Paralelização parcial de programas *SISAL* utilizando a biblioteca *MPI*

Raul Junji Nakashima
Instituto de Física de São Carlos
Grupo de Instrumentação Eletrônica
USP-São Carlos
Av. Dr. Carlos Botelho, 1465
São Carlos - SP - Brasil
CEP 13560-250

Gonzalo Travieso
Instituto de Física de São Carlos
Grupo de Instrumentação Eletrônica
USP-São Carlos
Av. Dr. Carlos Botelho, 1465
São Carlos - SP - Brasil
CEP 13560-250

Resumo

O trabalho desenvolvido teve como objetivo a implementação de um método para a paralelização parcial de programas, escritos na linguagem funcional *SISAL*, utilizando as bibliotecas do padrão *MPI* (*Message Passing Interface*). Para tal, propusemos a transformação dos programas *SISAL* através do particionamento do *loop* paralelo *forall*, utilizando o método de particionamento *slice* e a implementação do paralelismo utilizando o modelo de paralelização *SPMD* (*Single Program Multiple Data*) com programas no estilo *mestre/escravo*. A validação de nossa proposta foi obtida através da realização de testes onde foram comparados os resultados obtidos com os programas *SISAL* originais e os programas *SISAL* com as alterações propostas.

Abstract

This paper describes a method for the partial parallelization of *SISAL* programs into programs with calls to *MPI* rotines. We focused on the parallelization of the *forall* loop (through slicing of the index range). The generated code is a *master/slave SPMD* program. The work was validated through the compilation of some simple *SISAL* programs and comparison of the results with an unmodified version.

1. Introdução

SISAL [FC90, Ske91, Can92b] é uma linguagem funcional que tem provado ser útil para a programação de aplicações numericamente intensivas, especialmente aplicações paralelas [FA95]. O *OSC* (*Optimizing SISAL Compiler*) [Can92a], cria código para diversos processadores de memória compartilhada e vetoriais [FC90]. No entanto, ele não cria código para máquinas de memória distribuída, [FA95].

Trabalhos estão sendo realizados na tentativa de suprir esta lacuna, entre eles: o desenvolvimento de análises estáticas para a distribuição de *arrays* e *loops*, inclusão de diretivas e *pragmas* para o alinhamento e distribuição de dados, desenvolvimento de um compilador baseado no *Split-C* e um baseado no *MPI* e explorar a sobreposição de computação/comunicação (Mais informações podem ser obtidas no endereço `http://zelea.usc.edu/pdpc/projects/sisal/PI2_26.html`).

O objetivo de nosso trabalho foi apresentar um método para a paralelização parcial de programas *SISAL* usando *MPI*, um padrão de interface para passagem de mensagem (paradigma largamente usado em certas classes de máquinas paralelas, especialmente aquelas com memória distribuída) [MPI94]. Um padrão para passagem de mensagem é um componente chave na construção de um ambiente de computação concorrente no qual aplicações, bibliotecas de *software* e ferramentas possam ser usadas transparentemente entre diferentes máquinas [MPI94a].

O modelo de programação *SPMD* (*Single Program Multiple Data*) [FC95, AG94], com processos executando no estilo *mestre/escravo* [HB93] foi escolhido para receber as rotinas de comunicação coletiva de *MPI*. Para a transformação de um programa *SISAL* para o modelo *SPMD* realizou-se o particionamento *slice*, [SC90], da estrutura *forall*, [Ske91], de modo a inserir cada fatia obtida em um processo diferente. O código fonte manipulado foi o do formato *IF1*, [SG85, SW85], representação em forma grafos acíclicos da linguagem *SISAL*.

Para a implementação do modelo proposto utilizamos o compilador *SISAL* do *Lawrence Livermore National Laboratories*, *OSC*; a implementação de *MPI*, do *Ohio Supercomputer Centre*, *LAM*; [BDV94] e o compilador *GNU C* da *Free Software Foundation, Inc.*

2. Background

2.1. SISAL

SISAL (*Stream and Iteration in a Single-Assignment Language*) algumas vezes chamada de linguagem de associação única, um caso especial de linguagem aplicativa, apresenta como importante característica a ausência de efeito colateral em razão da inexistência de estados globais.

Um programa *SISAL* é composto por uma coleção de partes separadas chamadas unidades de compilação. Cada unidade de compilação define um número arbitrário de funções e a natureza da interface para a interação com as outras unidades.

Uma função em *SISAL* computa um ou mais valores a partir dos parâmetros de entrada. Os tipos de *SISAL* incluem os tipos escalares básicos `boolean`, `integer`, `real`, `double_real`, `null` e `character`. Os valores de dados estruturados podem ser `record`, `array`, `union` ou `stream`.

A linguagem *SISAL* possui paralelismo em todos os níveis, desde as expressões mais simples até as estruturas mais complexas. Em *SISAL* temos como fontes de paralelismo os *loops forall*, os *loops* iterativos, as funções, as expressões múltiplas e as *streams* [Ske91]. A forma mais explícita de paralelismo é o *loop forall*. Ela é muito similar às estruturas de iteração, mas não é uma, pois inicia várias instâncias de corpos de *loop* independentes. Nesta estrutura cada instância do corpo é independente das outras, podendo ser executada em paralelo.

A expressão *forall* de *SISAL* é escrita como uma expressão que possui um gerador de valores que pode ser usado dentro do corpo do *loop*. A coleta dos dados produzidos pelas instâncias dos corpos do *forall* fica a cargo da parte de retorno de resultados do *forall*. De uma forma simplificada poderíamos expressar um *forall* como:

```
for nome in minimo,maximo                    (gerador de índices)
<instancias do forall>                       (corpo do forall)
returns expressão de gather/                 retorno
        expressão de reducao
end for
```

2.2. IF1 (Intermediate Form)

O sucesso de um compilador para linguagens funcionais depende grandemente do sucesso com que são realizadas as otimizações sobre os programas. Para auxiliar estas otimizações os compiladores *SISAL* fazem uso de uma representação em forma de grafos acíclicos chamada *IF1*.

Esta linguagem de grafos acíclicos representa de forma direta a linguagem *SISAL* através de suas estruturas: grafos, nós, arcos e tipos. A representação procede da seguinte maneira: os grafos representam as funções; os nós representam desde as expressões mais simples até as mais complexas como por exemplo um *loop forall*; os arcos representam o fluxo dos dados e os tipos são utilizados com linguagens onde exista a tipagem dos dados, como *SISAL*.

Muito além de ser apenas uma forma de representação de *SISAL*, *IF1*, simplifica em muito a manipulação dos programas, em virtude do número reduzido de estruturas. Programas que realizam operações sobre o código *IF1* tendem a ser relativamente pequenos e de fácil entendimento.

Para a representação, por exemplo, do *loop forall* de *SISAL* temos o nó composto *forall* de *IF1*. Ele também é constituído de três partes, que são chamados de subgrafos: *gerador*, *corpo* e *retorno*; além de uma semântica que define a interação entre os subgrafos e a dependência de dados no nó composto.

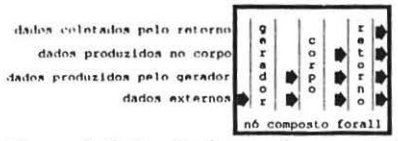

Figura 1. Ilustração de um nó composto *forall*

A figura acima, representa um nó composto *forall*, com os três subgrafos componentes *gerador*, *corpo* e *retorno*; e as dependências de dados entre os subgrafos e entre o nó composto e os subgrafos.

2.3. *MPI*

O *MPI* é uma tentativa para a padronização dos sistemas de passagem de mensagem existentes. Em virtude disto, ele incorpora aspectos de variados sistemas ao invés de adotar como padrão um sistema específico [SOH+95]. Podemos utilizar a biblioteca *MPI* para especificarmos a comunicação entre conjuntos de processos formando um programa concorrente. O paradigma de passagem de mensagem é muito atrativo devido à fácil portabilidade de programas que o utilizam. Ele é compatível tanto com sistemas com memória distribuída como com sistemas de memória compartilhada, sendo que a passagem de mensagem não se tornará obsoleta com a expansão do sistema [MPI94].

Os conceitos elementares do padrão *MPI* são processos e mensagens. A comunicação ponto a ponto e a comunicação coletiva são os conceitos centrais do *MPI* enquanto que grupos de processos, contexto de comunicação, tipos de dados e topologias virtuais são outros conceitos importantes embutidos no padrão *MPI*.

A comunicação ponto a ponto envolve apenas dois processos: um processo enviando uma mensagem e um processo recebendo esta mensagem. Todos os procedimentos de comunicação ponto a ponto do *MPI* têm uma versão bloqueante e uma versão não bloqueante. O modo de uma operação de comunicação ponto a ponto determina o momento em que uma operação *send* é iniciada ou quando ela é completada. Os modos de operação são: *padrão*, *ready*, *síncrono* e *buffered*.

As rotinas de comunicação coletiva fornecem meios para a realização de comunicação coordenada entre grupos de processos [MPI94]. O padrão *MPI* define os seguintes tipos de comunicação coletiva:

- Sincronização *barrier* entre todos os membros do grupo;
- Funções de comunicação global: *broadcast*, *scatter* e *gather* e
- Operações de redução global tais como *sum*, *max*, *min* ou funções definidas pelo usuário.

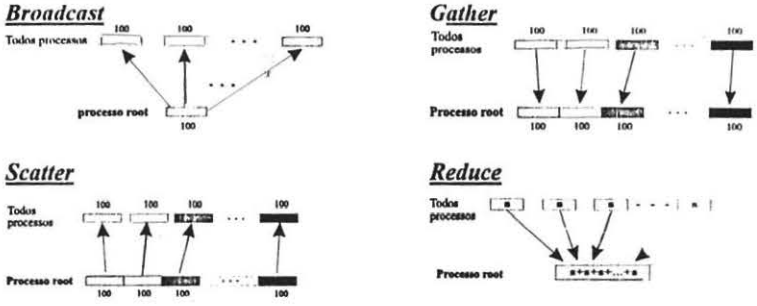

Figura 2. Ilustração das operações de comunicação coletiva

3. Paralelização dos programas e questões de implementação

Após estudos realizados, optou-se pela manipulação de programas *SISAL* representados na forma da linguagem de grafos *IF1*. Isto em virtude das propriedades de *IF1*. Podemos verificar entre outras coisas a simplicidade do código *IF1*, em função do reduzido número de regras de produções definindo sua sintaxe, o que o torna de fácil manipulação. Outro aspecto importante para a escolha de *IF1* foi a relativa independência com uma linguagem específica. Após o *front-end* [Can92a], uma das fases de compilação do *OSC*, foi obtido o código *IF1* equivalente de programas escritos em *SISAL*.

Para realizarmos chamadas das funções da biblioteca *MPI* escrita em *C* nos programas *SISAL* fizemos uso do suporte para interface com linguagens estrangeiras oferecidas pelo compilador *OSC*. Além da utilização deste suporte, foi criada uma biblioteca em *C*, para que fosse possível contornar o problema de incompatibilidade dos tipos entre *SISAL* e *MPI* e resolvido o problema de reordenação de nós realizados pelo compilador *SISAL*. A preocupação com reordenação dos nós das chamadas de funções *MPI* está no fato de que ela poderia gerar resultados inconsistentes ou mesmo *deadlock*, pois a ordem das funções de comunicação coletiva devem corresponder em todos os processos [SOH+95].

Quanto a questão da paralelização de programas devemos resolver os seguintes problemas: a escolha da fonte de paralelismo, escolha de um método de particionamento e o modelo de paralelização das tarefas a ser utilizado.

Escolhemos a expressão *forall* como nossa fonte de paralelismo, motivados pelo fato desta representar a forma mais explícita de paralelismo de *SISAL*, onde cada uma das instâncias do corpo pode ser executadas em paralelo e sendo o número de iterações conhecido no momento da entrada na estrutura.

O método de particionamento escolhido foi o *slice* dos *loops forall* [SC90] apenas no seu nível mais externo. O método *slice* consiste na divisão de todas as

instâncias do *forall* em grupos de instâncias consecutivas, de modo a formar *foralls* de menor número de iterações (divisão dos índices do *forall*). A forma para o cálculo do número de fatias do *forall* foi baseada no número de processos executando [SC90] em *MPI_COMM_WORLD*. Os índices foram calculados da seguinte maneira:

- Nº de elementos em cada fatia = total de instâncias do *forall* original / nº de processos
- Novo limite inferior = Nº de elementos em cada fatia * i + Limite inferior do *forall* original
  onde i é o *rank* do processo
- Novo limite superior = Novo limite inferior + Nº de elementos em cada fatia - 1

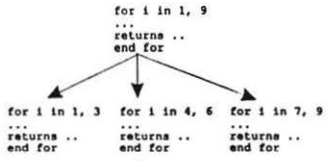

Figura 3 *Slice* de um *forall* com 9 instâncias sendo dividido em 3 *forall* de 3 instâncias.

Escolheu-se para a implementação do paralelismo o modelo *SPMD* (*Single Program Multiple Data*) [AG94, Pac95] um caso especial de *MIMD*, no qual todos os processadores executam o mesmo programa, se bem que em qualquer momento a instrução, e é claro os dados, podem ser diferentes para cada processador em virtude de desvios dependentes dos dados. Este estilo de programação traz uma série de benefícios [FC95].

Em *SPMD*, o número de processos ou tarefas é decidido antes do início da execução, assim evitando a sobrecarga do sistema operacional associado com o estilo *fork-join* de criação de processos durante a execução. O *SPMD* pode ser usado tanto em sistemas de memória compartilhada como em sistemas de memória distribuída (passagem de mensagem).

Além do estilo de programação *SPMD*, escolheu-se o modelo de processos *mestre/escravo* com a distribuição dos *slices* em apenas um nível.

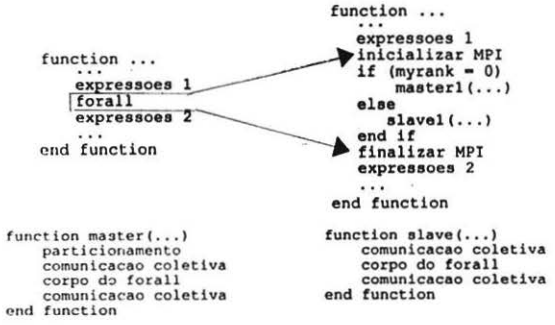

Figura 4 Particionamento *slice* criando processos *mestre* e *escravo* no modelo *SPMD*

Podemos observar na Figura 4 acima que no início e no final englobando as alterações são realizadas operações para a inicialização e finalização de *MPI* como é

exigido [MPI94, SOH+95]. Cada expressão *forall* foi substituída por uma expressão condicional com chamada para função *mestre* em uma condição e a função *escravo* na outra. Tanto a função *mestre* como a *escravo* possuem, além de uma fatia do *forall* original, rotinas de comunicação para a recepção e transmissão de dados.

Após a expressão *forall* podemos observar novamente expressões para comunicação coletiva necessárias para reagrupar os resultados das várias fatias do *forall*. As operações a serem realizadas são *gather* ou *reduce* de acordo com a parte de retorno da expressão *forall*.

3.1 Algoritmo

Tomadas todos as decisões de como seria realizado o particionamento, foi desenvolvido um algoritmo que percorre o programa na forma de grafo até o final, realizando as alterações necessárias. No primeiro passo, o algoritmo procura pelos limites da primeira função que ainda não tenha percorrido. Em seguida, para cada função procura pelo primeiro *forall* não percorrido, marcando início e final caso encontre algum. Se encontrou um *forall*, realiza as alterações necessárias para a transformação em *SMPD*. Abaixo o algoritmo para as alterações necessárias nos programas *IF1*.

```
enquanto não percorrer todo o grafo faça
    marcar limites da função (início e o final da função)
    enquanto não encontrar a marca de final da função faça
       procurar por nós forall
       se encontrou nó forall
           marcar o inicio do nó forall
           marcar o final do nó forall
           inserir chamada para a função sisal_mpi_comm_size;
           inserir a expressão if para SMPD
                inserir chamada para a função sisal_mpi_comm_rank
                inserir expressão:
                     se número do rank é igual ao rank do mestre
                          chamada para a função mestre
                     senão
                          chamada para a função escravo
                     fim se
           fim inserir inserir a expressão if para SMPD
           iniciar declaração para a função mestre
                calcular o limite inferior e superior do nó forall
                chamada de sisal_mpi_scatter para o limite inferior
                chamada de sisal_mpi_scatter para o limite superior
                se existe nomes de valores dentro do forall
                    distribui-los com chamadas de sisal_mpi_bcast
                fim se
                copiar o forall da função original
                alterar os nomes dos limites do forall
                acertar as entradas do forall
                remover o forall original
                inserir chamadas para sisal_mpi_gather/reduce
           fim da declaração do mestre;
           iniciar a declaração para a função escravo
                chamada de sisal_mpi_scatter para o limite inferior
                chamada de sisal_mpi_scatter para o limite superior
                se existe valores não constantes dentro do forall
```

```
                    recebê-los com chamadas de sisal_mpi_bcast
                fim_se
                copiar o nó forall do mestre
                inserir chamadas para sisal_mpi_gather/reduce
            fim da declaração do escravo
        fim_se
    fim_enquanto
fim_enquanto
```

4. Resultados

O método utilizado para verificar a correção dos programas com nossas modificações foi a comparação dos resultados obtidos com um programa sem as alterações. Os resultados aqui apresentados não provam que o programa desenvolvido em nosso trabalho esteja completamente correto, pois não era objetivo de nosso trabalho a construção de uma parte completa de um compilador. Devemos lembrar que a proposta de nosso trabalho era provar a possibilidade de execução de programas *SISAL* em sistemas *MPI*.

1. Programa *SISAL* que utiliza um comando de redução

```
define main

function main(returns integer)
   for K in 1, 10
       returns value of sum K
   end for
end function
```

Resultado obtido do programa original

```
SUN SISAL 1.2 V12_9_1
55
```

Resultado obtido do programa particionado utilizando as bibliotecas *MPI*

```
2325 reduce running on n0 (o)
SUN SISAL 1.2 V12_9_1
12782 reduce running on n1
55
```

2. Programa que utiliza o comando de *gather*

Código *SISAL*

```
define main

function main(returns array[integer], array[integer])
   let
        z:=array_fill(0,9,0);
   in
        for i in 0,9
        returns array of i
                array of 47 * i
        end for
   end let
end function
```

Resultado obtido do programa original

```
SUN SISAL 1.2 V12_9_1
[ 0,9: # DRC=1 PRC=1
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
]
[ 0,9: # DRC=1 PRC=1
 0
 47
 94
 141
 188
 235
 282
 329
 376
 423
]
```

Resultado obtido do programa utilizando as bibliotecas MPI

```
2237 teste running on n0 (o)
SUN SISAL 1.2 V12_9_1
12491 teste running on n1
floyd{raul}66: [ 0,9: # DRC=1 PRC=1
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
]
[ 0,9: # DRC=1 PRC=1
 0
 47
 94
 141
 188
 235
 282
 329
 376
 423
]
```

5. Conclusões

O objetivo de nosso trabalho foi realizar um estudo das alterações necessárias em *SISAL*, para que pudéssemos fazer uso da biblioteca do padrão MPI. Como um dos resultados deste estudo optou-se pela manipulação do código *IF1* ao invés de *SISAL* e apresentamos propostas de alterações e adaptações nos programas fontes originais. Para a validação da nossa proposta implementamos um programa para inserir as transformações nos programas fontes e para verificar a validade das alterações realizamos execuções dos programas com as alterações. Abaixo as principais conclusões deste trabalho.

SISAL/IF1

A princípio estava definido que o código fonte a ser manipulado era *SISAL*, mas após estudos realizados constatamos que seria mais vantajoso a manipulação do código *IF1*. Além disso, *IF1* é uma representação direta de *SISAL* o que não traria problemas para a proposta de nosso trabalho. Constatamos que a simplicidade de *IF1*, composto de grafos, nós, arcos e tipos, facilitaria bastante o nosso trabalho na manipulação do código fonte. Um dos motivos para tal constatação foi que o número reduzido de estruturas de *IF1* permitiu que um pequeno número de regras de produção definisse a linguagem. Além disso, verificamos que a manipulação de um código fonte na forma de grafos permite que análises de dependência sejam realizadas simplesmente pela procura por arcos comuns entre os nós (operações). Como resultado destas propriedade *IF1* apresenta uma semântica clara e bem definida..

MPI

Do padrão *MPI* utilizamos principalmente as rotinas de comunicação coletiva na transmissão das mensagens além do suporte para a manipulação das mensagens e processos, o que reduz consideravelmente a abrangência dos problemas a serem resolvidos para a utilização de um sistema de passagem de mensagem. Caso este suporte não estivesse disponível necessitaríamos dispender um tempo razoável para a definição das rotinas necessárias. Além da redução dos esforços, ainda nos beneficiamos da portabilidade resultante da padronização.

Propostas de alteração e Implementação

O modelo de execução de processos escolhido (*mestre/escravo*) foi uma grata surpresa pois além de sua simplicidade, adaptou-se perfeitamente ao modelo de particionamento escolhido, *slice*, tornando clara as alterações necessárias nos programas *SISAL* originais. A estrutura escolhida para ser particionada, *forall*, colaborou na clareza para a transformação de um programa *SISAL* para o *SPMD*, em virtude de ser inerentemente paralela, podendo então ser particionada diretamente.

Podemos verificar pelo algoritmo e pela implementação que a utilização de um código fonte de um programa na forma de grafo realmente reduz o tamanho do programa que se é escrito para a manipulação do grafo.

A biblioteca para a ligação entre *SISAL* e *C* consegue contornar o problema em relação à incompatibilidade entre os tipos de *SISAL* e de *MPI*. Apesar de contornar o problemas dos tipos de *SISAL* e *MPI*, esta biblioteca não resolve completamente o problema pois, ela deveria ser criada dinamicamente de modo a suprir todas as necessidade momentâneas e não ser fixa como foi definida. Esta necessidade se deve ao fato que sendo fixa, não poderá suportar qualquer dimensão de *array*.

A limitação não impediu no entanto que pudéssemos testar as alterações propostas, ainda que com um número limitado de tipos. Podemos fazer esta afirmação pois na distribuição dos dados aos processos, distribuição dos índices do *forall* e reagrupamemto dos resultados não estão associados ao tipo do dado mas à estrutura apresentada pelo *forall*.

Bibliografia

[AG94] Almasi, G. S., and Gottlieb, A., Highly Parallel Computing, Benjamin/ Cummings, 1994.

[BDV94] Burns, G., Daoud R., and Vaigl, J., LAM: An Open Cluster Enviroment for MPI, Ohio Supercomputer Center, 1994,
http://www.epm.ornl.gov/~walker/mpi/papers/lam-mpi.ps.Z.

[Can92a] Cann, D. C., The optimizing SISAL compiler: Version 12.0, Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April, 1992.

[Can92b] Cann, D. C., SISAL 1.2: A Brief Introduction and Tutorial, UCRL-MA-110620, Lawrence Livermore National Laboratory, May, 1992.

[FA95] Freeh, V. W., and Andrews, G. R., fsc: A Sisal Compiler for Both Distributed- and Shared-Memory Machines, High Performance Functional Computing, 164-172, April, 1995.

[FC90] Feo, J. T., and Cann, D. C., A Report on the Sisal Language Project, Journal of Parallel and Distributed Computing 10, 349-366, December, 1990.

[FC95] Foisy, C., and Chailloux, E., Caml Fligth: a Portable SPMD Extension of ML for Distributed Memory Multiprocessor, High Performance Functional Computing, 83-96, April, 1995.

[HB93] Haines, M., and Böhm, W., Task Management, Virtual Shared Memory, and Multithreading in a Distributed Memory Implementation of Sisal, PARLE 93 - Parallel Architecture and Language Europe - Lecture Notes in Computer Science - Springer Verlag, 12-23, June, 1993.

[MPI94] Message Passing Forum, MPI: A Message Passing Interface Standard, International Journal of Supercomputer Applications, vol. 8, nos 3/4, 1994

[SC90] Sarkar, V., and Cann, D., POSC - a Partitioning and Optimizing SISAL Compiler, Proc. of the ACM International Conference on Supercomputing, 148-163, June, 1990.

[Ske91] Skedzielewski, S. K., Parallel and Functional Languages and Compilers, ACM PRESS, 1991

[SG85] Skedzielewski, S., and Glauert, J., IF1 An Intermediate Form for Applicative Languages, Manual M-170, Lawrence Livermore National Laboratory, Livermore, California, January, 1985

[SW85] Skedzielewski, S. K., and Welcome, M. L., Data Flow Graph Optimization in IF1, Functional Programming Language and Computer Architecture - Lecture Notes in Computer Science, 17-34, 1985

[SOH+95] Snir, M., Otto, S., Huss-Lederman, S., and Dongarra, J., MPI: The Complete Reference, The MIT Press, 1995
ou http://www.netlib.org/utk/papers/mpi-book/mpi-books.ps