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

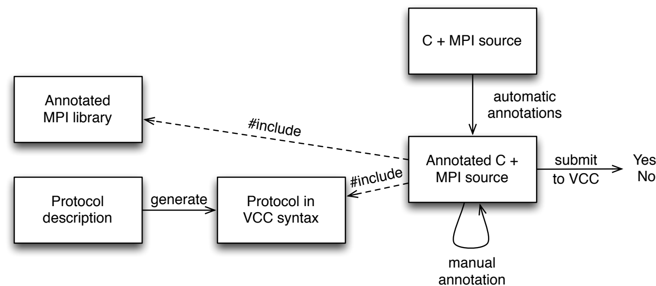

Figura 1. Abordagem de verificação

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, $\ldots$, 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).

```
1  _(ghost _(pure) \SessionType ftype (\integer rank)
2   _(ensures \result ==
3    seq(
4     action(size(), intRef(\lambda \integer y; y>0, 1)),
5     abs(body(\lambda \integer p;
6      seq(
7       action(val(), intRef(\lambda \integer x; x>0 && x%p==0, 1)),
8       abs(body(\lambda \integer n;
9        seq(
10        action(scatter(0), floatRef(\lambda float v; \true, n)),
11       seq(
12        loop(
13         seq(
14          foreach(0, p-1,
15           body(\lambda \integer i;
16            seq(
17             message(i, (p+i-1)%p,
18                       floatRef(\lambda float v; \true, 1))[rank],
19             message(i, (i+1)%p,
20                       floatRef(\lambda float v; \true, 1))[rank]))),
21          action(allreduce(MPI_MAX),floatRef(\lambda float v; \true, 1)))),
22        choice(
23         action(gather(0), floatRef(\lambda float v; \true, n)),
24         skip()
25        )))))))))
26   );
27 )
```

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

Código original	**Código anotado**

`/* Escolhas coletivas */`
`if( _collective_(expr)) {`
`  ...`
`} else {`
`  ...`
`}`
	`_(ghost \SessionType _cTrue  = choiceTrue(_type);)`
	`_(ghost \SessionType _cFalse = choiceFalse(_type);)`
	`_(ghost \SessionType _cCont  = next(_type);)`
	`if (expr) {`
	`  _(ghost _type = cTrue;)`
	`  ...`
	`  _(assert congruent(_type, skip()))`
	`} else {`
	`  _(ghost _type = cFalse;)`
	`  ...`
	`  _(assert congruent(_type, skip()))`
	`}`
	`_(ghost _type = cCont;)`

`/* Ciclos coletivos */`
`while (_collective_(expr)) {`
`  ...`
`}`
`/* similar p/ ciclos`
`   for e do-while */`
	`_(ghost \SessionType _lBody = loopBody(_type);)`
	`_(ghost \SessionType _lCont = next(_type);)`
	`while (expr) _ampi_loop {`
	`  _(ghost _type = _lBody;)`
	`  ...`
	`  _(assert congruent(_type, skip()))`
	`}`
	`_(ghost _type = lCont;)`

`/* foreach  */`
`int v = ...;`
`...`
`while (_foreach_(v, expr)) {`
`  ...`
`}`
`/* similar p/ ciclos`
`   for e do-while */`
	`int v = ...;`
	`...`
	`_(ghost STMap0       fBody = foreachBody(_type);)`
	`_(ghost \SessionType fCont = foreachCont(_type);)`
	`while(expr) {`
	`  _(ghost _type = fBody[v];)`
	`  ...`
	`  _(assert congruent(_type, skip())`
	`}`
	`_(ghost _type = fCont;)`

Tabela 1. Geração de anotações

conjunto de anotações para definir o tipo de sessão a ser consumido em cada iteração do ciclo, dependente do valor da variável de ciclo. Isto é ilustrado para a variável `v` no caso `_foreach_` da tabela 1.

O caso particular do ciclo do protocolo das diferenças finitas (linhas 6–9, figura 3), por não identificar nem o emissor nem o recetor das mensagens através de um *rank* constante, não requer ciclo algum ao nível do código C, como aliás pode ser verificado analisando as linhas 18–33 da figura 2. Consideremos alternativamente um protocolo em que o participante `0` envia uma mensagem para todos os outros.

```
foreach i = 0 .. procs-1 { message 0 i float }
```

Por exemplo, a implementação deste protocolo pode ser expressa da forma abaixo, definindo um ciclo para o participante `0`.

```
if (rank == 0) {
  int i = 0;
  while ( _foreach_(i, i < procs) )
    MPI_Send (... i++ ...); // 0 envia para i
} else {
   MPI_Recv (... 0 ...); // recebe de 0
}
```

Deste modo a geração de código anotado pode proceder como ilustrado na tabela 1.

Há uma ação nos protocolos que não corresponde a primitiva MPI alguma: `val`. É utilizada para que todos os participantes "injectem" no protocolo um mesmo valor, tipicamente um parâmetro do programa. À variável `n` do protocolo (linha 3, figura 3) corresponde a variável `psize` do programa C (linha 8, figura 2). Para que a correspondência seja feita é necessário que o programador coloque a marca `_apply_(psize)_` no ponto certo do programa, entre a chamada a `MPI_Comm_size` e a `MPI_Scatter`, de acordo com o protocolo da figura 3.

Introduzindo as marcas necessárias no programa da figura 2, e correndo o nosso anotador, obtemos o programa da figura 5, pronto a ser submetido ao VCC.

Avaliação. Avaliámos o processo de verificação para alguns exemplos retirados de livros de texto, considerando duas medidas: o tempo de verificação e a comparação entre o número de anotações geradas automaticamente e o número de anotações que tiveram de ser introduzidas manualmente. Os exemplos são disponibilizados para consulta em `http://www.di.fc.ul.pt/~edrdo/ampi-0.2.zip`. Os resultados para o tempo de verificação em função do número de processos encontram-se tabelados abaixo.

Programa	4	8	16	32	64
Diferenças Finitas [3]	14,8	28,0	70,1	260,0	—
Iteração de Jacobi [10]	11,0	16,0	12,4	11,5	12,0
Simulação N-body [5]	6,1	8,3	42,8	—	—

São mostrados os tempos de verificação de três programas distintos, tendo em conta a variação do número de processos entre 4 e 64. A unidade de tempo é o segundo e as entradas vazias indicam que a verificação não terminou após 5 minutos de espera. A primeira observação é um desempenho estável no caso da iteração de Jacobi. Isso poderá ser explicado pelo facto de o exemplo em causa usar apenas operações de comunicações coletiva e ter uma reduzida distinção entre as operações dos vários participantes. O contraste é bastante percetível com os outros dois exemplos, as diferenças finitas e a simulação N-body, que incluem comunicações ponto-a-ponto e tipos de comportamento distintos para participantes ou grupos de participantes, e para as quais o tempo de verificação cresce mais rápido do que o número de processos.

Na tabela abaixo estão detalhados o número de linhas de código no programa original, o número de anotações geradas automaticamente e manualmente.

Programa	Linhas	Anotações auto.	Anotações manuais
Diferenças Finitas [3]	256	38	17 (2,1)
Produto escalar de vectores [10]	357	38	30 (5,0)
Iteração de Jacobi [10]	429	55	56 (7,0)
Simulação N-body [5]	362	52	26 (3,7)

Verifica-se no caso geral que o número de anotações manuais é inferior ao número das automáticas, com uma exceção. Na coluna das anotações manuais é listado entre parênteses o número de anotações por função C anotada. Este número é relevante na medida em que cada função C contendo primitivas MPI tem de ser invariavelmente anotada com as necessárias pré e pós condições (algo não discutido neste artigo).

```
int main(int argc, char** argv _ampi_arg_decl) {
  int procs;              // Number of processes
  int rank;               // Process rank
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &procs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...
  int psize = atoi(argv[1]);            // Global problem size
  _apply_(psize);
  if (rank == 0)
    read_vector(work, lsize * procs);
  MPI_Scatter(work, lsize, MPI_FLOAT, &local[1], lsize, MPI_FLOAT, 0,
       MPI_COMM_WORLD);
  int left  = (procs + rank - 1) % procs;  // Left neighbour
  int right = (rank + 1) % procs;          // Right neighbour
  int iter = 0;
  // Loop until minimum differences converged or max iterations attained
  _(ghost \SessionType lBody = loopBody(_type);)
  _(ghost \SessionType lCont = next(_type);)
  while (!converged(globalerr) && iter < MAX_ITER)
    _(writes &globalerr)
    _(writes \array_range(local, (unsigned) lpsize + 2))
  {
    _(ghost _type = lBody;)
    if (rank == 0) {
      MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
      MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
      MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
    } else if (rank == procs - 1) {
      MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status);
      MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
      MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
    } else {
      MPI_Recv(&local[0],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD, &status);
      MPI_Send(&local[1],       1, MPI_FLOAT, left, 0, MPI_COMM_WORLD);
      MPI_Send(&local[lsize],   1, MPI_FLOAT, right,0, MPI_COMM_WORLD);
      MPI_Recv(&local[lsize+1], 1, MPI_FLOAT, right,0, MPI_COMM_WORLD, &status)
    }
    ...
    MPI_Allreduce(&localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, MPI_COMM_WORLD);
    ...
    _(assert congruence(_type, skip())
  }
  _(ghost _type = lCont;)
  ...
  _(ghost \SessionType cTrue = choiceTrue(_type);)
  _(ghost \SessionType cFalse = choiceFalse(_type);)
  _(ghost \SessionType cCont = next(_type);)
  if (converged(globalerr)) { // Gather solution at rank 0
    _(ghost _type = cTrue;)
    MPI_Gather(&local[1], lsize, MPI_FLOAT, work, lsize, MPI_FLOAT, 0,
         MPI_COMM_WORLD)
    ...
    _(assert congruence(_type, skip()))
  } else {
    _(ghost _type = cFalse;)
    ...
    _(assert congruence(_type, skip()))
  }
  _(ghost _type = cCont;)
  MPI_Finalize();
  return 0;
}
```

Figura 5. O programa das diferenças finitas com anotações VCC

5 Conclusão

Os resultados descritos neste artigo inserem-se num programa mais vasto de verificação de propriedades relacionadas com a troca de mensagens em programas *real-world*. Com a infraestrutura que montámos conseguimos, com limitada intervenção humana, verificar alguns programas com 200–500 linhas de código C.

Muitas perguntas se levantam. Trabalhámos com programas retirados de livros de texto [3,5,10], implementações de algoritmos bem conhecidos e para os quais foi relativamente simples gerar o protocolo, programas com uma codificação mais ou menos cuidada e que requerem poucas anotações manuais, programas que utilizam um subconjunto algo restrito de primitivas MPI. Mas até nestes casos se começam a verificar problemas com a dimensão do problema.

O que acontece com programas *real-world*, programas que não verificam algum destes pressupostos? Entre outras coisas, teremos de: a) suportar mais primitivas MPI. Tencionamos endereçar seguidamente as operações imediatas (`MPI_Isend`, `MPI_Irecv` e `MPI_Wait`); b) trabalhar para minimizar o número de anotações manuais necessárias ao bom funcionamento do anotador, utilizando alguma forma de inferência; c) fazer correr as nossas ferramentas sobre mais programas.

Agradecimentos. Este trabalho é suportado pela FCT através do projeto Advanced Type Systems for Multicore Programming (PTDC/EIA-CCO/122547/2010) e do programa multianual LaSIGE (PEst-OE/EEI/UI0408/2011).

Referências

1. Bronevetsky, G.: Communication-sensitive static dataflow for parallel message passing applications. In: CGO. pp. 1–12. IEEE Computer Society (2009)
2. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs, LNCS, vol. 5674, pp. 23–42. Springer (2009)
3. Foster, I.: Designing and building parallel programs. Addison-Wesley (1995)
4. Gopalakrishnan, G., Kirby, R.M., Siegel, S., Thakur, R., Gropp, W., Lusk, E., De Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal analysis of MPI-based parallel programs. CACM 54(12), 82–91 (2011)
5. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: portable parallel programming with the message passing interface, vol. 1. MIT press (1999)
6. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
7. The LLVM Team. Clang: a C Language Family Frontend for LLVM. http://clang.llvm.org/ (2012)
8. Marques, E., Martins, F., Vasconcelos, V., Ng, N., Martins, N.: Towards deductive verification of MPI programs against session types. In: PLACES (2013), (to appear)
9. MPI Forum: MPI: A Message-Passing Interface Standard – Version 3.0. High-Performance Computing Center Stuttgart (2012)
10. Pacheco, P.: Parallel programming with MPI. Morgan Kaufmann (1997)
11. Siegel, S., Gopalakrishnan, G.: Formal analysis of message passing. In: VMCAI. LNCS, vol. 6538, pp. 2–18 (2011)
12. Siegel, S., Rossi, L.: Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In: EuroPVM/MPI. LNCS, vol. 5205, pp. 274–282 (2008)
13. Xtext—language development made easy!, http://www.eclipse.org/Xtext/