Paralelização de Algoritmos de Enumeração para o Problema do Vector mais Curto em Sistemas de Memória Partilhada e Distribuída

Fábio Correia[1,2], Artur Mariano[2], and Alberto Proença[1]

[1] Departamento de Informática, Universidade do Minho
[2] Institute for Scientific Computing, Technische Universität Darmstadt
pg22817@alunos.uminho.pt,artur.mariano@sc.tu-darmstadt.de
aproenca@di.uminho.pt

Resumo A criptografia baseada em retículos tem vindo a tornar-se um tópico central ao longo da última década, uma vez que se acredita que este tipo de criptografia seja resistente a ataques infligidos com computadores quânticos. A segurança desta criptografia é medida pela eficácia e praticabilidade dos algoritmos que resolvem problemas centrais em retículos, como o problema do vector mais curto (PVC), e é, por isso, importante determinar qual o desempenho máximo destes algoritmos em arquitecturas computacionais de alto rendimento.
Neste sentido, este artigo apresenta, pela primeira vez, um estudo detalhado sobre o desempenho dos dois mais promissores algoritmos de resolução do PVC, o ENUM e uma variante eficiente da enumeração de Schnorr-Euchner, com e sem poda extrema. Em particular, são propostas versões paralelas destes algoritmos, desenvolvidas para óptimo balanço de carga e, consequentemente, melhor desempenho.
Conduziu-se uma extensa série de testes, quer em memória partilhada, para as variantes sem poda, quer em memória distribuída, para as variantes com poda. Os resultados mostram que as implementações em memória partilhada atingem, em certos casos, acelerações lineares até 16 *threads*. As implementações em memória distribuída, por seu turno, são aceleradas em cerca de 13 vezes para 16 processos, permitindo a resolução do PVC em retículos em dimensão 80 em menos de 250 segundos.

Keywords: PVC, Enumeração, Paralelização, OpenMP, MPI

1 Introdução

Retículos são sub-grupos discretos do espaço Euclidiano $\mathbb{R}^n$, com uma forte propriedade de periodicidade. Um retículo Λ, gerado por uma base $\mathbf{B} \in \mathbb{R}^{n \times m}$, que é por sua vez um conjunto de vectores linearmente independentes $\mathbf{b}_1,\ldots,\mathbf{b}_m$, em $\mathbb{R}^n$, é denotado por:

$$\Lambda(\mathbf{B}) = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{x} = \sum_{i=1}^{m} \mathbf{u}_i \mathbf{b}_i, \mathbf{u} \in \mathbb{Z}^m\}, \quad (1)$$

onde m é chamado o *rank* do retículo. Quando $n = m$, o retículo tem a propriedade *full rank*. Quando $n \geq 2$, o retículo possui infinitas bases diferentes.

Os retículos têm um vasto conjunto de aplicações, que vão desde a matemática à ciência da computação. Desde a última década, os retículos têm vindo a ganhar notoriedade pela sua aplicação em sistemas de criptografia, originando inclusivamente um campo em rápida expansão na última década: a criptografia baseada em retículos. O início do uso dos retículos em criptografia data de inícios dos anos 80, quando o algoritmo Lenstra–Lenstra–Lovász (LLL) foi usado para quebrar criptosistemas baseados no problema da mochila. No entanto, foi em meados dos anos 90 que os retículos ficaram em definitivo associados à criptografia, após vários esquemas de encriptação serem propostos [3,7,6].

Hoje em dia, a criptografia baseada em retículos é especialmente atractiva devido a uma crença global na sua imunidade contra ataques operados com computadores quânticos. Os sistemas criptográficos baseados em retículos podem apenas ser quebrados se certos problemas em retículos forem resolvidos num determinado espaço de tempo. Neste contexto, há dois problemas particularmente relevantes: o problema do vector mais próximo (PVP) e o problema do vector mais curto (PVC), o foco deste artigo. O PVC consiste em encontrar o vector mais curto de um retículo, cuja norma é denotada por $\lambda_1(\Lambda)$, ou, por outras palavras, encontrar o vector $\mathbf{x} \in \mathbb{Z}^n$ que minimiza a norma $\|\mathbf{uB}\|$. É, por isso, muito importante estudar o quão eficientes estes algoritmos podem ser implementados em arquitecturas modernas de alto desempenho, uma vez que isso determina a segurança dos sistemas criptográficos baseados em retículos.

Algoritmos que resolvem estes problemas são por norma ditos algoritmos de PVP e PVC, respectivamente. Do ponto de vista computational, é sabido que a variante de decisão do PVP é NP-difícil, enquanto a variante de decisão do PVC é NP-difícil apenas em reduções aleatórias [2]. Tratam-se, por isso, de problemas muito difíceis de resolver computacionalmente, pelo menos em retículos de dimensões altas. Existe ainda uma estreita relação entre os dois: o vector mais próximo da origem, excluíndo a mesma, é o vector mais curto do retículo.

O tempo de execução dos algoritmos que resolvem estes problemas pode ser reduzido ao aplicar algoritmos de redução das bases dos retículos, i.e., algoritmos que transformam uma base num conjunto de vectores curtos e aproximadamente ortogonais, também estes linearmente independentes. Assim, quanto mais ortogonais e curtos os vectores resultantes deste processo forem melhor a qualidade da base do retículo e, consequentemente, menor o tempo de execução dos algoritmos de resolução dos problemas mencionados anteriormente. Os dois algoritmos mais utilizados actualmente para este efeito são os algoritmos LLL e Korkine-Zolotarev por blocos (BKZ). Este último pode gerar bases com melhor ou menor qualidade dependendo de um parâmetro denominado por tamanho de bloco (β).

De todos os algoritmos que resolvem o PVC, destacam-se os algoritmos de enumeração, que com poucas mudanças foram transformados numa heurística. Esta heurística, chamada poda extrema, é actualmente a mais eficiente no que diz respeito à resolução do PVC. Nesta categoria de algoritmos destacam-se principalmente o ENUM, proposta em [10] como sub-rotina do algoritmo de

redução BKZ, e uma variante do mesmo, originalmente proposta por Agrell et al. [1] e posteriormente melhorada em [5], daqui em diante designada de SE++. Os algoritmos são, de resto, descritos pormenorizadamente na Secção 4. Até hoje não foram publicados nem estudos comparativos dos dois algoritmos, nem estudos sobre a escalabilidade de ambas as variantes em sistemas multiprocessamento.

Nesse sentido, procuramos com este artigo responder a duas perguntas:

- Qual destas duas variantes é algoritmicamente mais eficiente e atinge o melhor desempenho entre versões sequenciais com mesmo grau de optimização?
- Quão eficientemente podem estas duas variantes ser paralelizadas, i.e., qual delas é mais escalável em sistemas multiprocessamento?

O remanescente deste artigo está organizado da seguinte forma. A Secção 2 introduz a notação usada neste artigo. A Secção 3 enuncia algoritmos e implementações já existentes que resolvem o PVC. A Secção 4 contém uma descrição detalhada dos algoritmos SE++ e ENUM, bem como a técnica *poda extrema*. Na Secção 5 é descrita a abordagem de paralelização para cada um destes algoritmos. A Secção 6 mostra uma análise do desempenho e escalabilidade para cada uma das implementações. Por fim, a Secção 7 conclui o artigo.

2 Notação

A norma euclideana de um vector $\mathbf{v} \in \mathbb{R}^n$, denotada por $\|\mathbf{v}\|_2$ ou simplesmente $\|\mathbf{v}\|$, é a distância entre a origem do retículo e o ponto dado pelo vector $\mathbf{v}$, i.e. $\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} \mathbf{v}_i^2}$, onde $\mathbf{v}_i$ indica a i-nésima coordenada do vector $\mathbf{v}$.

O termo *vector nulo* é usado para identificar vectores cuja norma é zero, ou seja, a origem do retículo. Vectores e matrizes são escritos a negrito, aparecendo os vectores em minúsculas e as matrizes em maiúsculas, como por exemplo com o vector $\mathbf{v}$ e a matriz $\mathbf{M}$. Os seus elementos são, respectivamente, denotados por $\mathbf{v}_i$ e $\mathbf{M}_{i,j}$. As funções auxiliares usadas nos algoritmos são escritas como em `função`(). A transposta da matriz $\mathbf{M}$ é dada por $\mathbf{M}^T$. O valor absoluto de a é dado por $|a|$. Um retículo Λ, gerado por uma base $\mathbf{B}$, é denotado por $\Lambda(\mathbf{B})$.

Em computação paralela, a aceleração A_p e a eficiência E_p são definidas como

$$A_p = \frac{T_1}{T_p}, E_p = \frac{A_p}{p} = \frac{T_1}{pT_p}, \tag{2}$$

onde T_p indica o tempo de execução da aplicação usando p processadores.

3 Trabalho Relacionado

O PVP e o PVC foram extensivamente estudados durante as últimas décadas. Em particular, formaram-se duas principais famílias de algoritmos de PVC [9]. Uma das famílias é a dos algoritmos de crivo, algoritmos aleatorizados e probabilísticos, que geram vectores aleatórios e os vão transformando até um determinado critério de paragem ser verificado. A segunda família é a dos algoritmos

de enumeração, que enumeram vectores à volta da origem do retículo dentro de um determinado raio de procura, posteriormente escolhendo o mais curto entre eles. Os algoritmos de enumeração passaram a ser a abordagem mais eficiente para resolver o PVC quando, em 2010, foi publicada a técnica de poda extrema [4]. Esta técnica permite que se reduza substancialmente o número de vectores enumerados ao mesmo tempo que a probabilidade de encontrar o vector mais curto é reduzida, embora numa proporção substancialmente mais baixa.

Uma das maiores evidências da importância do PVC é o repositório SVP-challenge[3], onde equipas de investigação podem indicar os retículos para os quais conseguem resolver o PVC. Em particular, vários algoritmos de PVC têm sido implementados nas mais diversas arquitecturas computationais (e.g. [8]).

4 Os Algoritmos

Esta secção descreve detalhadamente os algoritmos de enumeração propostos por Ghasemmehdi e Agrell [5] e por Gama et al. [4], daqui em diante chamados de SE++ e ENUM. Estes algoritmos são versões melhoradas dos algoritmos propostos por Agrell et al. em [1] e por Schnorr e Euchner em [10], respectivamente. De seguida é também descrita a técnica de enumeração com poda extrema, proposta por Gama et al. [4], e como aplicá-la aos algoritmos mencionados anteriormente.

4.1 SE++

O SE++ foi proposto como um algoritmo de resolução de PVP, embora também tenha sido apresentada uma variante que resolve o PVC em [1]. Posteriormente, foi proposta em [5] uma variante deste algoritmo que evita operações redundantes para o PVP.

Antes da invocação do algoritmo, é necessário fazer um pré-processamento da base do retículo. Nesta fase, a matriz que contém os vectores da base, denotada por $\mathbf{B}$, é reduzida (por exemplo, com os algoritmos BKZ ou LLL). A matrix resultante é então transformada numa matrix triangular inferior, denominada $\mathbf{G}$, podendo ser usada a decomposição QR ou de Cholesky para o efeito (ver [1] para uma descrição mais detalhada). Esta transformação pode ser vista como uma mudança do sistema de coordenadas da base. Por fim, o algoritmo SE++ é alimentado com a dimensão n do retículo, e a inversa de $\mathbf{G}$, i.e., $\mathbf{H} = \mathbf{G}^{-1}$, a qual, por sua vez, também é triangular inferior.

O processo de descodificação da hiperesfera pode ser descrito de 2 formas distintas. Matematicamente, trata-se de um processo que enumera pontos do retículo dentro de uma hiperesfera, cujo raio diminui progressivamente, ao passo que, conceptualmente, este processo possa ser visto como uma travessia de uma árvore. Esta última descrição ajuda também a perceber melhor a abordagem proposta para a paralelização do algoritmo.

A descodificação da hiperesfera consiste em encontrar o vector mais curto do retículo, ou seja, o vector mais próximo da origem, que representa o centro da

[3] http://www.latticechallenge.org/svp-challenge/

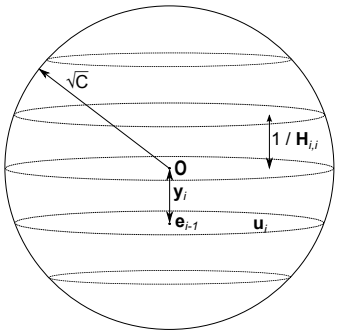


Figura 1. Representação da hiperesfera de dimensão i subdividida em camadas de dimensão $i-1$.

hiperesfera. $\sqrt{C}$ é a norma do vector mais curto encontrado até ao momento, e, portanto, representa o raio da mesma, tal como se pode observar na Figura 1.

A hiperesfera pode ser dividida em camadas de dimensões inferiores, onde a dimensão da camada que está a ser examinada a cada instante é denominada por i. Cada camada na dimensão i pode ser dividida em várias camadas paralelas na dimensão $i-1$, sendo a distância entre cada uma das camadas na dimensão $(i-1)$ dada por $1/H_{i,i}$.

O algoritmo itera sobre todas as camadas, começando na camada em dimensão n e parando quando voltar à mesma. O processo de visitar uma camada na dimensão inferior será denominado por *descida*, enquanto que o processo de visitar uma camada na dimensão superior será denominado por *subida*.

Para cada camada, é computada a sua projecção ortogonal $\mathbf{e}_i$ nos vectores da base, onde $\mathbf{e}_i = (\mathbf{E}_{i,1}, \mathbf{E}_{i,2}, ..., \mathbf{E}_{i,n})$, e o seu deslocamento ortogonal à camada actual $\mathbf{y}_i$. A camada na dimensão $i-1$ que está a ser examinada é determinada pelo coeficiente $\mathbf{u}_i$. Cada uma das camadas paralelas na dimensão $i-1$ é visitada em zigue-zague conforme o refinamento de Schnorr-Euchner [10]. O valor de u_i é actualizado com o auxílio de Δ_i, que contém o passo que é necessário tomar para visitar a próxima camada paralela. O quadrado da distância da camada actual à origem é dado por $\mathbf{dist}_i$, sendo este valor o responsável pela direcção que o algoritmo toma. Caso $\mathbf{dist}_i < C$, o algoritmo *desce*, caso contrário *sobe*.

Quando é atingida uma camada na dimensão *zero*, o vector mais curto nesse momento, i.e., $\mathbf{u}$, é guardado em $\hat{\mathbf{u}}$ e o valor de C é actualizado. O algoritmo pára quando regressa à camada na dimensão n e devolve o vector mais curto encontrado, $\hat{\mathbf{u}}$. Por fim, o vector encontrado tem de ser transformado para o sistema de coordenadas inicial, i.e., $\mathbf{x} = \hat{\mathbf{u}}\mathbf{B}$, sendo $\mathbf{x}$ o vector mais curto do retículo.

Conceptualmente, este algoritmo pode ser mapeado numa travessia em profundidade de uma árvore, cuja altura é $n+1$. A raiz da árvore é a origem, enquanto as folhas são todos os vectores cuja norma é menor do que C. Cada um dos nós da árvore representa uma camada da hiperesfera e os filhos desses

nós são camadas na dimensão $i-1$. O algoritmo percorre a árvore em busca do vector mais curto até voltar à raiz novamente. Sempre que é encontrada uma folha, o valor de C é actualizado, diminuindo o número de nós ainda por visitar.

Ghasemmehdi e Agrell [5] mostraram que existem várias operações redundantes na computação das projecções ortogonais no algoritmo original de Agrell et al. [1] e propuseram utilizar um vector auxiliar $\mathbf{d}$ para guardar as posições inciais das mesmas. Por exemplo, o valor $\mathbf{d}_i = k$ indica que para computar $\mathbf{E}_{i,i}$ é apenas necessário calcular os valores de $\mathbf{E}_{j,i}$, para $j = k-1, k-2, ..., i$.

4.2 ENUM

O ENUM é um algoritmo de enumeração originalmente proposto por Schnorr e Euchner [10] como sub-rotina do algoritmo BKZ, usado, por sua vez, na redução de bases. Gama et al. [4] propuseram uma forma de evitar operações redundantes, semelhante à de Ghasemmehdi e Agrell [5], para o ENUM. Nos dois primeiros artigos é possível encontrar uma descrição do algoritmo e como os modificar por forma a evitarem as referidas operações redundantes.

Tal como o SE++, o ENUM enumera todos os pontos dentro de uma hiperesfera, à procura do vector mais curto, percorrendo as camadas da hiperesfera igualmente em zigue-zague. Além disso, também este algoritmo pode ser mapeado numa travessia em profundidade de uma árvore. No entanto, existem algumas diferenças em relação ao SE++, no pré-processamento da base, no ponto de partida do algoritmo e na forma como algumas das camadas são percorridas.

O pré-processamento do ENUM inclui ainda a redução da base (e.g., com os algoritmos BKZ ou LLL) tal como o SE++, mas em vez de usar uma das decomposições QR ou de Cholesky, usa a ortogonalização de Gram-Schmidt. Esta ortogonalização gera uma matriz triangular inferior $\mathbf{I}$ (respectivamente μ em [10] e [4]) e um vector que contém os quadrados das normas da ortogonalização $\|\mathbf{b}_1^*\|^2, ..., \|\mathbf{b}_n^*\|^2$ de $\mathbf{B}$. Ambos são passados ao ENUM como argumentos.

O ponto de partida neste algoritmo é definido como sendo uma das folhas e não a raiz da árvore. Isto deve-se ao facto da primeira folha encontrada ser sempre o primeiro vector da base. Assim, é possível poupar as n primeiras iterações que correspondem a percorrer a árvore da raiz até à primeira folha. No entanto, o impacto desta optimização é reduzido dado que se trata de um número reduzido de iterações. Portanto, o valor de C é inicializado a $\|\mathbf{b}_1^*\|^2$ e a variável que contém o vector mais curto até ao momento é incializada a $\hat{\mathbf{u}} = (1, 0, ..., 0)$.

A diferença mais significativa entre os algoritmos é evitar percorrer sub-ramos simétricos da árvore através do uso de uma variável auxiliar *last_nonzero* (em [4]). Para cada vector $\hat{\mathbf{u}}$ que é encontrado na árvore, existe um ramo simétrico onde poderá ser encontrado $-\hat{\mathbf{u}}$. No entanto, como ambos têm a mesma norma e apenas é necessário encontrar um deles, a computação do ramo simétrico da árvore pode ser evitada. Portanto, o valor de *last_nonzero* indica o nível da árvore a partir do qual esta não necessita de ser percorrida em zigue-zague, evitando assim a computação do ramo simétrico da árvore.

4.3 SE++ e ENUM com Poda Extrema

Em 2010, Gama et al. [4] propuseram uma alteração ao ENUM que, apesar de o transformar numa heurística, consegue diminuir consideravelmente o número de nós que é necessário visitar e, consequentemente, o tempo de execução do algoritmo. No entanto, a probabilidade de encontrar o vector mais curto também é reduzida, embora num factor consideravelmente mais pequeno. Esta técnica é chamada de *poda extrema* e pode ser aplicada a algoritmos de enumeração.

Para descobrir quais os ramos da árvore que não necessitam de ser computados, define-se uma função polinomial com base na heurística gaussiana. Esta heurística fornece uma boa estimativa da norma do vector mais curto num retículo Λ e é dada por $FM(\Lambda) = \frac{\Gamma(n/2+1)^{1/n}}{\sqrt{\pi}} \times \det(\Lambda)^{1/n}$, onde $\Gamma(x)$ representa a função gamma, dada por $\Gamma(x) = (x-1)!$, e $\det(\Lambda)$ o determinante do retículo Λ. Para aumentar a probabilidade de encontrar o vector mais curto multiplica-se a estimativa da norma por um factor de 1.05, tal como descrito em [8]. A função delimitadora irá usar esta estimativa para calcular o valor máximo para o quadrado da distância de um nó da árvore à raiz para cada um dos níveis da mesma. Cada um destes valores é computado com base numa função polinomial.

Tal como referido anteriormente, cada uma das chamadas ao algoritmo de enumeração com poda extrema possui uma probabilidade de apenas 10% de encontrar o vector mais curto, de acordo com os testes feitos em [8]. No entanto, é possível aumentar esta probabilidade ao efectuar várias chamadas ao algoritmo em bases diferentes mas que geram o mesmo retículo. Assim, para cada uma destas chamadas, é necessário efectuar dois passos prévios: (1) aleatorizar a base e (2) efectuar o pré-processamento correspondente. O primeiro passo consiste em gerar uma base diferente para o mesmo retículo. O pré-processamento consiste em reduzir a base aleatorizada (quanto melhor a redução maior a probabilidade de encontrar o vector mais curto) e efectuar a decomposição QR ou de Cholesky, no caso do SE++, ou a ortogonalização de Gram-Schmidt, no caso do ENUM.

Kuo et al. [8] também verifiram que 44 chamadas ao algoritmo de enumeração com poda extrema garantem uma probabilidade de sucesso acima de 99%. Portanto, será este o valor usado nos resultados apresentados na Secção 6.2.

Em [4] é demonstrado como transformar o ENUM num algoritmo de enumeração com poda extrema. É possível aplicar as mesmas alterações ao SE++ nos passos correspondentes.

5 Paralelização

5.1 SE++

Como referido anteriormente, ambos os algoritmos SE++ e ENUM podem ser mapeados numa travessia de uma árvore. Cada ramo pode ser computado em paralelo, sendo apenas necessária sincronização para actualizar o melhor vector e o quadrado da sua respectiva norma. A implementação foi escrita em C e cria tarefas, distribuídas por um conjunto de *threads*, com o OpenMP. Para cada um dos ramos que irá computar em paralelo é criada uma tarefa, que é adicionada

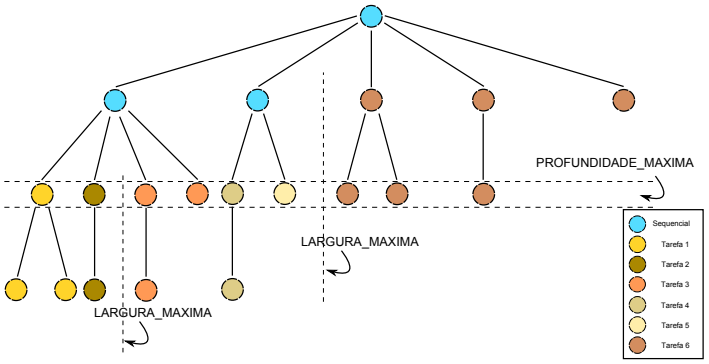


Figura 2. Mapeamento da execução do algoritmo numa árvore. Exemplificação dos parâmetros LARGURA_MAXIMA= 3 e PROFUNDIDADE_MAXIMA= $n-2$ (4 *threads*).

a uma fila de espera e escalonada pelo OpenMP assim que haja *threads* à espera de trabalho. A sua ordem de execução é também ela definida pelo sistema.

A nossa implementação paralela consiste em criar várias tarefas através de uma travessia em largura, ao passo que cada uma das tarefas é executada em profundidade. Dado que a árvore é bastante desbalanceada (como se pode observar na Figura 2), são usados dois parâmetros para garantir um equilíbrio entre o balanceamento de carga computacional e a granularidade de cada uma das tarefas: PROFUNDIDADE_MAXIMA e LARGURA_MAXIMA. Quando é atingida a PROFUNDIDADE_MAXIMA é criada uma tarefa para cada um dos nós nesse nível. Este parâmetro define o nível da árvore a partir do qual são criadas tarefas, permitindo criar tarefas mais leves para melhorar o balanceamento de carga. No entanto, se for atingida a LARGURA_MAXIMA, a *thread* que irá executar essa tarefa não irá executar apenas a sub-árvore respectiva, mas também todos os irmãos dessa sub-árvore (excluindo os irmãos para os quais já foram criadas tarefas). Este parâmetro também se aplica quando a PROFUNDIDADE_MAXIMA ainda não foi atingida, ou seja, podem ser criadas tarefas em qualquer nível da árvore caso o valor de um dos parâmetros seja atingido. A LARGURA_MAXIMA é atingida assim que $|\Delta_i|$ = LARGURA_MAXIMA. Estes parâmetros foram definidos como PROFUNDIDADE_MAXIMA $= n - \log_2($ $\#Threads)$, onde $\#Threads$ denota o número de *threads* que foram criadas pelo OpenMP, e como LARGURA_MAXIMA $= 6$, baseado em testes empíricos com retículos nas dimensões 40, 50 e 60. O funcionamento destes parâmetros pode ser observado na Figura 2.

Como forma de optimizar o uso de memória alocada, cada *thread* aloca uma estrutura com as variáveis que podem ser reutilizadas (a matriz $\mathbf{E}$ e os vectores $\mathbf{u}$, $\mathbf{y}$, $\mathbf{dist}$, $\mathbf{\Delta}$ e $\mathbf{d}$ em [5]). Assim, não é necessário alocar e libertar estas variáveis de cada vez que uma tarefa é executada, mas sim apenas incializá-las no início da execução da mesma.

O valor C e o vector $\hat{\mathbf{u}}$ são guardados em variáveis globais, partilhadas por todas as *threads*. Isto permite que as demais *threads* possam actualizar estes valores concorrentemente assim que encontrem um vector melhor que o anterior, o que pode eventualmente reduzir a quantidade de trabalho de outras *threads*. Uma zona crítica do OpenMP é utilizada como mecanismo de sincronização para a actualização destas variáveis. Para impedir a criação de tarefas de sub-árvores que numa execução sequencial nunca iriam ser executadas, C é inicializado a $1/\mathbf{H}_{1,1}$ e $\hat{\mathbf{u}}$ a $(1, 0, ...0)$.

5.2 ENUM

Esta implementação, tal como a anterior, também foi escrita em C e também usa OpenMP para a criação de tarefas. Os parâmetros definidos anteriormente também são usados nesta abordagem, com os mesmos valores e, portanto, a Figura 2 também é válida para o ENUM. No entanto, foi necessário alterar o fluxo de execução para que a mesma abordagem também pudesse ter sido aplicada a este algoritmo. Assim, em vez do algoritmo começar a execução a partir de uma das folhas, começa a partir da raiz, tal como acontece com o SE++.

Outra semelhança entre as implementações é o uso de uma estrutura auxiliar para guardar as variáveis afectas a cada *thread*, i.e., a matriz σ e os vectores υ, $\mathbf{c}$, ρ, ω e $\mathbf{r}$ (em [4]). Tal como no SE++, esta implementação também guarda globalmente o valor de C e o vector $\hat{\mathbf{u}}$, sendo também actualizados dentro de uma zona crítica do OpenMP. Estas duas variáveis são incializadas a $C = \|\mathbf{b}_1^*\|^2$ e a $\hat{\mathbf{u}} = (1, 0, ..., 0)$.

5.3 SE++ e ENUM com Poda Extrema

Ambos os algoritmos com poda extrema foram implementados em C, com o padrão MPI para a distribuição de trabalho pelos vários processos. O OpenMP não foi usado pois a biblioteca NTL[4], usada para a redução das bases dos retículos e, no caso do ENUM, também para a computação da ortogonalização de Gram-Schmidt, não oferece suporte para execução concorrente em memória partilhada.

Tal como visto anteriormente, é necessário efectuar várias chamadas ao algoritmo de enumeração com poda extrema (tanto no caso do SE++ como no do ENUM), para garantir uma elevada probabilidade de sucesso. Cada chamada requer uma aleatorização da base e um pré-processamento da base aleatorizada. Cada uma destas iterações (aleatorização, pré-processamento e chamada do algoritmo de enumeração com poda extrema) é independente das restantes, sendo possível distribuí-las pelos processos sem necessidade de sincronização.

Dado que o tempo de execução da enumeração depende da qualidade da base, as várias iterações demoram tempos diferentes. Assim, usou-se um processo adicional apenas para distribuir as iterações pelos processos e recolher os melhores vectores encontrados no final da execução de todas as iterações. Isto garante uma melhor distribuição da carga computacional, já que cada processo pode solicitar

[4] http://www.shoup.net/ntl/

mais iterações assim que conclua a anterior. Para isso, cada processo apenas necessita de solicitar o número da iteração que irá computar. Este número é usado como semente aleatória para aleatorização da base.

6 Resultados

Os testes foram conduzidos num computador com 2 processadores Sandy Bridge Intel Xeon E5-2670 (dois *sockets*), cada um com 8 núcleos, equipados com *simultaneous multi-threading* (SMT). O computador tem um total de 128 GB de memória de acesso aleatório (RAM). Os códigos foram compilados apenas com `GNU g++ 4.6.1`, excepto os códigos paralelos de SE++ e ENUM com poda extrema, onde foi ainda usado o `OpenMPI 1.4.3`. Os códigos foram compilados com a flag -O2 (-O3 revelou ser um pouco mais lenta que -O2). Adicionalmente, foram utilizadas as bibliotecas NTL (para a redução da base do retículo, em ambos algoritmos, e para a ortogonalização de Gram-Schmidt no algoritmo ENUM) e Eigen[5] (para a computação da decomposição QR, inversa e transposta de matrizes no algoritmo SE++). Foram usadas bases de Goldstein-Mayer para retículos aleatórios, disponíveis no sítio SVP-challenge, todas geradas com a semente 0. Cada código foi executado três vezes e foi escolhido o melhor resultado.

6.1 SE++ e ENUM

Esta secção compara o desempenho dos algoritmos SE++ e ENUM, tal como descritos nas Secções 4.1 e 4.2, e das suas versões paralelas, descritas nas Secções 5.1 e 5.2. O tempo de execução do pré-processamento da base não foi incluído nos tempos de execução apresentados, visto que se trata de um valor residual em comparação com os processos de descodificação da hiperesfera. As implementações foram testadas em retículos reduzidos pelo algoritmo BKZ (com tamanho de bloco $\beta = 20$) nas dimensões 50, 52, 54, 56, 58 e 60.

A Figura 3 mostra os tempos de execução das versões sequenciais dos algoritmos SE++ e ENUM. O ENUM corre mais rápido em todas as instâncias, o que se deve principalmente ao facto deste algoritmo não visitar ramos simétricos, ao contrário do SE++. Outras razões incluem operações pesadas que são efectuadas no algoritmo SE++, como é o caso da divisão que é efectuada no cálculo de $\mathbf{y}_i$.

O tempo de execução aumenta com a dimensão do retículo. No entanto, isto não acontece no retículo na dimensão 58 visto que a sua base é de qualidade reduzida e, como já referido anteriormente, o desempenho da enumeração (tanto SE++ como ENUM) depende fortemente da qualidade da base.

As versões paralelas foram testadas para os mesmos retículos, usando 1 a 32 *threads*. Como mostrado nas Figuras 4(a) e 4(b), as implementações escalam linearmente até 8 *threads* e quase linearmente com 16 *threads*. As implementações também beneficiam de SMT, dado que existem dependências entre instruções, o que impede o uso das unidades funcionais em cada núcleo na sua totalidade.

[5] http://eigen.tuxfamily.org/

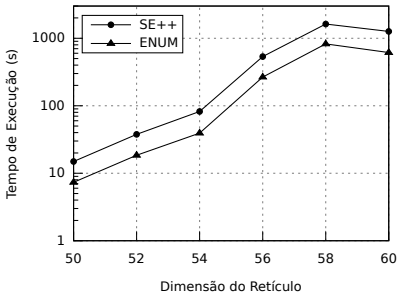
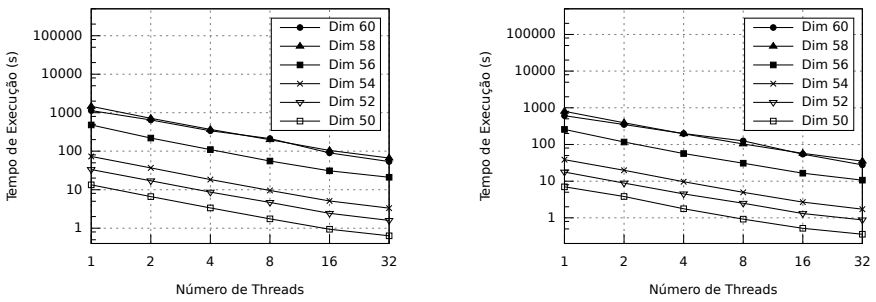


Figura 3. Desempenho das versões sequenciais dos algoritmos SE++ e ENUM para retículos aleatórios entre as dimensões 50 e 60. As bases foram reduzidas com o algoritmo BKZ com $\beta = 20$.

As Tabelas 1 e 2 mostram a aceleração e a eficiência para os vários números de *threads*, respectivamente para os algoritmos SE++ e ENUM. A última linha encontra-se destacada porque concerne um caso especial (o uso de SMT). As versões paralelas podem ter uma quantidade de trabalho menor do que a versão sequencial, pois algumas *threads* podem encontrar, a qualquer momento, um vector mais curto do que aquele que estaria a ser analisado numa execução sequencial. Isto justifica a ocorrência de acelerações super-lineares em alguns casos.

Para a maior parte dos restantes casos, a eficiência toma valores superiores a 90% até 8 *threads*. A eficiência diminui para 16 *threads* presumivelmente devido ao uso de dois *sockets*, que é menos eficiente do que o uso de apenas um, devido à organização por acessos não-uniformes à RAM. No entanto, o retículo de dimensão 56 foi um caso extraordinário, pois apesar do uso de dois *sockets*, consegue atingir acelerações lineares para 16 *threads*. Isto acontece presumivelmente devido ao facto do *overhead* que advém do uso de dois *sockets* ser compensado



(a) SE++ (b) ENUM

Figura 4. Desempenho das versões paralelas dos algoritmos SE++ e ENUM para retículos aleatórios entre as dimensões 50 e 60. As bases foram reduzidas com o algoritmo BKZ com $\beta = 20$.

	Dimensão do Retículo											
	50		52		54		56		58		60	
Threads	A	E	A	E	A	E	A	E	A	E	A	E
2	2.02x	101%	1.97x	98%	1.99x	100%	2.19x	109%	2.05x	103%	1.74x	87%
4	3.97x	99%	3.86x	97%	4.00x	100%	4.36x	109%	3.98x	99%	3.32x	83%
8	7.64x	96%	7.18x	90%	7.65x	96%	8.61x	108%	7.35x	92%	5.30x	66%
16	14.17x	89%	13.79x	86%	14.36x	90%	15.53x	97%	14.01x	88%	12.53x	78%
32	21.04x	66%	21.05x	66%	22.01x	69%	22.77x	71%	22.06x	69%	21.01x	66%

Tabela 1. Aceleração (A) e eficiência (E) para o algoritmo SE++ em 6 retículos aleatórios, cujas bases foram reduzidas pelo algoritmo BKZ com $\beta = 20$.

	Dimensão do Retículo											
	50		52		54		56		58		60	
Threads	A	E	A	E	A	E	A	E	A	E	A	E
2	1.84x	92%	1.99x	99%	1.94x	97%	2.20x	110%	2.05x	102%	1.74x	87%
4	3.99x	100%	3.93x	98%	3.97x	99%	4.56x	114%	4.11x	103%	3.07x	77%
8	7.65x	96%	7.05x	88%	7.72x	97%	8.39x	105%	7.75x	97%	4.84x	60%
16	13.52x	85%	13.47x	84%	14.29x	89%	15.62x	98%	13.93x	87%	11.17x	70%
32	19.70x	62%	20.11x	63%	22.24x	70%	24.20x	76%	22.72x	71%	21.83x	68%

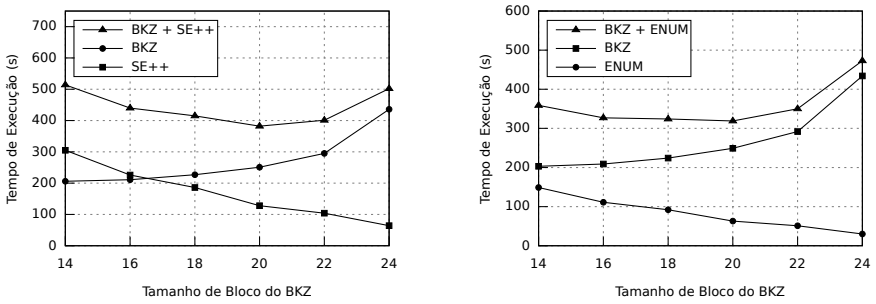
Tabela 2. Aceleração (A) e eficiência (E) para o algoritmo ENUM em 6 retículos aleatórios, cujas bases foram reduzidas pelo algoritmo BKZ com $\beta = 20$.

pela poupança de trabalho relativamente à execução sequencial. Para o retículo de dimensão 60, a eficiência é menor do que em relação aos restantes casos, provavelmente devido a um pior aproveitamento da localidade de *cache* e, consequentemente, um aumento dos acessos à RAM. Isto pode ser justificado pelos valores obtidos com 32 *threads*, onde as acelerações se mantiveram, em comparação com os restantes casos, sendo o uso de SMT responsável por esconder a latência destes acessos.

6.2 SE++ e ENUM com poda extrema

Nesta secção, são comparados os desempenhos de ambos os algoritmos com poda extrema, tal como descritos na Secção 4.3, e das suas versões paralelas, descritas na Secção 5.3. Neste caso, o tempo de pré-processamento também é incluído nos tempos de execução apresentados, uma vez que o tempo de redução da base representa uma parte considerável do total do tempo de execução. As implementações foram testadas em retículos reduzidos pelo algoritmo BKZ nas dimensões 70, 72, 74, 76, 78 e 80.

As versões paralelas de ambas as implementações foram testadas para 1 a 32 processos que efectivamente executam trabalho e um processo adicional que apenas trata do envio de trabalho e recolha dos melhores vectores. Dado que o tempo de execução deste último é desprezável em função do tempo total de execução,



(a) SE++ (b) ENUM

Figura 5. Soma dos tempos de execução das várias iterações das implementações paralelas do SE++ e ENUM com poda extrema. Resultados para 32 processos e para um retículo de dimensão 80, cuja base foi reduzida com o algoritmo BKZ com diferentes valores de β.

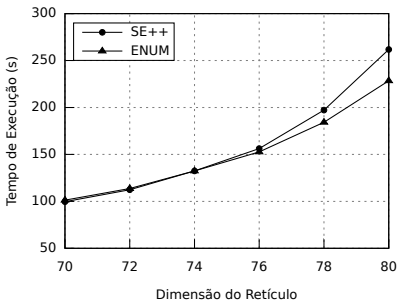
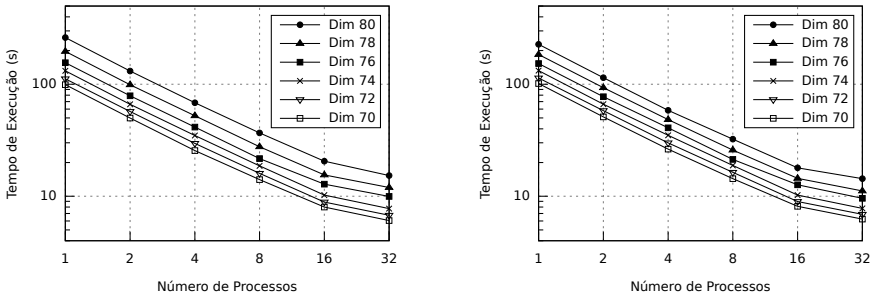


Figura 6. Desempenho das versões sequenciais dos algoritmos SE++ e ENUM com poda extrema para retículos aleatórios entre as dimensões 70 e 80. As bases foram reduzidas pelo algoritmo BKZ com $\beta = 20$.

foi omitido nos resultados mencionados, ou seja, o número de processos mencionados refere-se apenas aos processos que efectivamente executam trabalho.

Dado que a redução da base apresenta tempos de execução significativos em relação ao tempo total de execução e, além disso, influencia largamente o tempo de execução da enumeração com poda extrema (tanto SE++ como ENUM), foram efectuados testes para descobrir o valor óptimo para β. Em particular, foi avaliado o retículo na dimensão 80, por se tratar do retículo com maior dimensão entre os retículos testados. Como se pode observar nas Figuras 5(a) e 5(b), o tempo de execução do BKZ aumenta com o tamanho de bloco, muito embora o tempo de execução dos algoritmos de enumeração, com poda extrema, diminuia. As curvas, que dizem respeito ao tempo de execução do BKZ e do algoritmo de enumeração (tanto SE++ quanto ENUM) conjugados, têm a forma de uma parábola, com mínimo no tamanho de bloco 20. Portanto, daqui em diante, é usado $\beta = 20$ como tamanho de bloco do BKZ para os restantes testes.



(a) SE++ (b) ENUM

Figura 7. Desempenho das versões paralelas dos algoritmos SE++ e ENUM com poda extrema para retículos aleatórios entre as dimensões 70 e 80. As bases foram reduzidas com o algoritmo BKZ com $\beta = 20$.

	Dimensão do Retículo											
	70		72		74		76		78		80	
Processos	A	E	A	E	A	E	A	E	A	E	A	E
2	1.99.x	99%	1.96x	98%	1.99x	100%	1.98x	99%	1.99x	99%	1.99x	99%
4	3.89x	97%	3.78x	95%	3.79x	95%	3.76x	94%	3.76x	94%	3.82x	96%
8	7.06x	88%	7.00x	87%	7.07x	88%	7.19x	90%	7.12x	89%	7.11x	89%
16	12.42x	78%	12.63x	79%	12.89x	81%	12.18x	76%	12.70x	79%	12.73x	80%
32	16.39x	51%	16.57x	52%	17.02x	53%	15.66x	49%	16.46x	51%	17.09x	53%

Tabela 3. Aceleração (A) e eficiência (E) para o algoritmo SE++ com poda extrema em 6 retículos aleatórios, cujas bases foram reduzidas pelo algoritmo BKZ com $\beta = 20$.

Na Figura 6, é possível observar os tempos de execução das versões sequenciais da enumeração com poda extrema. Para retículos nas dimensões mais reduzidas, os tempos de execução são quase idênticos, pois a redução da base toma uma parte substancial do tempo total de execução. Quando a enumeração com poda extrema começa a ter um impacto significativo no tempo de execução pode observar-se que, tal como na enumeração sem poda extrema, o ENUM é mais rápido que o SE++. Além disso, também é possível verificar um aumento no tempo de execução à medida que aumenta a dimensão do retículo.

Como se pode observar nas Figuras 7(a) e 7(b), as implementações escalam linearmente até 4 processos. Para 8 e 16 processos, a escalabilidade diminui devido ao desbalanceamento de carga. Este desbalanceamento deve-se ao facto do algoritmo de redução BKZ usar como sub-rotina um algoritmo de enumeração e da poda extrema ser uma técnica que é aplicada a algoritmos de enumeração, cujos tempos de execução são substancialmente influenciados pela qualidade da base do retículo. Apesar de se tratar do mesmo retículo nas várias iterações, ao aleatorizar a base, é criada uma base diferente para o mesmo retículo. Portanto, os tempos de execução da sua redução e da poda extrema também variam em

	Dimensão do Retículo											
	70		72		74		76		78		80	
Processos	A	E	A	E	A	E	A	E	A	E	A	E
2	1.99x	99%	1.96x	98%	2.00x	100%	1.98x	99%	1.98x	99%	1.99x	99%
4	3.85x	96%	3.81x	95%	3.78x	95%	3.75x	94%	3.81x	95%	3.89x	97%
8	7.07x	88%	6.97x	87%	7.05x	88%	7.18x	90%	7.15x	89%	7.03x	88%
16	12.50x	78%	12.72x	79%	12.96x	81%	12.10x	76%	12.78x	80%	12.71x	79%
32	16.27x	51%	16.54x	52%	17.04x	53%	15.98x	50%	16.56x	52%	15.88x	50%

Tabela 4. Aceleração (A) e eficiência (E) para o algoritmo ENUM com poda extrema em 6 retículos aleatórios, cujas bases foram reduzidas pelo algoritmo BKZ com $\beta = 20$.

função da nova base. É possível verificar que estas implementações também beneficiam de SMT, o que se deve essencialmente a dependências entre instruções que impedem o uso total das unidades funcionais em cada núcleo.

As Tabelas 3 e 4 mostram a aceleração e a eficiência para as nossas implementações. A eficiência encontra-se acima dos 90% até 4 processos e próxima de 90% para 8 processos. A escalabilidade diminui para um maior número de processos, devido ao aumento do desbalanceamento como já referido anteriormente.

7 Conclusões

Este artigo apresenta o primeiro estudo comparativo dos algoritmos de enumeração para resolução do PVC propostos em [5] e [4], respectivamente denominados SE++ e ENUM. Para esse fim, são descritas abordagens para a paralelização dos algoritmos, (1) em sistemas multiprocessamento com memória partilhada, sem poda extrema e (2) em sistemas multiprocessamento com memória distribuída, com poda extrema. A poda extrema é uma importante técnica para transformar algoritmos de enumeração em heurísticas eficientes para o PVC. As nossas abordagens de paralelização, quer para o caso com poda extrema, quer para o caso sem a mesma, são aplicáveis a ambos os algoritmos, visto que os mesmos têm um fluxo computacional muito semelhante. A principal característica da nossa abordagem para o caso sem poda extrema é o uso de dois parâmetros que influenciam o número e o tamanho das tarefas geradas, por forma a potenciar um balanço de carga computacional óptimo entre as demais *threads*.

No caso da enumeração sem poda extrema, foram atingidas acelerações lineares até 8 *threads* e quase lineares para 16 *threads* em ambos os algoritmos. Em algumas instâncias são atingidas acelerações super-lineares devido à redução da carga de trabalho em relação à versão sequencial, uma consequência directa da execução do algoritmo em paralelo. A abordagem apresentada para a paralelização dos algoritmos com poda extrema pode, também ela, ser aplicada a ambos os algoritmos sem qualquer necessidade de modificações. A escalabilidade destas implementações é, no entanto, inferior às versões sem poda extrema, devido ao desbalanceamento de carga computacional. Este é um problema inerente

ao uso de poda extrema, porque (1) o número de chamadas ao algoritmo é fixado em 44, independentemente do número de processos usados e (2) é impraticável prever o tempo de execução de cada chamada. Ainda assim, foram atingidas acelerações de quase 13 vezes para 16 processos.

Deste modo, é possível concluir que a escalabilidade de algoritmos de enumeração sem poda extrema possa ser linear, embora a escalabilidade dos mesmos algoritmos, mas com poda extrema, seja ligeiramente reduzida, em sistemas com pelo menos 32 núcleos computacionais. Sequencialmente, o ENUM é mais eficiente que o SE++.

Como trabalho futuro, iremos avaliar a possibilidade do SE++ evitar a análise de ramos simétricos na árvore, como acontece no ENUM. Todavia, é de crer que esta optimização seja apenas aplicável na resolução do PVC.

Agradecimentos: Este trabalho é financiado por Fundos Nacionais através da FCT - Fundação para a Ciência e a Tecnologia no âmbito do projeto PEst-OE/EEI/UI0752/2014

Referências

1. Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest Point Search in Lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.
2. M. Ajtai. Generating Hard Instances of Lattice Problems (Extended Abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM.
3. Miklós Ajtai and Cynthia Dwork. A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(65), 1996.
4. Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice Enumeration Using Extreme Pruning. In *EUROCRYPT*, pages 257–278, 2010.
5. Arash Ghasemmehdi and Erik Agrell. Faster Recursions in Sphere Decoding. *IEEE Transactions on Information Theory*, 57(6):3530–3536, 2011.
6. Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In Jr. Kaliski, BurtonS., editor, *Advances in Cryptology — CRYPTO '97*, volume 1294 of *Lecture Notes in Computer Science*, pages 112–131. Springer Berlin Heidelberg, 1997.
7. Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In *Lecture Notes in Computer Science*, pages 267–288. Springer-Verlag, 1998.
8. Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme Enumeration on GPU and in Clouds: How Many Dollars You Need to Break SVP Challenges. CHES'11, pages 176–191, Berlin, Heidelberg, 2011. Springer-Verlag.
9. Thijs Laarhoven, Joop van de Pol, and Benne de Weger. Solving Hard Lattice Problems and the Security of Lattice-Based Cryptosystems. Cryptology ePrint Archive, Report 2012/533, 2012.
10. Claus-Peter Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Math. Program.*, 66:181–199, 1994.