bibliotecas com implementações de algoritmos em teoria dos grafos.

Biblioteca	Linguagem	Paralelismo	Flexibilidade	Recursos	Disponibilidade
PBGL	C++	✓	✓	✓	livre, gratuita
MTGL	C++	✓	✓	✓	livre, gratuita
STAPL	C++	✓	✓	✓	indisponível
LEDA	C++	✗	✓	✗	versão gratuita limitada
LEMON	C++	✗	✓	✓	livre, gratuita
GTL	C++	✗	✓	✓	indisponível
Ocamlgraph	OCaml	✗	✓	✓	livre, gratuita
JGraphT	Java	✗	✓	✓	livre, gratuita
yFiles	Java	✗	✗	✓	versão gratuita limitada

uma interface para programação paralela em sistemas de memória compartilhada, baseada na execução de uma quantidade massiva de threads de usuário.

Portanto, a MTGL e Magical compartilham objetivos em relação às características consideradas. Além disso, a MTGL é um projeto maior (dadas as suas aplicações em larga escala em projetos relacionados do laboratório) e em desenvolvimento ativo. No entanto, nosso projeto visa fornecer uma ferramenta particularmente mais fácil de usar em pesquisas e aplicações sobre teoria dos grafos (ver Seção 3.2 para uma descrição da API da biblioteca), contribuindo assim com uma opção diferente para explorar paralelismo multicore. Finalmente, por basear-se em um pacote popularmente consolidado como OpenMP em vez de uma API própria (QThreads), é possível que Magical ofereça um ambiente mais favorável para o desenvolvimento colaborativo, facilitando a participação de novos desenvolvedores.

2.3. Outras Bibliotecas

Outras bibliotecas relacionadas incluem: STAPL descrita por Buss et al. (2010), LEDA (ver Mehlhorn et al. (1997)), LEMON, vide Dezső et al. (2010), GTL – Graph Template Library (2011), Ocamlgraph, vide Conchon et al. (2007), JGraphT (2011) e yFiles for Java (2011). Entretanto, nenhuma destas bibliotecas pode ser comparada diretamente com Magical, seja por não oferecer implementações paralelas, seja por não estar disponível atualmente. A tabela 1 sintetiza uma breve descrição acerca de tais projetos segundo os critérios estabelecidos no começo desta seção.

Finalmente, projetos mais específicos incluem: Combinatorial Blas, para análise de grafos e mineração de dados, descrita por Aydin Bulu, John R. Gilbert (2011); SWARM, um arcabouço para programação paralela, utilizado na implementação de alguns algoritmos básicos envolvendo grafos; e METIS, uma coleção de algoritmos para particionamento de grafos, vide Karypis e Kumar (1995). Embora usem implementações paralelas, o foco de tais projetos é fornecer softwares para aplicações específicas, o que é bastante diferente da proposta de Magical.

3. Projeto da Biblioteca

Esta seção apresenta as principais características da biblioteca proposta. Descrevemos a arquitetura concebida para Magical e sua interface de programação de aplicação (API), que caracterizam as ideias centrais motivando este trabalho e sua contribuição.

3.1. Paralelização para Sistemas Multicore

Conforme descrito na seção 2, dentre todas as bibliotecas de grafos que pudemos relacionar, apenas a MTGL é projetada para explorar a tendência de arquiteturas multicore para paralelismo de alto desempenho. Além disso, o uso excessivo de metaprogramação e outras técnicas avançadas para genericidade em C++ nos projetos

Tabela 2: Algoritmos atualmente disponíveis na biblioteca Magical.

Problema	Autor do Algoritmo	Implementação
Caminhos Mínimos de Um para Todos	Dijkstra	Sequencial
	Bellman-Ford	Sequencial
Caminhos Mínimos de Todos para Todos	Johnson	Paralela
Árvore Geradora Mínima	Borůvka	Paralela
Tour Euleriano	Hierholzer	Paralela

existentes, como a PBGL (ver seção 2.1), incorre em notável perda de legibilidade, vide Google C++ Style Guide (2012). Assim, Magical foi projetada desde o início para facilitar a programação paralela de algoritmos em grafos a partir de sistemas multicore: desde a codificação, por meio da interface descrita na seção a seguir, até o tempo de compilação (exigindo apenas flag de opção do OpenMP) e execução, e.g. configurando número de threads e escalonamento por padrão (ajustável via arquivo de configuração).

Em termos de arquitetura, considerando o foco em arquiteturas de memória compartilhada, optamos pela paralelização por meio de *threads* de execução. A programação com threads é sabidamente difícil e propensa a erros, conforme descreve Lee (2006). Entretanto, esta opção foi feita visando a facilidade de uso da nova biblioteca: se por um lado a escolha por troca de mensagens tornaria nossa biblioteca mais previsível, por outro provavelmente tornaria sua utilização mais complexa.

Em particular, Magical é desenvolvida em C++ padrão, empregando OpenMP 3.0 para programação paralela (ver OpenMP Architecture Review Board (2008)). Além de fornecer uma API de alto nível, reduzindo a possibilidade de erros como sincronização, OpenMP é um padrão livre e padronizado pelos principais compiladores da linguagem. Através de nossa experiência avaliando as diferentes bibliotecas, consideramos que a instalação e configuração de outros pacotes pode ser muito difícil. Por exemplo, embora teoricamente o uso de Boost requeira apenas instalação de pacotes e ligação com a biblioteca, diferenças na configuração do sistema (e.g. dependências, conflitos entre versões de pacotes, locais de instalação) podem tornar o processo oneroso; no caso de máquinas compartilhadas e com restrições de permissão, o seu uso pode ser realmente desafiador.

Já em termos de algoritmos paralelos, deve estar claro neste ponto que este trabalho e a biblioteca Magical não visam superar exemplos de sofisticadas paralelizações descritas na literatura, que alcançam resultados notáveis por meio de implementações altamente especializadas para certas arquiteturas computacionais. De fato, Magical deve oferecer uma alternativa simples para projetos utilizando algoritmos em grafos, a fim de melhor utilizar o desempenho oferecido por sistemas multicore.

A atual coleção de algoritmos incluídos na biblioteca é listada na tabela 2, sendo todo o projeto é disponibilizado abertamente no sítio:
`http://code.google.com/p/magical-project/` .

3.2. Interface da Biblioteca

O projeto da API da biblioteca é fundamental, procurando facilitar sua utilização e proporcionar uma ferramenta geral para a programação de algoritmos em grafos. Neste sentido, a interface de Magical permite que o paralelismo da biblioteca seja transparente para o usuário e sua aplicação. Da mesma forma, o projeto visa reduzir

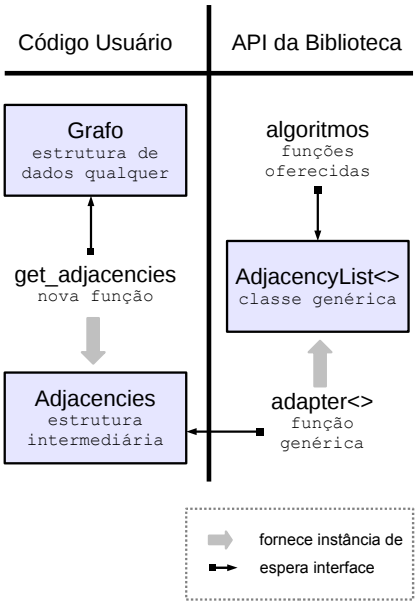


Figura 1: Modelo alternativo para uso com estruturas de dados existentes.

tanto quanto possível o esforço de programação do usuário na integração da biblioteca em códigos existentes. Por exemplo, consideramos inviável exigir que o usuário remodele uma aplicação anterior ou que tenha que lidar com questões de paralelização (como na interface PBGL, indicado na seção 2.1), i.e. defendemos uma API projetada para facilidade de uso.

Com relação a estruturas de dados, este projeto oferece dois modelos de interface com a biblioteca. Primeiro, containers genéricos para grafos são fornecidos de modo a permitir que um novo projeto possa rapidamente utilizar algoritmos em Magical. Neste caso, dispondo das estruturas de dados fornecidas, utiliza-se diretamente as funções da biblioteca, cuja implementação está pronta para execução em uma arquitetura multicore. Implementadas como classes genéricas, tais containers têm tipos de dados padrão para a representação de vértices e arestas, que podem ser substituídos ou estendidos para incluir informações específicas da aplicação.

Alternativamente, um projeto existente baseado em estrutura de dados anteriores pode tirar proveito de Magical por meio de um simples *visitante* do grafo. Tal visitante consiste de uma única função, fornecendo uma visão específica do grafo atual. Do ponto de vista da biblioteca, o mecanismo é semelhante ao padrão de projeto *adaptador* descrito por Gamma et al. (1994), já que a estrutura de dados do cliente é adaptada para oferecer a interface esperada pelos algoritmos da biblioteca. Já para o usuário, trata-se apenas de um procedimento indicando vértices e adjacências em sua representação original.

O diagrama da figura 1 ilustra esta interface. Os algoritmos da biblioteca esperam que o grafo esteja representado como uma instância da classe genérica *AdjacencyList*, seja porque a aplicação do usuário foi originalmente projetada com ela (primeiro modelo de interface), seja porque uma estrutura de dados anterior foi adaptada (segunda alternativa). Esta alternativa se dá por meio da função genérica *adapter* de Magical, que fornecerá o objeto esperado desde que possa visitar as adjacências do grafo original.

Algoritmo 1: Procedimento geral do visitante para usar a biblioteca pela interface que adapta uma representação existente.

entrada: grafo $G(V,E)$ e custos $\omega(i,j)$, $\forall(i,j) \in E$
saída : instância *adaptee* da classe *AdjacencyList* da biblioteca

```
1 para cada u ∈ V faça
2     Adj[u] ← ∅
      // registra nós vizinhos
3     para cada v ∈ V : (u,v) ∈ E faça
4         Adj[u] ← Adj[u] ∪ {(v, ω(u,v))}
  // procedimento fornecido pela biblioteca
5 adaptee ← adapter(Adj, |V|)
6 retorna adaptee
```

Tal representação intermediária de adjacências é o que se requer do usuário para a completa utilização da biblioteca, e foi projetada para exigir o mínimo de esforço. Especificamente, ela envolve percorrer o grafo coletando adjacências de cada nó. No diagrama da figura 1, a função *get_adjacencies* é responsável por fornecer esta representação, e o algoritmo 1 descreve como deve ser implementada. Argumentamos que esta função pode ser facilmente programada pelo usuário porque requer a compreensão apenas de sua própria estrutura de dados original. Na página do projeto (`http://code.google.com/p/magical-project/`) incluimos exemplos práticos do uso da biblioteca, ilustrando que a função a ser programada é bastante simples. De fato, exemplos com menos de 30 linhas de código mostram como duas aplicações distintas usariam a biblioteca.

Ressaltamos que a alternativa de interface com adaptador não é nem inédita (mas sim baseada em um padrão de projeto, como descrito anteriormente nesta seção), nem a mais eficiente assintoticamente. A complexidade de tempo envolvida na adaptação tem custo de $O(|V|+|E|)$, ou $O(|V|^2)$ para uma representação matricial. Desenvolvemos esta interface considerando que o custo de tradução é amortizado pelo uso de um algoritmo de maior complexidade, tal como no estudo de caso apresentado a seguir. Além disso, a opção representa um compromisso em relação à simplicidade do uso de Magical e ao reduzido esforço de programação que a atual interface da biblioteca requer. Entre trabalhos futuros no projeto, prevemos estender a interface para incluir um modelo que apenas referencia a própria estrutura de dados do usuário; embora esta alternativa certamente exija maior trabalho do usuário, dispensa o custo de tradução descrito.

4. Estudo de Caso: Caminhos Mínimos de Todos para Todos

O problema de caminhos mínimos e suas variações estão entre os problemas fundamentais em otimização combinatória. Com muitas aplicações diferentes e uma área de pesquisa ativa, são frequentemente utilizados na avaliação de implementações em grafos, vide Dezső et al. (2010).

Para uma avaliação inicial, abordamos o problema de caminhos mínimos de todos para todos (*all-pairs shortest-paths* – APSP). No caso em que o grafo $G(V,E)$ não apresenta arcos de custo negativo, basta aplicar um algoritmo de caminhos mínimos de um para todos (*single-source shortest-paths* – SSSP), a partir de cada vértice. Pode-se então escolher o algoritmo de Dijkstra, que oferece a solução mais eficiente assintoticamente para o problema de SSSP: $O(|E|+|V| \times \lg|V|)$ utilizando heaps de

Algoritmo 2: Algoritmo de Johnson para o APSP.

Entrada: grafo $G(V,E)$, com função de custo $\omega : E \to \Re$
Saída : comprimento dos caminhos mínimos $d(i,j)$, $\forall (i,j) \in V \times V$, ou *falso* se G contém circuito de custo negativo (i.e. não existe solução)

1 adicione a V um vértice artificial s, e inclua em E arcos (s,i) de custo $\omega(s,i) = 0, \forall i \in V$
2 execute o algoritmo de *Bellman-Ford* com vértice de origem s, obtendo $\delta(i)$, $\forall i \in V$
3 **se** *Bellman-Ford retorna falso* **então**
  `// G contém circuito negativo`
4 **retorna** *falso*
5 **senão**
6 **para cada** $(i,j) \in E$ **faça**
    `// nova função de custo`
7 $\hat{\omega}(i,j) \leftarrow \omega(i,j) + \delta(i) - \delta(j)$
8 **para cada** $i \in V$ **faça**
    `// definição dos caminhos mínimos`
9 execute o algoritmo de *Dijkstra* com vértice de origem i usando a função de custo $\hat{\omega}$, obtendo $\hat{d}(i,j)$, $\forall j \in V$
10 $d(i,j) \leftarrow \hat{d}(i,j) + \delta(j) - \delta(i)$

Fibonacci. Já no caso em que G inclui arcos negativos, torna-se necessário outro algoritmo para o problema de APSP. Johnson (1977) apresenta uma elegante solução, permitindo arcos de custos negativos e mantendo a mesma complexidade assintótica de $|V|$ aplicações do algoritmo de Dijkstra: $O(|V| \times |E| + |V|^2 \times \lg |V|)$, a menor conhecida para o problema de APSP, vide Cormen et al. (2009).

O algoritmo 2 apresenta o procedimento, que é baseado na técnica de reescritura dos custos dos arcos para valores não negativos, mas preservando os caminhos mais curtos do grafo original. Define-se inicialmente um vértice artificial s com arcos de custo nulo conectando-o a todos os outros vértices. Utiliza-se então o algoritmo de Bellman-Ford para o problema de SSSP para determinar o comprimento $\delta(i)$ dos caminhos mais curtos a partir de s até cada outro vértice i. Destaca-se que o uso do algoritmo de Bellman-Ford fornece tanto uma verificação da existência de circuitos negativos no grafo original quanto um conjunto específico de valores δ associados a cada par de vértices, que permitem finalmente mapear custos não negativos para os respectivos arcos. Deste modo, o algoritmo de Dijkstra para SSSP pode ser aplicado a partir de cada vértice no grafo original, determinando caminhos mais curtos equivalentes. Cormen et al. (2009) fornece mais detalhes sobre o algoritmo e a técnica de redefinição de custos dos arcos.

A partir de um grafo $G(V,E)$, a complexidade assintótica do algoritmo é fornecida pelas $|V|$ aplicações do algoritmo de Dijkstra a partir de cada vértice de origem. Assim, o tempo de execução de sua implementação depende principalmente do laço iniciando na linha 8 do algoritmo 2. Sendo as iterações deste laço independentes umas das outras, é possível executá-las em paralelo. No caso de um ambiente multicore, utilizamos primitivas para distribuição da carga do trabalho (*worksharing*) como diretivas de paralelismo de laços em OpenMP.

Avaliamos o desempenho da implementação paralela por meio do *speedup* e da eficiência. O primeiro consiste da razão entre o tempo de execução da implementação sequencial (T_{seq}) e da versão paralela utilizando n núcleos de processamento (T_n), i.e.

$S_n = T_{seq}/T_n$. Já a eficiência é definida por $E_n = S_n/n$ (de forma que $0 < E_n \leq 1$).

Os experimentos utilizam um conjunto de três máquinas, descritas na tabela 3. Usamos um sistema Ubuntu Server 10.04 LTS (GNU/Linux kernel 2.6.32-32), e todas as medidas de tempo se referem ao tempo de relógio (real) decorrido.

Instâncias de entrada foram retirados da TSPLib, um conjunto de instâncias para as diferentes variantes do problema do caixeiro viajante, descrita por Reinelt (1995) e amplamente utilizada na literatura. Em particular, todas as instâncias utilizadas neste trabalho são simétricas e o grafo subjacente é completo. Como indicado na tabela 4, o identificador de cada instância inclui a sua dimensão (i.e. número de vértices) como um sufixo.

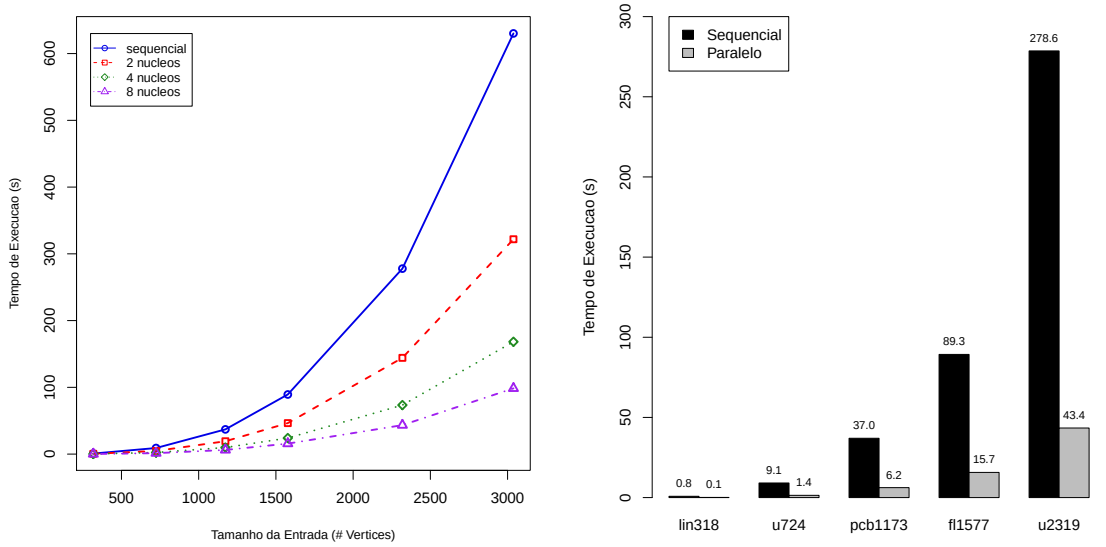
A figura 2(a) apresenta o tempo de execução com grafos de tamanho crescente. Verifica-se que a implementação se beneficia da quantidade de núcleos com speedup um pouco abaixo do ideal (linear). Percebemos também um melhor desempenho em execuções que exigem maior processamento, já que o aumento do tamanho da entrada parece amortizar o overhead de processamento paralelo.

Os valores na tabela 4 sumarizam o speedup e eficiência obtidos quando utilizando a máquina III. Embora a paralelização indique um ganho de desempenho escalável até 8 núcleos, verificamos uma redução na taxa de eficiência com esta configuração, uma vez que passa-se a utilizar núcleos em ambos processadores (existem 4 em cada, vide tabela 3), refletindo as diferentes latências de memória cache envolvidas.

Por fim, para avaliar o desempenho da biblioteca contra uma implementação puramente sequencial, que justifica seu uso e desenvolvimento, executamos uma série

Tabela 3: Processadores das máquinas utilizadas.

Modelo	Clock	#Núcleos	Cache
Intel Core 2 Duo E8400	3.0 GHz	2	6 MB
Intel Xeon E5520	2.26 GHz	4	8 MB
2 x Intel Xeon E5620	2×2.4 GHz	2×4	2×12 MB



(a) Tempo para diferentes entradas e número de núcleos.

(b) Execução na Máquina III (8 núcleos).

Figura 2: Tempos de execução verificados com a Máquina III (CPU com 8 núcleos).

Tabela 4: Speedup (S_n) e eficiência (E_n) verificados ao utilizar n núcleos da máquina III, para entradas de tamanho crescente.

Instância	S_2 (E_2)	S_4 (E_4)	S_8 (E_8)
lin318	1.88 (94%)	3.55 (89%)	5.91 (74%)
u724	1.91 (96%)	3.79 (95%)	6.51 (81%)
pcb1173	1.92 (96%)	3.77 (94%)	5.98 (75%)
fl1577	1.92 (96%)	3.72 (93%)	5.69 (71%)
u2319	1.93 (96%)	3.78 (95%)	6.42 (80%)
pcb3038	1.96 (98%)	3.74 (94%)	6.31 (79%)

de testes utilizando as três máquinas descritas. Os gráficos nas figuras 2(b), 3(a) e 3(b) permitem comparar os tempos de execução para diferentes instâncias em cada máquina. Destaca-se que, pelas diferenças na arquitetura de cada processador, os valores indicados nestas figuras diferem daqueles obtidos ao utilizar menos núcleos de uma mesma plataforma (como na tabela 4). Argumenta-se com estes resultados que, mesmo no caso do processador de menor capacidade, pode-se obter desempenho superior utilizando a biblioteca paralela.

5. Conclusão

Este artigo descreve Magical, um projeto em andamento visando a concepção e implementação de uma biblioteca de algoritmos em grafos para sistemas multicore. Até onde sabemos, não existia uma alternativa de biblioteca preparada para se beneficiar do paralelismo de memória compartilhada e que oferecesse uma interface de programação amigável (facilitando sua integração em projetos existentes).

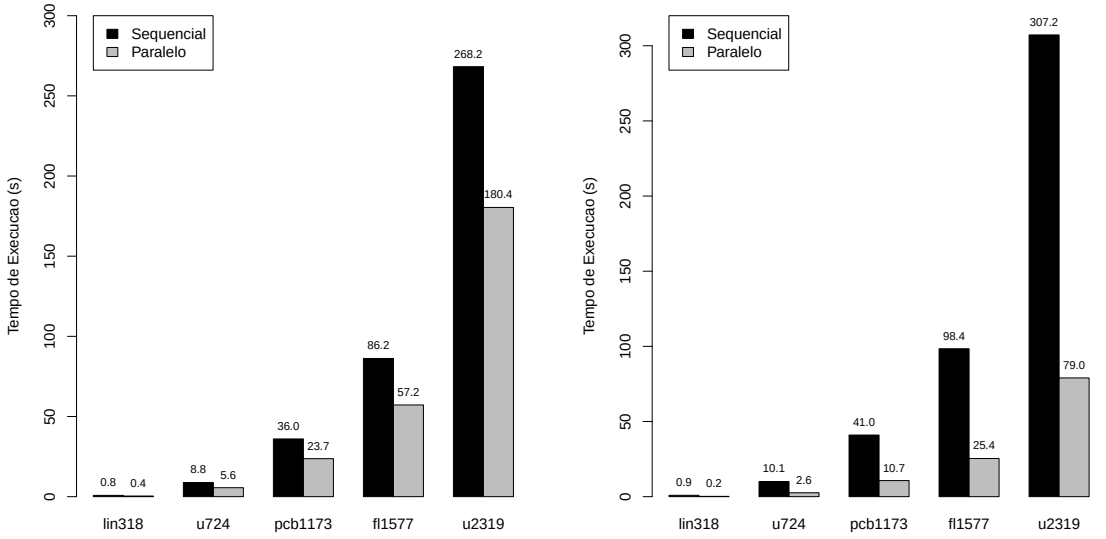
A relevância desta proposta reside na ubiquidade dessas arquiteturas e na importância para projetos de uma série de domínios de aplicação da existência de mecanismos para usufruir destas plataformas de maneira mais fácil e eficaz. Nossa biblioteca de código aberto é projetada tanto para explorar adequadamente o paralelismo de memória compartilhada quanto para oferecer uma interface de programação que facilite sua utilização. A nossa proposta encapsula a complexidade da programação paralela nas mãos dos programadores da biblioteca.

Apresentamos neste trabalho a arquitetura da biblioteca, descrevendo as decisões de projeto que refletem nossos objetivos. Fornecemos também uma descrição geral sobre a interface proposta e um estudo de caso sobre um conjunto inicial de algoritmos implementados (problemas de caminhos mais curtos).

Os resultados experimentais em instâncias do problema do caixeiro viajante mostraram um ganho de desempenho quase linear com o número de núcleos do processador. A biblioteca viabiliza redução de tempos de execução quando utilizando diferentes arquiteturas e instâncias do problema. Os trabalhos futuros incluem a avaliação do desempenho e escalabilidade comparada às outras bibliotecas de grafos, aumentar o conjunto de algoritmos paralelos implementado na biblioteca e oferecer interfaces alternativas para a integração em projetos existentes.

Agradecimentos

Agradecemos ao incentivo relativo à bolsa de iniciação científica da Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG), que viabilizou parte do presente trabalho.



(a) Execução na Máquina I (2 núcleos).

(b) Execução na Máquina II (4 núcleos).

Figura 3: Comparação dos tempos de execução de uma implementação puramente sequencial e da biblioteca paralela, para diferentes instâncias e máquinas.

Referências

Aydin Bulu, John R. Gilbert (2011), 'The combinatorial blas: design, implementation, and applications', *International Journal of High Performance Computing Applications*.

Barrett, B. W., Berry, J. W. e Richard C. Murphy, K. B. W. (2009), 'Implementing a portable Multi-threaded Graph Library: The MTGL on Qthreads', *Parallel and Distributed Processing Symposium, International* **0**, 1–8.

Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N. M. e Rauchwerger, L. (2010), STAPL: standard template adaptive parallel library, *in* 'Proceedings of the 3rd Annual Haifa Experimental Systems Conference', SYSTOR '10, ACM, New York, NY, USA, pp. 14:1–14:10.

Chan, A., Dehne, F. e Taylor, R. (2005), CGMgraph/CGMlib: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines, *in* 'International Journal of High Performance Computing Applications', Springer.

Conchon, S., Filliâtre, J.-C. e Signoles, J. (2007), 'Designing a generic graph library using ML functors', *Selected papers from the Eighth Symposium on Trends in Functional Programming (TFP07)* **8**, 141–158.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. e Stein, C. (2009), *Introduction to Algorithms*, 3rd edn, The MIT Press.

de Heus, M. (2011), *Towards a Library of Parallel Graph Algorithms in Java.*, Bachelor's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands.

Dezső, B., Jüttner, A. e Kovács, P. (2010), *LEMON – an open source c++ graph template library, Workshop on generative technologies (WGT), Paphos, Cyprus.*

Gamma, E., Helm, R., Johnson, R. e Vlissides, J. M. *(1994),* Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edn, Addison-Wesley Professional.*

Google C++ Style Guide *(2012), 'On the use of Boost libraries'. <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml?showone=Boost#Boost>. Última visita em 30 de janeiro, 2012.*

Gregor, D. e Lumsdaine, A. *(2005), The Parallel BGL: A generic library for distributed graph computations,* in *'Parallel Object-Oriented Scientific Computing (POOSC)'.*

GTL – Graph Template Library *(2011). <www.fim.uni-passau.de/en/fim/faculty/chairs/theoretische-informatik/projects.html>. Última visita em 30 de junho, 2011.*

JGraphT *(2011). <http://www.jgrapht.sourceforge.net>. Última visita em 30 de junho, 2011.*

Johnson, D. B. *(1977), 'Efficient algorithms for shortest paths in sparse networks',* Journal of the ACM **24***, 1–13.*

Karypis, G. e Kumar, V. *(1995), Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0, Technical report, Karypsis Lab, University of Minnesota.*

Lee, E. A. *(2006), 'The problem with threads',* Computer **39***(5), 33–42.*

Mehlhorn, K., Näher, S. e Uhrig, C. *(1997), The LEDA platform for combinatorial and geometric computing,* in *P. Degano, R. Gorrieri e A. Marchetti-Spaccamela, eds, 'Automata, Languages and Programming', Vol. 1256 of* Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 7–16.*

OpenMP Architecture Review Board *(2008), 'OpenMP Application Programming Interface, version 3.0'.*

ParGraph *(2012). <http://pargraph.sourceforge.net>, por F. Hielscher e P. Gottschling. Última visita em 30 de janeiro, 2012.*

Reinelt, G. *(1995),* TSPLib 95*, Universität Heidelberg.*

Siek, J., Lee, L.-Q. e Lumsdaine, A. *(2002),* The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley.*

yFiles for Java *(2011). <www.yworks.com/en/products>. Última visita em 30 de junho, 2011.*