Uma rede com topologia *folded clos* é definida por duas camadas distintas, formando um grafo bipartido tal que cada nó da primeira camada está ligado a todos os nós da segunda (ver Figura 2d). Os caminhos interessantes entre cada par de nós da primeira camada (que corresponde ao conjunto N) são todos simultaneamente de menor custo e disjuntos entre si e o seu número é igual ao número de nós da segunda camada, dado por n. Para cobrir todos esses caminhos, $k = n$, $h = 0$ e $f = 1$. Na rede usada nos testes, $n = 6$. As redes *folded clos* também são utilizadas em centros de dados e têm a propriedade de assegurar a disjunção máxima de caminhos. É interessante referir que, com a *fat tree* e a *folded clos*, é executado o primeiro caso do algoritmo.

A aplicação do algoritmo a estas quatro redes permitiu verificar que este seleccionou exactamente os caminhos esperados. A execução do algoritmo para cada rede (*i.e.*, para todos os pares $(x,y) \in N^2$ tais que $x < y$) não excedeu 140 milissegundos.

O algoritmo do Spain apresenta resultados semelhantes, excepto com a rede *fat tree* quando os caminhos de x para y têm de subir dois níveis. Nesses casos, $k = 8$ (porque há 8 caminhos óptimos), mas o conjunto dos caminhos seleccionados tem cardinalidade quatro. Este comportamento deve-se ao critério de paragem do algoritmo. O Spain aplica o algoritmo de Dijkstra para descobrir um caminho de custo mínimo de x para y. Depois, aumenta significativamente os pesos dos arcos desse caminho (adicionando-lhes o número de arcos do grafo, quando todos os pesos iniciais são 1). Estes dois passos são repetidos até terem sido descobertos k caminhos de x para y ou até o caminho obtido ser igual a um dos previamente descobertos. Ora, nos casos que estamos a analisar, os quatro primeiros caminhos encontrados têm a forma: $x\,v_1\,a\,w_1\,y$, $x\,v_2\,a\,w_2\,y$, $x\,v_1\,b\,w_1\,y$ e $x\,v_2\,b\,w_2\,y$, onde v_1 e v_2 representam os nós adjacentes a x, w_1 e w_2 representam os nós adjacentes a y, e a e b denotam os nós do nível de cima. Nesse momento, os arcos incidentes em x ou y têm todos peso 49 $(1+24+24)$ e os incidentes em a ou b têm peso 25. O problema é que o custo actual de qualquer um dos 8 caminhos óptimos é o mesmo e, como o algoritmo de Dijkstra é determinista, o quinto caminho obtido é igual ao primeiro descoberto. Os próprios autores reconhecem esta limitação ([6], secção 8.3).

4.2 Redes Não Regulares

Foram usados três *backbones* reais e uma rede aleatória. Para os testes foram removidos das redes reais canais paralelos e nós que não acrescentavam diversidade. Nas redes reais o custo dos canais usado é proporcional à distância entre os nós e na rede aleatória o peso de todos os arcos é 1. A rede GÉANT é uma rede europeia de investigação e educação que interliga as diferentes redes de investigação e educação dos países europeus. A configuração testada tem 31 nós e 50 canais. A Internet 2 é um consórcio norte-americano que liga universidades, centros de investigação, corporações e outras redes regionais de educação e investigação nos Estados Unidos da América. A configuração testada tem 25 nós e 44 canais. A *NTT Communications* é uma corporação subsidiária da *NTT* (*Nippon Telegraph and Telephone*) e possui uma rede IP com uma distribuição geográfica mundial. A configuração testada tem 27 nós e 63 canais. A rede aleatória utilizada foi gerada com base numa distribuição de *Poisson*, com um grau médio dos nós de valor 3. A configuração testada tem 30 nós e 48 arcos. Nos quatro casos, todos os nós são origem e destino de tráfego.

Tabela 1: Resultados da execução do algoritmo nas redes não regulares (com $k = 3$, $h = 5$ e $f = 5$ ($f = 6$, na rede aleatória))

Redes	% de pares em que foram seleccionados			% de pares que suportam			% cobertura
	1 caminho	2 caminhos	3 caminhos	0 falhas	1 falha	2 falhas	de menor custo
GÉANT	1.29	1.72	96.99	1.29	75.7	23.01	100
Internet 2	0.33	0.33	99.33	0.33	98.67	1.0	100
NTT	0.85	1.71	97.44	0.85	68.95	30.2	100
Aleatória	0.0	0.23	99.77	2.99	60.23	36.78	90.64

Tabela 2: Resultados da execução do Spain nas redes não regulares (com k = 3)

Redes	% de pares em que foram seleccionados			% de pares que suportam			% cobertura
	1 caminho	2 caminhos	3 caminhos	0 falhas	1 falha	2 falhas	de menor custo
GÉANT	0.0	12.69	87.31	0.0	74.19	25.81	95.71
Internet 2	0.0	31.33	68.67	0.0	88.0	12.0	100
NTT	0.0	10.83	89.17	0.0	65.81	34.19	95.45
Aleatória	0.0	4.83	95.17	0.0	51.72	48.28	75.13

Com estas redes, o nosso algoritmo foi executado com os parâmetros $k = 3$, $h = 5$ e $f = 5$ e o do Spain com $k = 3$. Na rede aleatória, devido a todos os arcos terem peso 1, f tomou o valor $h+1$, ou seja, 6, tal como aconteceu nas redes regulares. Os resultados do nosso algoritmo encontram-se na Tabela 1 e os do Spain na Tabela 2. As tabelas apresentam a distribuição do número de caminhos seleccionados e a distribuição da tolerância a falhas, ambas em função dos pares de nós. O número de falhas suportadas é o número de caminhos disjuntos seleccionados menos uma unidade. A percentagem da cobertura de menor custo é (100 vezes) o quociente entre a soma do número de caminhos de custo mínimo seleccionados e a soma do número de caminhos de custo mínimo existentes. Esse valor é 100 se, para todos os pares de nós, o conjunto dos caminhos seleccionados contém todos os caminhos de menor custo.

Os resultados do nosso algoritmo mostram que este selecciona todos os caminhos de menor custo que existem, com excepção da rede aleatória, onde existem pares com mais de 3 caminhos de custo mínimo. Para que todos fossem seleccionados, seria necessário aumentar o valor de k.

Ao nível da tolerância a falhas, os resultados mostram que nem todos os pares de nós suportam pelo menos uma falha, existindo uma pequena fracção que não suporta nenhuma. Essa fracção, nas redes reais, coincide com a fracção de pares para os quais apenas é seleccionado um caminho. Estes são casos excepcionais, bem identificados, que acontecem quando só há um caminho de menor custo que tem comprimento ou custo muito baixo e os caminhos alternativos, por oposição, têm comprimentos ou custos muito elevados (não sendo portanto considerados próximos dos óptimos). Aumen-

tando o valor de h ou de f (e de k no caso da rede aleatória) para estes pares, consegue-se eliminar estas excepções e garantir que todos os pares toleram pelo menos uma falha.

Comparativamente, o algoritmo do Spain privilegia a tolerância a falhas face à distribuição de carga. Selecciona um conjunto de caminhos que garante a tolerância a pelo menos uma falha, embora não seleccione todos os melhores caminhos (ou seja, os de custo mínimo). Por outro lado, e tal como já tinha acontecido com as redes regulares, o algoritmo selecciona menos caminhos (como se observa comparando as primeiras três colunas das tabelas). Infelizmente, quando o algoritmo pára a geração de caminhos prematuramente, aumentar o valor de k não altera o resultado.

Com o algoritmo do Spain, os resultados apresentados para cada uma das redes foram calculados em menos de 350 milissegundos. Com o nosso algoritmo, os resultados foram obtidos em menos de 120 segundos, excepto para a NTT. Neste caso a execução completa demorou cerca de 4 horas. Há aproximadamente 35 pares com uma, duas ou três dezenas de milhares de caminhos interessantes, para cada um dos quais o tempo médio de execução foi de cerca de 5 minutos. Curiosamente, o maior conjunto de caminhos interessantes (com 29416 elementos) foi processado em menos de um segundo.

Em resumo, o nosso algoritmo, com os parâmetros adequados, selecciona os caminhos interessantes, isto é, tenta satisfazer os dois objectivos: tolerância a falhas e distribuição de carga. Adicionalmente, o algoritmo permite facilmente classificar a qualidade dos caminhos seleccionados para cada par (porque compara o valor de k com a cardinalidade do conjunto dos caminhos de custo mínimo e com a cardinalidade do conjunto dos caminhos interessantes e sabe calcular a disjunção do conjunto retornado). Numa segunda fase, isto permite a evolução do algoritmo para um mais inteligente que, ao detectar conjuntos anómalos ou deficientes, por exemplo, sem nenhuma tolerância a falhas, adaptaria os parâmetros para esses casos. Os valores de k, h e f funcionariam então como valores mínimos, não limitando a qualidade do conjunto de caminhos seleccionados. Uma segunda alternativa para resolver a intolerância a uma falha (se se detectasse que a disjunção do conjunto computado era um), seria executar o algoritmo de Dijkstra no grafo sem os arcos de um dos caminhos óptimos, acrescentando-se o caminho obtido ao conjunto retornado.

5 Conclusões e Trabalho Futuro

Neste artigo foi apresentado um algoritmo para escolha de caminhos entre pares de nós de um grafo que modeliza uma rede de computadores. O problema surge no contexto do encaminhamento multi-caminho com escolha a priori dos caminhos pelo quais o tráfego injectado na rede deve ser distribuído.

O algoritmo desenvolvido, avaliado e testado através de diversas redes sintéticas bem conhecidas, e outras reais, revela-se mais abrangente e flexível que outros algoritmos referenciados na literatura para atacar o mesmo problema. Com efeito, trata-se do primeiro algoritmo que tenta sistematicamente acomodar vários tipos de critérios simultaneamente, nomeadamente: não permitir a explosão do número de caminhos seleccionados pois impõe um limite ao número de caminhos distintos seleccionados; selecção apenas de caminhos cuja diferença de custo ou comprimento em relação aos caminhos óptimos seja limitada e portanto mais realistas; suporte de tolerância a falhas; e suporte

de distribuição de carga e da exploração da capacidade disponível do ponto de vista de cada par de nós.

Adicionalmente, o algoritmo calcula igualmente de que forma o critério tolerância a falhas é suportado no que diz respeito às falhas de canais. Finalmente, o algoritmo tem um tempo de execução significativo sobretudo quando o espaço de exploração de hipóteses cresce devido aos parâmetros utilizados, mas este aspecto não é preocupante pois o cálculo de caminhos para cada par de nós pode ser executado em paralelo.

Este trabalho faz parte de um trabalho mais geral e tem várias etapas seguintes: o estudo de técnicas e algoritmos de agregação de caminhos que permitam diminuir a complexidade espacial (do *control plane*) do encaminhamento na rede concreta, isto é, diminuição do número de identificadores necessários para suportar o encaminhamento multi-caminho; avaliação da adequação dos caminhos seleccionados ao suporte global de engenharia de tráfego, isto é, ao suporte de diversos tipos de optimização da utilização da capacidade disponível; avaliação da adequação dos caminhos seleccionados ao suporte de tolerância a falhas; e estudo de formas de gestão dinâmica da distribuição do tráfego na rede com base nesta filosofia.

Referências

1. M. Caesar, M. Casado, T. Koponen, J. Rexford, and S. Shenker. Dynamic route recomputation considered harmful. *SIGCOMM Comput. Commun. Rev.*, 40(2):66–71, Apr. 2010.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.
3. E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1(269-270):6, 1959.
4. O. M. Heckmann. *The Competitive Internet Service Provider*. Wiley Series in Communications Networking & Distributed Systems. Wiley-Interscience, Chichester, UK, 1 edition, 2006.
5. A. Markopoulou, G. Iannaccone, C.-N. C. S. Bhattacharyya, Y. Ganjali, and C. Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM Transactions on Networks*, 16(4):749–762, 2008.
6. J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. Mogul. Spain: Design and algorithms for constructing large data-center ethernets from commodity switches. Technical report, Tech. Rep. HPL-2009-241, HP Labs, 2009.
7. J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 18–18. USENIX Association, 2010.
8. E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. *IETF, RFC 3031*, 2001.
9. H. Saito, Y. Miyao, and M. Yoshida. Traffic engineering using multiple multipoint-to-point lsps. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 894 –901 vol.2, 2000.
10. M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. *SIGMETRICS Performance Evaluation Review-Measurement and Evaluation*, 39(1):97, 2011.
11. N. Wang, K. H. Ho, G. Pavlou, and M. Howarth. An overview of routing optimization for internet traffic engineering. *IEEE Communications Surveys and Tutorials*, 10(1):36–56, 2008.

Especificação e Verificação de Protocolos para Programas MPI

Nuno Dias Martins, César Santos, Eduardo R. B. Marques, Francisco Martins, and Vasco T. Vasconcelos

LaSIGE, Faculdade de Ciências, Universidade de Lisboa

Resumo Message Passing Interface (MPI) é a infraestrutura padrão de troca de mensagens para o desenvolvimento de aplicações paralelas. Duas décadas após a primeira versão da sua especificação, aplicações baseadas em MPI correm hoje rotineiramente em super computadores e em grupos de computadores. Estas aplicações, escritas em C ou Fortran, exibem intrincados comportamentos, o que torna difícil verificar estaticamente propriedades importantes, tais como a ausência de corridas ou impasses.
A abordagem que apresentamos neste artigo pretende validar programas escritos na linguagem C utilizando primitivas de comunicação MPI. Encontra-se dividida em duas linhas orientadoras. Definimos um protocolo de comunicação numa linguagem que criámos para o efeito, inspirada nos tipos de sessão. Este protocolo é depois traduzido para a linguagem de um verificador de *software* para C, o VCC. Anotamos depois o programa C com asserções que levam o verificador a provar ou a refutar a conformidade do programa com o protocolo. Grande parte das anotações necessárias são introduzidas automaticamente por um anotador que desenvolvemos. Até à data conseguimos validar com sucesso vários programas que atestam o sucesso da nossa abordagem.

1 Introdução

Message Passing Interface (MPI) [9] é um padrão para programação de aplicações paralelas de alto desempenho, suportando plataformas de execução com centenas de milhares de *cores*. Baseado no paradigma de troca de mensagens, a infraestrutura MPI pode ser utilizada em programas C ou Fortran.

Um único programa define o comportamento dos vários processos (de acordo com o paradigma *Single Program, Multiple Data*), utilizando chamadas a primitivas MPI, por exemplo para comunicações ponto-a-ponto ou para comunicações colectivas. O uso de MPI levanta vários problemas. É muito fácil escrever um programa contendo um processo que bloqueie indefinidamente à espera de uma mensagem, que exiba corridas na troca de mensagens entre processos, ou em que o tipo e a dimensão dos dados enviados e esperados por dois processos não coincidam. Em suma, não é possível, de um modo geral, garantir à partida (em tempo de compilação) propriedades fundamentais sobre a execução de um programa.

Lidar com este desafio não é trivial. A verificação de programas MPI utiliza regularmente técnicas avançadas como verificação de modelos ou execução simbólica [4,11]. Estas abordagens deparam-se frequentemente com o problema de escalabilidade, dado

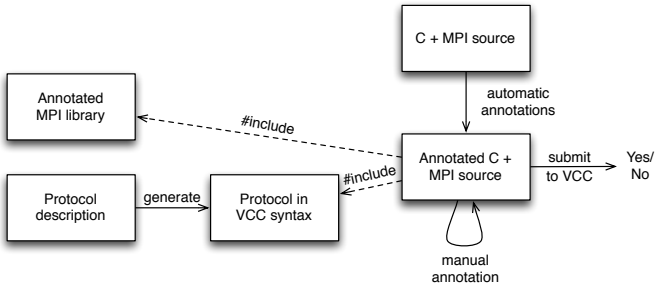

Figura 1. Abordagem de verificação

o espaço de procura crescer exponencialmente com o número de processos considerados. Assim sendo, a definição do espaço de procura pode estar limitado na prática a menos que uma dezena de processos na verificação de aplicações *real-world* [12]. A verificação é adicionalmente complicada por vários aspectos adicionais como a existência de diversos tipos de primitivas MPI com diferentes semânticas de comunicação [11], ou a dificuldade em destrinçar o fluxo colectivo e individual de processos num único corpo comum de código [1].

A abordagem que consideramos para a verificação de programas MPI é baseada em *tipos de sessão multi-participante* [6]. A ideia base é começar por especificar o protocolo de comunicação a ser respeitado pelo conjunto dos participantes constantes num dado programa. Este protocolo é expresso numa linguagem definida para o efeito (baseada em tipos de sessão). Validamos depois a aderência ao protocolo por parte de um dado programa. Se a relação de aderência for efectiva ficam garantidas propriedades como a ausência de condições de impasse e a ausência de corridas nas trocas de mensagens.

A visão global do processo de especificação e verificação de protocolos está ilustrada na figura 1. Definimos uma linguagem formal de descrição de protocolos, apropriada à expressão dos padrões mais comuns de programas MPI. A partir de um protocolo expresso nessa linguagem (*Protocol description*) geramos um *header* C que exprime o tipo num formato compatível com a ferramenta de verificação dedutiva VCC [2] (*Protocol in VCC syntax*). Para além do protocolo, a verificação é ainda guiada por um conjunto de contratos pré-definidos para primitivas MPI (*Annotated MPI library*) e por anotações no corpo do programa C (*Annotated C+MPI source*), quer geradas automaticamente ou, em número tipicamente mais reduzido, introduzidas pelo programador. Estes aspectos representam uma evolução bastante relevante do nosso trabalho anterior [8], onde não fazíamos uso de uma linguagem formal para definição do protocolo, e onde o processo dependia exclusivamente de anotações e definições produzidas manualmente.

O resto do artigo está estruturado do seguinte modo. Começamos com um programa exemplo (secção 2) ilustrando o tipo de primitivas MPI que endereçamos, e que servirá de base de discussão. Apresentamos depois a linguagem de especificação de protocolos (secção 3) e o processo de verificação de programas C face a um dado protocolo (sec-

ção 4). Terminamos o artigo com algumas conclusões e discussão sobre trabalho futuro (secção 5).

2 Exemplo motivador

O exemplo que consideramos é o do cálculo das diferenças finitas a uma dimensão, através do algoritmo iterativo descrito em [3]. A partir de um vetor inicial X^0 calculam-se sucessivas aproximações à solução do problema X^1, X^2, . . ., até que uma condição de convergência se verifique ou que o número máximo de iterações tenha sido atingido. A figura 2 mostra o código de um programa C, adaptado também de [3]. Ilustramos seguidamente os seus aspectos essenciais.

```
1  int main(int argc, char** argv) {
2    int procs;              // Number of processes
3    int rank;               // Process rank
4    MPI_Init(&argc, &argv);
5    MPI_Comm_size(MPI_COMM_WORLD, &procs);
6    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7    ...
8    int psize = atoi(argv[1]);            // Global problem size
9    if (rank == 0)
10      read_vector(work, lsize * procs);
11   MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0,
         MPI_COMM_WORLD);
12   int left  = (procs + rank - 1) % procs;  // Left neighbour
13   int right = (rank + 1) % procs;          // Right neighbour
14   int iter = 0;
15   // Loop until minimum differences converged or max iterations attained
16   while (!converged(globalerr) && iter < MAX_ITER)) {
17     ...
18     if (rank == 0) {
19       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
20       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
21       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
22       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
23     } else if (rank == procs - 1) {
24       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
25       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
26       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
27       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
28     } else {
29       MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
30       MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
31       MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
32       MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status)
33     }
34     ...
35     MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
36     ...
37   }
38   ...
39   if (converged(globalerr)) { // Gather solution at rank 0
40     MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0,
           MPI_COMM_WORLD);
41     ...
42   }
43   ...
44   MPI_Finalize();
45   return 0;
46 }
```

Figura 2. Excerto do programa MPI para o problema das diferenças finitas (adaptado de [3])

O programa exemplo estipula o comportamento de todos os processos, podendo o comportamento de cada processo participante divergir em função do seu número de processo, designado por *rank*. O número total de processos, `procs` na figura, definido apenas em tempo de execução, e o `rank` de cada processo são obtidos respetivamente através das primitivas `MPI_Comm_size` e `MPI_Comm_rank` (linhas 5 e 6).

Neste algoritmo o participante `0` começa por ler o vetor inicial X^0 (linhas 9–10) e distribui depois o vetor por todos os participantes (linha 11, chamada a `MPI_Scatter`). Cada participante fica responsável pelo cálculo local de uma parte do vetor, de tamanho igual para todos os participantes.

Seguidamente, o programa executa um ciclo (linhas 16–37), especificando trocas de mensagens ponto-a-ponto (`MPI_Send`, `MPI_Recv`) entre cada processo e os seus vizinhos esquerdo (`left`) e direito (`right`), considerando uma topologia em anel segundo o *rank* dos participantes. A troca de mensagens tem como objetivo distribuir os valores de fronteira necessários ao cálculo local devido a cada participante. A diferença entre os vários participantes, `0` (linhas 19–22), `procs-1` (linhas 23–27), e restantes (linhas 28–32) na ordem das chamadas a `MPI_Send` e `MPI_Recv` tem por fim evitar bloqueio, já que a troca de mensagens nas primitivas consideradas é bloqueante (síncrona e *unbuffered*) tanto para quem envia como para quem recebe.

Após a troca de mensagens, e ainda dentro do ciclo, o erro global é calculado com uma operação de redução e comunicado a todos os participantes (`MPI_Allreduce`, linha 35). O ciclo termina quando se verifica a condição de convergência, ou após um número pré-definido de iterações. Após o ciclo, se tiver havido convergência, o participante `0` agrega a solução final, recebendo de cada um dos outros uma parte do vetor (usando `MPI_Gather`, linhas 39–40).

O código apresentado é extremamente sensível a variações na estrutura das operações MPI. Por exemplo, a omissão de uma qualquer das operações de envio/receção de mensagem nas linhas 19–32 conduz a uma condição de bloqueio onde pelo menos um processo ficará eternamente à espera de conseguir enviar ou receber uma mensagem. Outro exemplo: trocando as linhas 20 e 21 no código, teríamos uma situação de bloqueio em todos os processos no envio ou receção de uma mensagem.

3 A linguagem de especificação de protocolos

Os protocolos que regem as comunicações globais num programa MPI são descritos numa linguagem de protocolos, desenhada especificamente para o efeito. As ações básicas dos nossos protocolos descrevem comunicações MPI individuais (**`message`**, **`gather`**, **`scatter`**, **`broadcast`**), a obtenção do número de processos (**`size`**) e a abstração sobre valores (**`val`**). Estas ações básicas são compostas através de operadores de sequência (`;`), ciclo (**`foreach`**), e de fluxo de controlo coletivo (**`loop`** e **`choice`**).

Um possível protocolo para o nosso exemplo encontra-se na figura 3. A linha 2 introduz o número de processos através da variável `p` e a linha 3 a dimensão do problema através da variável `n`. A diferença é que **`size`** corresponde a uma primitiva MPI, enquanto que **`val`** não tem correspondência com nenhuma primitiva MPI. No segundo caso o valor de `n` tem de ser indicado explicitamente pelo programador (*vide* secção 4). Os valores constantes nos protocolos podem ser de género inteiro (**`integer`**) ou vírgula flutuante

```
1  protocol FiniteDifferences {
2    size p: positive;                  // número de processos
3    val n: {x: natural | x % p == 0}; // dimensão do problema
4    scatter 0 float[n];
5    loop {
6      foreach i: 0 .. p - 1 {
7        message i, (p + i - 1) % p float
8        message i, (i + 1) % p float;
9      };
10     allreduce max float
11   };
12   choice
13     gather 0 float[n]
14   or
15     {}
16 }
```

Figura 3. Protocolo para o programa de diferenças finitas

(`float`), bem como vetores (`integer[n]` ou `float[n]`). Além disso qualquer um destes géneros pode ser *refinado*. O género `{x:natural|x%p==0}` denota um número inteiro não negativo, múltiplo do número de processos `p`. Os géneros `natural` e `positive` são na verdade abreviaturas de `{x:integer|x>=0}` e de `{x:integer|x>0}`, respetivamente.

O exemplo na figura 3 contém também algumas primitivas de comunicação. A linha 4 descreve uma operação de distribuição, iniciada pelo processo *rank* `0`, de um vetor de números em vírgula flutuante de dimensão `n`, sendo que o processo i (`0` $\leq i <$ `p`) recebe um i-ésimo pedaço (de dimensão `n/p`) do vetor original. As linhas 7 e 8 descrevem trocas de mensagens ponto a ponto. No primeiro caso é trocada uma mensagem entre o processo *rank* `i` e o processo *rank* `(i+1)%p` contendo um número em vírgula flutuante. A linha 10 descreve uma operação em que todos os processos comunicam um número em vírgula flutuante, o máximo entre eles é calculado e posteriormente distribuído por todos os processos. Finalmente a linha 13 descreve a operação `scatter`, recolhendo no processo `0` um vetor de dimensão `n`, composto por secções proveniente dos vários processos.

O ciclo `foreach` nas linhas 6–9 é intuitivamente equivalente à composição sequencial de `p` cópias das linhas 7–8, com `i` substituído por `0`, `1`, ..., `p-1`. As linhas 5 e 12 são exemplos do que apelidamos de *estruturas de controle coletivas*. O primeiro caso descreve um ciclo em que todos os processos decidem conjuntamente, mas sem comunicar entre eles, prosseguir ou abandonar o ciclo. O segundo caso descreve uma escolha em que todos os processos decidem conjuntamente (novamente sem comunicar) seguir pela linha 13 ou pela linha 15. A linha 15 denota um bloco de operações vazio.

Para esta linguagem implementámos um *plugin* Eclipse que verifica a boa formação dos protocolos e que gera um ficheiro na linguagem VCC, tal como descrito na secção seguinte (a figura 4 apresenta um exemplo do código gerado). O *plugin* foi escrito recorrendo à ferramenta Xtext [13].

4 Verificação de protocolos

A aderência de um programa C+MPI a um protocolo de comunicação é verificado utilizando a ferramenta VCC, tendo em conta os seguintes passos preliminares (*vide* figura 1).

```
_(ghost _(pure) \SessionType ftype (\integer rank)
 _(ensures \result ==
  seq(
   action(size(), intRef(\lambda \integer y; y>0, 1)),
   abs(body(\lambda \integer p;
    seq(
     action(val(), intRef(\lambda \integer x; x>0 && x%p==0, 1)),
     abs(body(\lambda \integer n;
      seq(
       action(scatter(0), floatRef(\lambda float v; \true, n)),
       seq(
        loop(
         seq(
          foreach(0, p-1,
           body(\lambda \integer i;
            seq(
             message(i, (p+i-1)%p,
                        floatRef(\lambda float v; \true, 1))[rank],
             message(i, (i+1)%p,
                        floatRef(\lambda float v; \true, 1))[rank]))),
          action(allreduce(MPI_MAX),floatRef(\lambda float v; \true, 1)))),
        choice(
         action(gather(0), floatRef(\lambda float v; \true, n)),
         skip()
        ))))))))
  );
)
```

Figura 4. Diferenças finitas, função de projecção na linguagem VCC

- A partir da especificação do protocolo é gerado automaticamente um *header* C contendo a codificação do protocolo na linguagem VCC. Este *header* deve ser incluído no ficheiro C principal por forma a importar a definição do protocolo.
- O corpo do programa é modificado através da introdução automática de anotações, necessárias para guiar a verificação e orientadas pelo fluxo de controlo de programa. Não é necessário anotar cada chamada MPI, uma vez que os contratos destas estão definidos à partida no *header* `mpi.h`.
- O programador adiciona manualmente anotações complementares, incluindo marcas para guiar a verificação/anotação em aspectos que não conseguem ser inferidos automaticamente pelo passo acima, ou que se relacionam com aspectos técnicos de verificação de programas C utilizando o VCC, por exemplo associados ao uso de memória.

Ilustramos em seguida estes aspectos, juntamente com o processo de verificação por parte do VCC.

O protocolo em formato VCC. O protocolo para o problema das diferenças finitas constante na figura 3 é traduzido, pela ferramenta introduzida na secção anterior, na função VCC descrita na figura 4. São relevantes à sua compreensão alguns detalhes sintáticos: os blocos de anotação VCC são expressos na forma `_(annotation block)`; a palavra chave **`ghost`** indica que determinado bloco de anotações codifica uma definição "fantasma" necessária à lógica de verificação, mas de outra forma externa à lógica do programa C em si; a palavra chave **`pure`** descreve uma função sem efeitos colaterais, e a palavra chave **`ensures`** descreve a pós-condição de uma dada função; a sintaxe `(\lambda type x; f[x])` codifica uma função anónima de domínio `type`.

De modo a capturar fielmente um protocolo como aquele descrito na figura 3, criámos um tipo de dados VCC a que chamámos `\SessionType` e que contém construtores para cada uma das acções básicas (`size`, `val`, `scatter`) e das primitivas de controle (`seq`, `loop`, `choice`, `foreach`). A única exceção é a primitiva **`message`** que é traduzida num construtor `send`, `recv` ou `skip`, tal como descrito abaixo. Deste modo, a ferramenta gera uma *função de projeção* VCC com assinatura

```
\SessionType ftype(\integer rank)
```

e que representa o protocolo global quando visto pelo prisma do participante `rank`, em linha com a teoria estabelecida para tipos de sessão multi-participante [6]. Tanto os géneros refinados (`{x:natural|x%p==0}`, linha 3 na figura 3) como a introdução de variáveis (`n`, na mesma linha) são codificados recorrendo a expressões **`lambda`** (linhas 5–7 na figura 4). As mensagens ponto-a-ponto são traduzidas para a forma `message(from, to,type)[rank]`, tal como ilustrado na figura 3, linhas 11–12 e 13–14. Estes termos são depois transformados (projetados) em função de `rank`, como definido pelos seguintes axiomas na lógica VCC (e em linha com o descrito em [6]).

```
_(axiom \forall \integer from, to;
        \forall \SessionData sd;
        from != to ==> message(from,to,sd)[from] == action(send(to),sd))
_(axiom \forall \integer from, to;
        \forall \SessionData sd;
        from != to ==> message(from,to,sd)[to] == action(recv(from),sd))
_(axiom \forall \integer from, to, r;
        \forall \SessionData sd;
        r != from && r != to ==> message(from,to,sd)[r] == skip())
```

O processo de verificação. A verificação da aderência do programa C ao tipo projetado analisa o fluxo de controlo do programa entre o ponto de inicialização (chamada a `MPI_Init`, linha 4, figura 2) e de término (`MPI_Finalize`, linha 44, mesma figura). O protocolo é inicializado através da função `ftype` e é depois progressivamente reduzido, de modo a que no término esteja num estado congruente a `skip()` (por exemplo, um ciclo coletivo vazio—`loop{}`—ou um *foreach* sem desdobramentos possíveis—`foreach (0,-1,...)`—são ambos congruentes a `skip()`). Para manter estado, a verificação manipula uma variável **`ghost`** do tipo `\SessionType` desde o ponto de entrada da função `main()`. A inicialização e término são definidos com chamadas a respetivamente `MPI_Init` e `MPI_Finalize`, cujos contratos ilustram a lógica global de verificação:

```
int MPI_Init(... _(ghost GhostData gd) _(out \SessionType typeOut))
  _(ensures typeOut == ftype(gd->rank))
  ...
int MPI_Finalize(... _(ghost \SessionType typeIn))
  _(ensures congruent(typeIn, skip())
  ...
```

O predicado `congruent` usado em `MPI_Finalize` exprime a congruência entre dois termos `\SessionType`.

Entre inicialização e término, a verificação tem de lidar com a progressiva redução do tipo em função de chamadas a primitivas de comunicação. Como exemplo, considere-se um fragmento do contrato de `MPI_Send`.

```
int _MPI_Send(void *buf, int count,
              MPI_Datatype datatype, int dest, ...
```

```
            _(ghost \SessionType typeIn)
            _(out \SessionType typeOut))
_(requires actionType(first(inType)) == send(dest))
_(requires actionLength(first(inType)) == count)
_(requires refTypeCheck(refType(first(inType)), buf, count))
_(requires datatype == MPI_INT ==> \thread_local_array ((int *) buf, count))
_(requires datatype == MPI_FLOAT ==> \thread_local_array ((float *) buf, count))
_(ensures  outType == next(inType))
...
```

O contrato estipula (na ordem mostrada) que: a primeira ação possível para o tipo de entrada é `send(dest)`, onde `dest` é o destinatário especificado no programa; a dimensão do vetor constante na ação corresponde ao parâmetro `count` da função `MPI_Send`; os dados a transmitir verificam as restrições de refinamento de tipos e são regiões válidas de memória; e, finalmente, como pós-condição, que o tipo após a execução da primitiva MPI é a continuação do tipo de entrada. As restantes primitivas de comunicação têm contratos similares.

Para lidar com o fluxo de controlo do programa, em particular ciclos e escolhas coletivas, são necessárias anotações diretas no corpo do programa. Estas são geradas automaticamente, num processo detalhado abaixo. Concentramo-nos agora no seu significado, usando para tal o ciclo coletivo do exemplo das diferenças finitas.

```
_(ghost \SessionType body = loopBody(_type);)
_(ghost \SessionType cont = next(_type);)
while (!converged(globalerr) && iter < MAX_ITER)
  ...
{
  _(ghost _type = body;)
  ...
  _(assert congruent(_type, skip())
}
_(ghost _type = cont);
...
```

O fragmento ilustra a extração dos tipos correspondentes ao corpo do ciclo (`body`) e à sua continuação (`cont`). O corpo tem de ser um construtor `loop { ... }`, o que no caso acontece na linha 12, figura 4). A verificação determina que o tipo no final do corpo do ciclo tem de ser congruente com `skip()`. Após o ciclo, a verificação prossegue com a análise do resto do programa, utilizando a continuação `cont` do termo.

A geração de anotações. Uma grande parte das anotações discutidas acima é introduzida de forma automática por um programa anotador. A função do anotador é ler código C e derivar o grosso das anotações necessárias à validação do programa. Em termos de implementação, o anotador utiliza a plataforma clang/LLVM [7] para processar código C. O anotador é incapaz de decidir se está ou não em presença de fluxo coletivo (**`loop`** ou **`choice`**) ou de um ciclo **`foreach`**. Para o guiar, o programador deve introduzir no código C marcas `_collective_` e `_foreach_`. Baseado nestas marcas, o processo de anotação automático está sumariado na tabela 1.

A marca `_collective_` identifica uma escolha ou ciclo coletivo. As anotações geradas destinam-se a extrair do tipo de sessão os corpos (dos ciclos ou da escolha) e as respetivas continuações. De forma análoga, a marca `_foreach_` identifica ciclos no código C que devem ser vistos como em correspondência com protocolos **`foreach`**. A marca especifica qual a variável de iteração do ciclo e em resposta o anotador gera um