# Geração de Números Perfeitos

**Rafael Tenfen**[1]

[1] Universidade do Estado de Santa Catarina (UDESC)

rafaeltenfen.rt@gmail.com

***Resumo.*** *Números perfeitos são números inteiros positivos em que a soma de seus divisores positivos excluindo o próprio número, é igual ao seu valor. Há várias formas utilizadas para verificar se um número é perfeito, como por exemplo encontrar um a um os divisores dos antecessores, ou utilizando a fórmula de Euclid. Nesse trabalho, será abordado a questão de geração de números perfeitos de modo sequencial, e como realizar a refatoração para o algoritmo utilizando técnicas de programação paralela avançada.*

## 1. Introdução

Um número natural é dito perfeito se é igual a soma de seus divisores, excluindo o próprio número. Assim, $n$ é perfeito se e somente se

$$\sigma(n) = \sum_{d|n} d = 2n$$

Matemáticos tem estudado esses números por aproximadamente dois milênios, no entanto, o mistério de suas propriedades e sua existência mantém não solucionado, levantando algumas questões. Há infinitos números perfeitos? Existe algum número perfeito ímpar? Todas essas questões, resistiram ao teste do tempo e permanecem em aberto [Holdener 2002].

Utilizando a técnica de *Mersenne prime* em que $(2^p - 1)$ talvez seja primo. Um número primo $p$ é um número inteiro positivo que contém somente dois positivos divisores, 1 e o próprio número $p$ [Crandall and Pomerance 2006]. Até o momento, foram descobertos cinquenta e um números perfeitos, o último por sinal em sete de dezembro de dois mil e dezenove, pode ser representado da seguinte forma: $2^{82.589.932}(2^{82.589.933} - 1)$, nenhum dos números perfeitos encontrados é impar [Mersenne Research 2019]. Contudo, ninguém pode garantir que não existam números perfeitos ímpares e nem a sua finitude.

Euclid foi bem sucedido em utilizar números primos e *Mersenne primes* para estabelecer o **Teorema A**. Utilizando o teorema de Euclid, foi possível encontrar os quatro primeiros números perfeitos - 6, 28, 496, 8128. Logo após, Euler em 1747 com o **Teorema B**. mostrou que todo número perfeito que é par pode ser representado pela aplicação da regra de Euclid [Pollack and Shevelev 2012].

**Teorema A** (Euclid). Se $p$ é um número primo e $2^p - 1$ também é um número primo. Então $n = 2^{p-1}(2^p - 1)$ é um número perfeito.

**Teorema B** (Euler). Todo número perfeito par têm forma $2^{p-1}(2^p - 1)$, onde $p$ e $2^p - 1$ são primos.

## 2. Problema

A implementação de geração de números perfeitos, assim como inúmeros outros problemas de programação pode ser feita de modo sequencial, mas preferencialmente deve ser utilizado técnicas de programação paralela para o seu desenvolvimento, utilizando essas técnicas espera-se uma melhora no desempenho em relação ao mesmo problema escrito de modo sequencial.

Para aplicar a paralelização de geração de números perfeitos, será utilizado uma arquitetura que pode executar, de forma paralela com o OpenMP e também distribuída com o MPI, utilizando fluxos de instruções independentes, sendo que cada fluxo tem seu próprio fluxo de dados, chamada de MIMD (*multiple instruction, multiple data*).

O OpenMP é definido como uma API (*Application Programming Interface*) para programação paralela de memória compartilhada. Ele foi projetado para que toda *thread* existente na solução tenha possibilidade de acessar toda a memória disponível [Bianchini et al. ].

Já o MPI (*Message Passing Interface*), foi originalmente escrito para escrever aplicações e bibliotecas que desejam utilizar ambientes de memória distribuída. A vantagem principal de estabelecer uma interface de comunicação de mensagens para ambientes de memória distribuída é a portabilidade e facilidade do uso [Walker and Dongarra 1996].

## 3. Proposta de Implementação

Nessa seção, será abordada a metodologia **PCAM** que define quatro etapas do processo de paralelização para o problema de geração de números perfeitos. Essa separação de 4 etapas, apresentada na Figura 1, reflete uma forma que induz a pensar nos aspectos fundamentais de paralelização, permitindo a quebra de um problema em pedaços pequenos que possam ser justamente calculados ao mesmo tempo por diferentes núcleos de processamento [Schnorr and Nesi 2019].

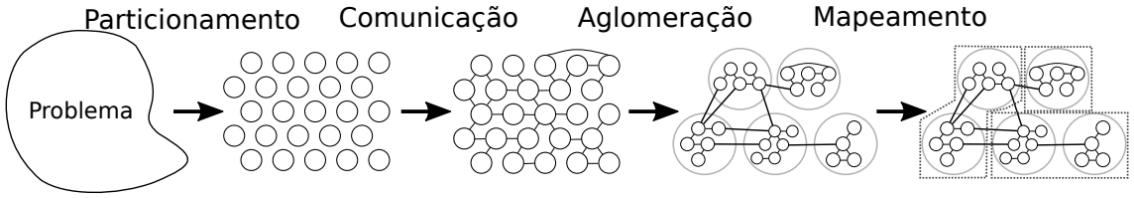


**Figure 1. Modelo metodológico PCAM com suas quatro fases: particionamento, para quebrar o problema em pedaços menores; comunicação, para definir quais pedaços devem comunicar-se entre si; aglomeração, para agrupar pedaços com critérios de localidade; e mapeamento, para atribuir a unidades computacionais os grupos que devem ser processados [Schnorr and Nesi 2019]**

Utilizando programação sequencial foram implementadas duas formas de geração de números perfeitos. A primeira forma denominada de *"brute force"*, utiliza-se apenas da técnica de dividir números antecessores um a um para verificar seus possíveis divisores, realizar a soma desses divisores e verificar se o resultado da soma é igual ao número, definindo assim o número como perfeito.

A segunda forma denominada de *"Euclid"*, utiliza-se da fórmula de Euclid, apresentada no **Teorema A** da seção 1 que verifica se um número é perfeito utilizando a verificação de números primos. Dessa forma, é utilizado a geração de números primos, aplicados a fórmula de Euclid para geração dos números perfeitos.

### 3.1. Particionamento

O particionamento é uma abordagem significativa para uma eficiente atribuição de executável a *tasks*, a fim de utilizar computação paralela no processamento de instruções. O particionamento é uma divisão de partes independentes para resolvê-las em paralelo. Desse modo, pequenas tarefas devem ser definidas, para serem processadas de forma otimizada e evitem dados e cálculos duplicados. Quanto menores as partições ficam, maior o potencial do paralelismo [Hoettger et al. 2013].

Na paralelização do código sequencial criado, será utilizado um conjunto de particionamento de dados, também chamadas de decomposição de domínio. A fim de paralelizar a geração de números perfeitos, o particionamento ocorre que cada *thread* deve receber a mesma quantidade de números para realizar a verificação de números perfeitos na forma de *"brute force"*, e em caso de sobra, ou seja, os lotes não possam ser divididos igualitariamente, a *thread* principal deve fazer a verificação dos números restantes.

### 3.2. Comunicação

Em um programa paralelo, é comum que as tarefas necessitem trocar informações para realizar suas operações. Em **PCAM**, a fase de comunicação envolve justamente o projeto destas atividades de troca de dados. Nos cenários onde inexiste a necessidade de comunicação [Schnorr and Nesi 2019].

Para o problema abordado, será utilizado comunicação local em que a operação de computação de uma determinada tarefa (um ponto no domínio de problema discretizado) necessita dados de um pequeno número de tarefas (pontos) vizinhas. Nesse contexto, as tarefas tem vizinhos claramente definidos e imutáveis em função do particionamento estabelecido anteriormente, dessa forma, as comunicações são frequentemente fixas, ditas estruturadas. A discretização a ser realizada de maneira estática é fixa desde a concepção na fase de particionamento do projeto até a execução do código.

### 3.3. Aglomeração

A fase de aglomeração tem por objetivo tornar o algoritmo abstrato das fases precedentes em algo mais realista, de acordo com os limites impostos pela configuração da plataforma de execução alvo. O objetivo principal é obter um programa eficiente, além de que nessa fase, deve ser definido a granularidade não fixa das tarefas [Schnorr and Nesi 2019].

A aglomeração na geração de números perfeitos, acontece ao finalizar a soma dos possíveis divisores a fim de verificar o perfeccionismo do número e adicionando-os a um vetor, a fim de agrupar todos os números perfeitos gerados.

### 3.4. Mapeamento

O quarto e ultimo estágio da metodologia **PCAM**, chamado de mapeamento, consiste em definir onde cada tarefa será executada. Os requisitos fundamentais na atividade explícita de mapeamento envolvem colocar tarefas concorrentes em unidades de processamento

diferentes, de forma que tais tarefas sejam de fato executadas em paralelo, além de alocar tarefas que se comunicam frequentemente em locais próximos da topologia de interconexão, tanto física quanto lógica [Schnorr and Nesi 2019].

No problema apresentado nesse trabalho, com base nos demais estágios do **PCAM** em que o particionamento utilizará um conjunto de particionamento de dados e também particionamento das operações, a comunicação definida é estática, estruturada e local. Então, o mapeamento a ser utilizado é de uma *thread* por processador.

## 4. Resultados

Nessa seção, será apresentado os resultados obtidos através da implementação de duas abordagens com o OpenMP e MPI, além da combinação das duas abordagens, e também a descrição do hardware das máquinas utilizadas.

Todos os resultados utilizaram a forma *brute force*, pois o método de Euclid utiliza uma fórmula para verificação de números perfeitos, então a sua execução sequencial é mais rápida que fazer a distribuição e comunicação entre as *threads*, por esse motivo a sua execução foi excluída dos resultados.

### 4.1. OpenMP

A execução com OpenMP utilizou os seguintes alcances de números para verificação de números perfeitos: 0-100000, 0-300000 e 0-600000. Além das seguintes quantidades de *threads*: 1, 2, 4, 6 e 8, para cada alcance, e por fim foram testados os seguintes *schedules*: *static*, *dynamic* e *guided* para cada alcance e número de *threads* definidas.

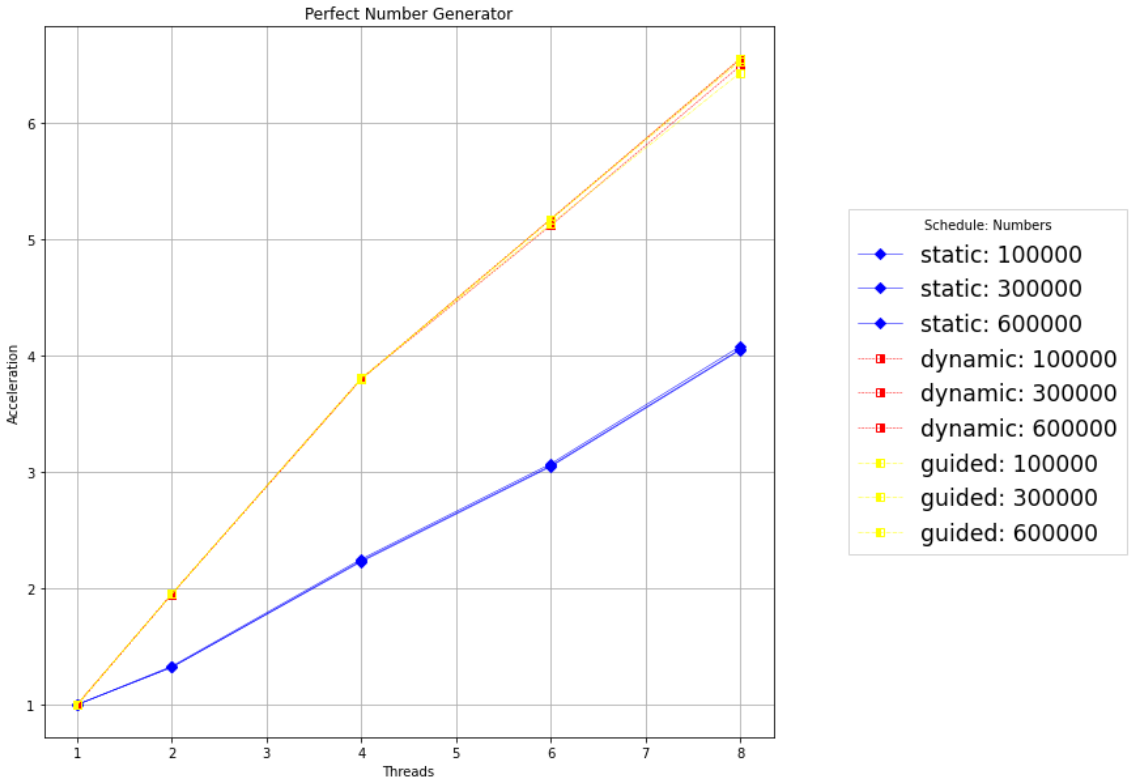


**Figure 2. Todas as execuções do OpenMP agrupadas em um único gráfico.**

Como é possível observar na Figura 2, os *schedules dynamic* e *guided* tiveram os melhores resultados. As linhas ficaram sobrepostas devido à pouca diferença entre os resultados testados. Para coleta de dados, foi criado um *script* em *bash* para realizar 2 execuções e fazer a média do tempo gasto. O processamento dos dados e o gráfico foram desenvolvidos utilizando *python*.

#### 4.1.1. Hardware

A máquina utilizada para a execução dos testes foi a ens5 tem 8 núcleos do modelo (Intel(R) Xeon(R) CPU E3-1230 v6 @ 3.50GHz) e 48GB de memória.

### 4.2. MPI

A execução com o MPI utilizou os seguintes alcances de números para verificação de números perfeitos: 0-100000, 0-300000 e 0-600000. Além das seguintes quantidades de *threads*: 1, 2, 4, 8, 16 e 24 para cada alcance e número de *threads* definidas. Para a coleta de dados dos gráficos foi realizado apenas uma execução.

Conforme apresentado na Figura 3, até 16 *threads* obteve-se uma boa aceleração, pois foram executados apenas em 2 máquinas, já para alcançar 24 *threads*, foi necessário 4 máquinas, aumentando a necessidade de comunicação entre as máquinas através da rede e obtendo uma aceleração pior do que com 16 *threads* e apenas 2 máquinas.

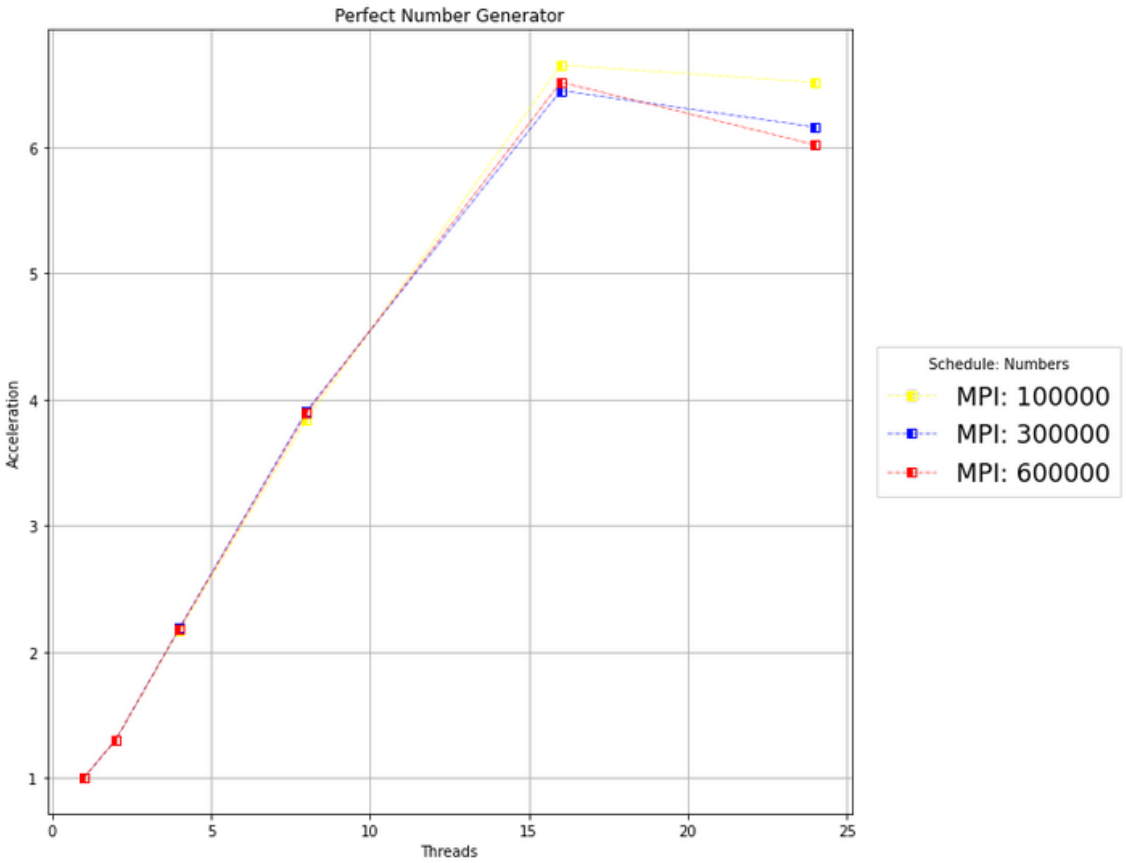


**Figure 3. Todas as execuções do MPI agrupadas em um único gráfico.**

### 4.2.1. Hardware

O hardware utilizado no MPI teve diferenças de acordo com o número de *threads*, o hardware das máquinas está definido na Tabela 1. Para os testes com as *threads* 1, 2, 4, 8 foi utilizado a máquina ens5, já para 16 *threads* foi utilizado as máquinas ens5 e ens4, para o teste com 24 *threads* foram utilizadas todas as máquinas apresentadas na Tabela 1: ens1, ens2, ens4 e ens5.

| Nome | CPU's | Model | Memória |
|---|---|---|---|
| ens1 | 4 | AMD Phenom(tm) II X4 B93 Processor | 8GB |
| ens2 | 4 | AMD Phenom(tm) II X4 B93 Processor | 8GB |
| ens4 | 8 | Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz | 16GB |
| ens5 | 8 | Intel(R) Xeon(R) CPU E3-1230 v6 @ 3.50GHz | 48GB |

**Table 1. Hardware utilizado para execução de testes**

## 4.3. OpenMP e MPI

Ao utilizar as duas bibliotecas OpenMP e MPI, tiveram algumas complicações nas execuções para fazer com que o OpenMP utilize certa quantidade de processos, ou até o máximo de processos definidos. Entretanto, após as complicações resolvidas, foi possível executar os testes com os seguintes alcances de números perfeitos: : 0-100000, 0-300000 e 0-600000. Além das seguintes quantidades de *threads*: 1, 8, 16 e 24 para cada alcance, e por fim foram testados os seguintes *schedules*: *static*, *dynamic* e *guided* para cada alcance e número de *threads* definidas.

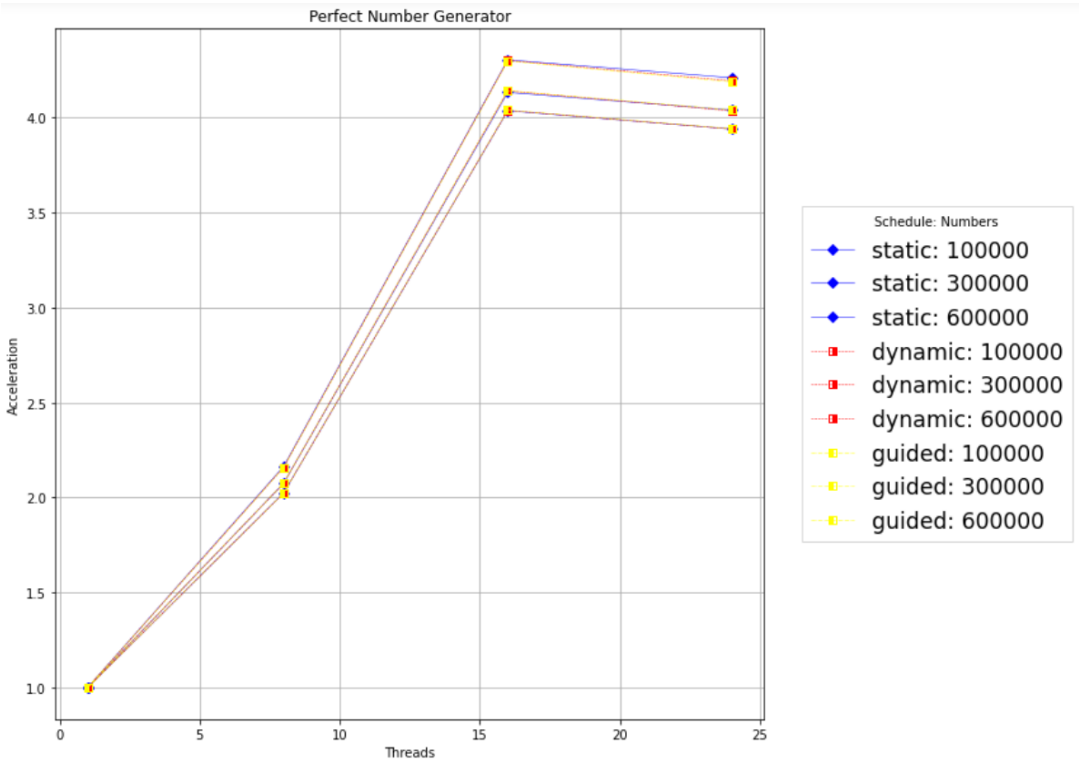


**Figure 4. Execuções das bibliotecas OpenMP e MPI agrupadas.**

Conforme apresentado na Figura 4, a execução teve um destino parecido com a Figura 3 pelo mesmo motivo apresentado na seção 4.2, em que mais máquinas precisaram ser utilizadas para alcançar 24 *threads*, dificultando a comunicação entre os processos. Os resultados apresentados no gráfico da Figura 4 foram processados com apenas uma rodada de teste.

### 4.3.1. Hardware

O hardware utilizado para as execuções do conjunto de bibliotecas OpenMP e MPI é o mesmo apresentado na Tabela 1. Porém, as execuções foram diferentes para cada número de *threads*, para apenas uma *thread* foi utilizado apenas a máquina ens5 com um processador, para 8 *threads* foram utilizadas todas as quatro máquinas, cada uma com 2 processadores, somando 8 processadores com apenas uma *thread* cada, de acordo com o mapeamento definido na seção 3.4. Já para 16 *threads*, todas as 4 máquinas foram utilizadas com 4 *threads* cada, totalizando os 16 processadores. Para as 24 *threads* foi utilizado o máximo de processadores de cada máquina, totalizando as 24 *threads* para 24 processadores.

## 5. Conclusão

As fases precedentes da metodologia **PCAM** permitem o particionamento e a definição das comunicações necessárias para a resolução paralela de um problema. O resultado destas fases é um algoritmo abstrato que contém potencialmente muitas tarefas, tendo em vista que o objetivo é identificar a menor operação possível que possa ser executada concorrentemente com as demais [Schnorr and Nesi 2019].

Portanto, foi aplicado as técnicas da metodologia **PCAM** apresentadas na seção 3 que teve como base o livro *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering* [Foster 1995], com o auxílio do OpenMP e MPI para aprimorar o desempenho da geração de números perfeitos utilizando técnicas de programação paralela avançada.

Enfim, obteve-se um grande entendimento do assunto de programação paralela avançada após realizar o paralelismo de um simples problema como a geração de números perfeitos utilizando memória compartilhada e também memória distribuída.

## References


Bianchini, C. P., Vilabôas, F. G., and Castro, L. N. Paralelismo de tarefas utilizando openmp 4.5.

Crandall, R. and Pomerance, C. B. (2006). *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media.

Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA.

Hoettger, R., Igel, B., and Kamsties, E. (2013). A novel partitioning and tracing approach for distributed systems based on vector clocks. In *2013 IEEE 7th International Con-*

*ference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*, volume 2, pages 670–675. IEEE.

Holdener, J. A. (2002). A theorem of touchard on the form of odd perfect numbers. *The American mathematical monthly*, 109(7):661–663.

Mersenne Research, I. (2019). List of known mersenne prime numbers - primenet.

Pollack, P. and Shevelev, V. (2012). On perfect and near-perfect numbers. *Journal of Number Theory*, 132(12):3037–3046.

Schnorr, L. and Nesi, L. (2019). *Projetando e Construindo Programas Paralelos. Anais da Escola Regional de Alto Desempenho da Região Sul*, volume v. 19ed.

Walker, D. W. and Dongarra, J. J. (1996). Mpi: a standard message passing interface. *Supercomputer*, 12:56–68.