



Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Faculdade de Engenharia

Bruno Ramos Carrilho

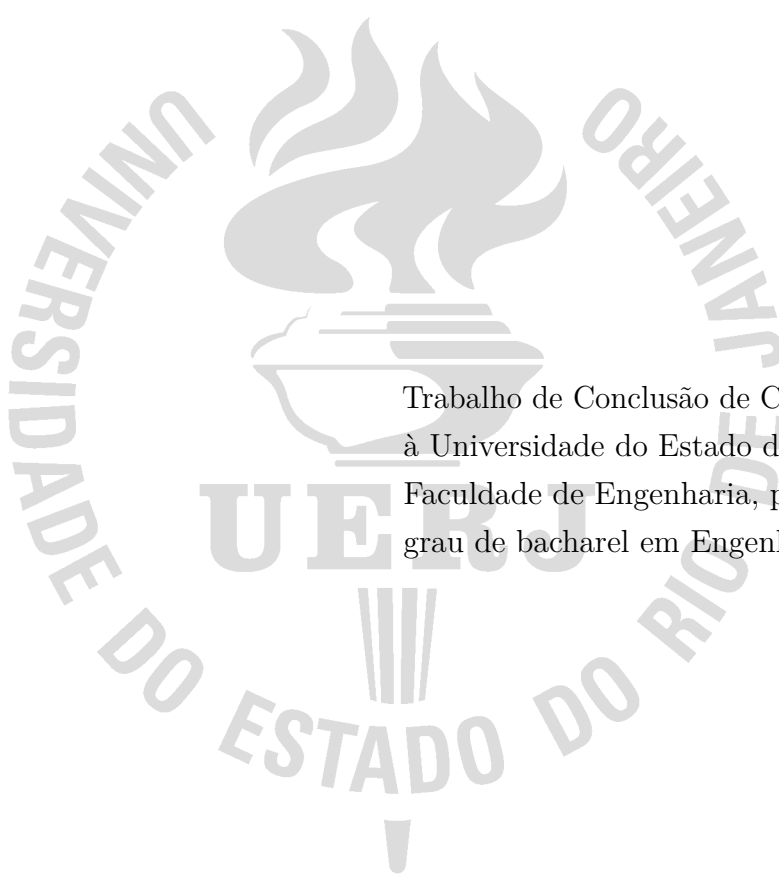
**Avaliação comparativa de troca de mensagens de alto
desempenho em C e Julia**

Rio de Janeiro

2020

Bruno Ramos Carrilho

**Avaliação comparativa de troca de mensagens de alto desempenho em C e
Julia**



Trabalho de Conclusão de Curso apresentado
à Universidade do Estado do Rio de Janeiro,
Faculdade de Engenharia, para obtenção do
grau de bacharel em Engenharia Elétrica.

Prof.^a Dra. Cristiana Barbosa Bentes

Rio de Janeiro

2020

Ficha elaborada pelo autor através do
Sistema para Geração Automática de Ficha Catalográfica da Rede Sirius - UERJ

C317 Carrilho, Bruno Ramos
Avaliação comparativa de troca de mensagens de
alto desempenho em C e Julia / Bruno Ramos Carrilho.
- 2020.
113 f.

Orientadora: Cristiana Barbosa Bentes
Trabalho de Conclusão de Curso apresentado à
Universidade do Estado do Rio de Janeiro, Faculdade
de Engenharia, para obtenção do grau de bacharel em
Engenharia Elétrica.

1. Julia - Monografias. 2. Troca de Mensagens -
Monografias. 3. computação distribuída - Monografias.
I. Bentes, Cristiana Barbosa. II. Universidade do
Estado do Rio de Janeiro. Faculdade de Engenharia.
III. Título.

CDU 621.3

Bruno Ramos Carrilho

Avaliação comparativa de troca de mensagens de alto desempenho em C e Julia

Trabalho de Conclusão de Curso apresentado à Universidade do Estado do Rio de Janeiro, Faculdade de Engenharia, para obtenção do grau de bacharel em Engenharia Elétrica.

Trabalho aprovado. Rio de Janeiro, 4 de junho de 2020.

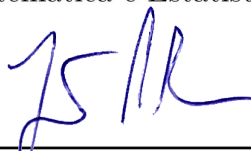
Banca examinadora:



Prof.^a Dra. Cristiana Barbosa Bentes (Orientadora)
Faculdade de Engenharia - UERJ



Prof.^a Dra. Maria Clicia Stelling de Castro
Instituto de Matemática e Estatística - UERJ



Prof. Dr. João Araújo Ribeiro
Faculdade de Engenharia - UERJ

Rio de Janeiro
2020

Resumo

Carrilho, B. R. Avaliação comparativa de troca de mensagens de alto desempenho em C e Julia. 2020. 113 f. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Elétrica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2020.

Este trabalho faz uma comparação das estruturas para processamento paralelo oferecidas pelas linguagens C com a biblioteca MPI e Julia. A ideia é explicitar a viabilidade da linguagem Julia para programação paralela em ambientes de memória distribuída. São comparadas diversas estruturas de troca de mensagens de Julia com a biblioteca MPI em C. O foco da comparação é o de avaliar a simplicidade e clareza de código, o desempenho e os pacotes e bibliotecas disponíveis. Para atingir este objetivo produziu-se programas de teste que exploram as estruturas de ambas as linguagens. Estes programas tiveram seus tempos de execução medidos e seus códigos analisados. Os resultados da comparação entre os tempos de execução medidos dos programas de teste possuem desempenho comparável em Julia em 2 dos seus 5 programas estudados com os programas de C. A linguagem Julia quando comparada a C acerca da sua simplicidade e clareza de código, se mostrou mais concisa. Os pacotes disponíveis em Julia são mais modernos porém mais instáveis do que aqueles em C. Conclui-se que Julia é uma alternativa viável para C quando o objetivo é uma linguagem com alto desempenho em processamento paralelo com troca de mensagens por, não só oferecer desempenho similar, como também uma linguagem mais sucinta e moderna.

Palavras-chave: Julia. C. computação distribuída.

Abstract

Carrilho, B. R. Comparative evaluation of high performance message passing in C and Julia. 2020. 113 f. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Elétrica) – Faculdade de Engenharia, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2020.

This paper makes a comparison between the parallel processing structures offered by the languages C with the MPI library and Julia. The idea is to explicit the viability of Julia in distributed memory environments of parallel programming. Several message passing structures of Julia are compared with C's MPI library. The focus of the comparison is to analyze the simplicity and clarity of the code, the performance and the available packages and libraries. To achieve this goal, test programs were produced to explore the structures of both languages. These programs had their execution times measured and their codes analyzed. The comparison results between the execution times measured in the test programs yielded comparable performance in Julia in 2 of its 5 analyzed programs with C's programs. The Julia language when compared to C concerning its simplicity and clarity of code, revealed itself to be more concise. The available packages in Julia are more modern but more unstable than those of C. It is concluded that Julia is a viable alternative to C when the objective is a language with high performance in parallel computing with message passing for, not only offering similar performance, but also a more succinct and more modern language.

Keywords: Julia. C. distributed computing.

Lista de ilustrações

Figura 1 – Implementação de <i>@spawnat</i> e <i>@fetchfrom</i> utilizando o REPL.	30
Figura 2 – Linha que declara uso do pacote <i>Distributed</i>	30
Figura 3 – Adicionando 2 ao número de processos.	30
Figura 4 – Declarando a variável <i>number</i> usando a rotina <i>rand</i>	31
Figura 5 – Declarando a função quadrado.	31
Figura 6 – Utilização de <i>@spawnat</i>	31
Figura 7 – Uso de <i>fetch</i>	31
Figura 8 – Uso da rotina <i>fetchfrom</i> nos processos número 2 e 3.	31
Figura 9 – Implementação de <i>@distributed</i> junto de <i>SharedArrays</i> utilizando o REPL.	32
Figura 10 – Declaração dos pacotes utilizados e do número de processos adicionados.	32
Figura 11 – A função quadrado e o vetor de inteiros <i>numbers</i>	33
Figura 12 – <i>Vetorresultado</i> e o tipo <i>SharedArray</i>	33
Figura 13 – A utilização de <i>@distributed</i> gerando um <i>future</i>	33
Figura 14 – Recuperação do <i>future</i> “futuro” e impressão do conteúdo de <i>vetorresultado</i>	33
Figura 15 – Implementação de <i>pmap</i> utilizando o REPL.	34
Figura 16 – Pacote <i>Distributed</i> e adição ao número de processos.	34
Figura 17 – Declaração de quadrado e do vetor <i>numbers</i>	34
Figura 18 – Atribuindo <i>pmap</i> com seus argumentos ao <i>vetorresultado</i>	34
Figura 19 – Estrutura de um código MPI.	40
Figura 20 – Rotina <i>MPI_Send</i>	40
Figura 21 – Rotina <i>MPI_Recv</i>	41
Figura 22 – Rotina <i>MPI_Bcast</i>	41
Figura 23 – Rotina <i>MPI_Reduce</i>	41
Figura 24 – Vetor Soma C MPI - Declarações.	42
Figura 25 – Vetor Soma C MPI - Entrando em <i>Main</i>	42
Figura 26 – Vetor Soma C MPI - Populando <i>Vetor1</i> e <i>Vetor2</i> com Números Aleatórios.	42
Figura 27 – Vetor Soma C MPI - Início do Ambiente de MPI.	43
Figura 28 – Vetor Soma C MPI - Mensagem de Erro.	43
Figura 29 – Vetor Soma C MPI - Lidando com Valores de <i>NUM_P</i> Indivisíveis por <i>Size</i>	43
Figura 30 – Vetor Soma C MPI - Passagem de Dados por MPI usando de <i>MPI_Send</i> e <i>MPI_Recv</i>	44
Figura 31 – Vetor Soma C MPI - <i>Workers</i> Processando os Valores Recebidos e os Retornando ao Processo 0.	44
Figura 32 – Vetor Soma C MPI - Processo 0 Fazendo sua Parte e Recebendo os Valores dos <i>Workers</i>	45

Figura 33 – Vetor Soma C MPI - Fim da Medida do Tempo de Execução.	45
Figura 34 – Vetor Soma C MPI - Fim da Medida do Tempo de Execução.	45
Figura 35 – Vetor Soma C MPI - Fim da Medida do Tempo de Execução.	45

Lista de tabelas

Tabela 1 – Tempos de Execução dos Exemplos Soma de Vetores usando Julia, C e <i>Python</i> Sequenciais. O 'n.p.m.' significa que foi impossível a medida de tempo.	28
Tabela 2 – Tabela de Conversão entre tipo de dado de MPI e tipo de dado de C. .	39
Tabela 3 – Tempos de Execução dos Programas de C Utilizando 4 Processos. . . .	73
Tabela 4 – Tempos de Execução dos Programas de C Utilizando 2 Processos. . . .	73
Tabela 5 – Tempos de Execução dos Programas de Julia Utilizando 4 ou 5 Processos. Nomes marcados com um * utilizaram de 5 processos.	74
Tabela 6 – Tempos de Execução dos Programas de Julia Utilizando 2 ou 3 Processos. Nomes marcados com um * utilizaram de 3 processos.	74
Tabela 7 – Tempos de Execução dos Programas de C e Julia de Canais Remotos Utilizando 4 ou 5 Processos. Nomes marcados com um * utilizaram de 5 processos.	75
Tabela 8 – Tempos de Execução dos Programas de C e Julia de Canais Remotos Utilizando 2 ou 3 Processos. Nomes marcados com um * utilizaram de 3 processos.	75

Sumário

1	INTRODUÇÃO	15
2	JULIA - A LINGUAGEM	17
2.1	Introdução a Julia	17
2.2	Julia Sequencial	17
2.3	Paralelismo em Julia	19
2.3.1	Co-rotinas	20
2.3.2	Processamento Distribuído	21
2.4	Pacotes Utilizados	23
2.5	Exemplos de Julia	24
2.5.1	Exemplo Comparativo Entre Julia, C e <i>Python</i>	24
2.5.1.1	Soma de Dois Vetores Usando Julia Sequencial	24
2.5.1.2	Soma de Dois Vetores Usando <i>Python</i> Sequencial	25
2.5.1.3	Soma de Dois Vetores usando C Sequencial	27
2.5.1.4	<i>Benchmark</i> dos Exemplos Comparativos de Soma de Vetor.	28
2.5.2	Exemplo usando <i>@spawnat</i>	29
2.5.3	Exemplo usando <i>@distributed</i>	31
2.5.4	Exemplo usando <i>pmap</i>	33
3	MECANISMOS DE COMUNICAÇÃO EM SISTEMAS DISTRIBUÍ-	
	DOS	35
3.1	Introdução a MPI	35
3.2	Comandos de MPI	36
3.3	Exemplos de Códigos usando MPI	39
3.3.1	A Estrutura de um Código C com MPI	39
3.3.2	Exemplo Soma de Vetores	42
4	AVALIAÇÃO DE DESEMPENHO	47
4.1	Ambiente de Execução	47
4.2	Programas de Teste	48
4.2.1	Programas de Julia	48
4.2.1.1	Análise do Código.	48
4.2.1.2	Análise do Código <i>pmap.jl</i>	50
4.2.1.3	Análise do Código <i>distributed-sharedarrays.jl</i>	51
4.2.1.4	Análise do Código <i>remote-do-sharedarrays.jl</i>	54
4.2.1.5	Análise do Código <i>@spawnat-remotechannels.jl</i>	57

4.2.2	Programas de C	61
4.2.2.1	Análise do Código <i>send-receive.c</i>	61
4.2.2.2	Análise do Código <i>scatter-gather.c</i>	64
4.2.2.3	Análise do Código <i>bcast-reduce.c</i>	65
4.2.2.4	Análise do Código <i>scatter-reduce.c</i>	67
4.2.2.5	Análise do Código <i>simulacao-remote-channels.c</i>	68
4.3	Avaliação de Desempenho	73
5	CONCLUSÕES	77
	REFERÊNCIAS	79
	APÊNDICES	81
	APÊNDICE A – CÓDIGOS DE JULIA	83
A.0.1	<i>exemplo-sequencial.jl</i>	83
A.0.2	<i>@spawnat.jl</i>	83
A.0.3	<i>pmap.jl</i>	85
A.0.4	<i>remote-do.jl</i>	86
A.0.5	<i>distributed.jl</i>	89
A.0.6	<i>@spawnat-remotechannels.jl</i>	90
	APÊNDICE B – CÓDIGOS DE C	95
B.0.1	<i>exemplo-soma-vetor.c</i>	95
B.0.2	<i>exemplo-send-receive.c</i>	95
B.0.3	<i>exemplo-comunicacao-ponto-a-ponto.c</i>	98
B.0.4	<i>send-receive.c</i>	99
B.0.5	<i>scatter-gather.c</i>	102
B.0.6	<i>bcast-reduce.c</i>	104
B.0.7	<i>scatter-reduce.c</i>	107
B.0.8	<i>simulacao-remote-channels.c</i>	109
	APÊNDICE C – CÓDIGO DE PYTHON	113
C.0.1	<i>soma-vetor.py</i>	113

1 Introdução

Nos dias atuais, C ainda é a linguagem principal que é utilizada quando o objetivo é alto desempenho. C é uma linguagem que foi criada em 1972 que se tornou o padrão comparativo devido aos seus excelentes resultados. C, porém, é uma linguagem de mais baixo nível quando comparado a *Python*, e, por isso, sua escrita pode ser complexa, de difícil entendimento e convoluta. Atualmente, tem-se utilizado linguagens interpretadas em seu lugar, por serem de mais fácil portabilidade e manutenibilidade. *Python*, por exemplo, tem sido uma das linguagens mais populares faz algum tempo devido não só às suas muitas bibliotecas disponíveis graças aos inúmeros colaboradores por trás dele, mas também devido a sua simples leitura e de ser uma linguagem bastante portátil. Usuários com conhecimento da língua inglesa, até mesmo leigos em programação, conseguem extrapolar de um código de *Python* bem escrito, qual o objetivo deste, já em C, apesar de um código ser bem escrito e bem estruturado, ele pode alienar usuários que possuam alguma experiência.

Julia é uma linguagem que foi lançada em 2012. A ambição dos seus criadores é gananciosa, como os próprios disseram. A linguagem Julia foi criada com o seguinte intento:

We want the speed of C with the dynamism of Ruby. We want a language that's homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python [...] We want it interactive and we want it compiled.¹ [Bezanson et al. \(2020b\)](#).

Julia se trata de uma linguagem que foi criada tendo em mente a otimização de diversas aplicações que estão em alta no momento, como: *data mining*, *machine learning*, computação paralela, entre outros.

Neste trabalho, pretende-se fazer uma comparação das estruturas para processamento paralelo oferecidas pela linguagem C com a biblioteca MPI e pela linguagem Julia. O foco do trabalho são as estruturas específicas para programação paralela com troca de mensagens utilizados em ambientes distribuídos. O objetivo desta monografia é compará-las nas questões de: simplicidade e clareza de código, desempenho, pacotes e bibliotecas disponíveis.

Foram estudadas as seguintes estruturas para comunicação de processos em sistemas distribuídos de Julia: *@spawnat*, *RemoteChannels*, *remote-do*, *@distributed*, *pmap* e

¹ O trecho correspondente na tradução é: “Queremos a velocidade de C com o dinamismo de *Ruby*. Queremos uma linguagem que seja homoicônica, com *true macros* tal qual *Lisp*, mas com notações matemáticas familiares e óbvias que nem *Matlab*. Queremos algo que seja utilizável para programação geral que nem *Python* [...] Queremos que seja interativa e queremos que seja compilada.”

SharedArrays. Já em C utilizou-se da biblioteca de troca de mensagens *Message Passing Interface*(MPI). Na biblioteca MPI foram estudadas as seguintes estruturas: *MPI_Send*, *MPI_Recv*, *MPI_Scatter*, *MPI_Gather*, *MPI_Reduce* e *MPI_Bcast*.

Os resultados apontaram Julia como uma alternativa viável para C e para *Python*, com base no seu desempenho comparável a C utilizando de paralelismo bem como a sua clareza de código.

O restante deste trabalho está ordenado da seguinte forma:

- Capítulo 2: Neste capítulo a linguagem Julia é discutida e diferenciada das demais. São esclarecidos os aspectos de execução sequencial e de execução paralela; São explicados suas rotinas e os pacotes utilizados no projeto bem como alguns exemplos que comparam Julia, C e *Python*.
- Capítulo 3: Neste capítulo a *lib* MPI é explicada, junto às suas rotinas, tipos de dados específicos e são apresentados alguns exemplos de códigos empregando MPI.
- Capítulo 4: Neste capítulo o ambiente de execução é detalhado, são explicados os programas de teste utilizados e é feita a comparação entre estes programas levando em consideração o seu tempo de execução medido e sua clareza.
- Conclusões: Neste capítulo os resultados encontrados no Capítulo 4, bem como as informações acerca destes e outras características sobre as estruturas de troca de mensagem em sistemas de memória distribuídas expostas são consultados e comparados para deliberar os pontos positivos e negativos entre as estruturas estudadas.

2 Julia - A Linguagem

2.1 Introdução a Julia

Possuindo como seu foco todas as características positivas das linguagens mais populares, Julia prova ser rápida, fluida, adaptativa aos problemas simples tanto quanto aos complexos e dispendiosos além de possuir fácil leitura do seu código. As razões para isso são muitas, dentre elas o fato de ser uma linguagem *typeless*. Este fato confere muita liberdade, deixa o código mais legível e ganha tempo para o programador. Isso é um fator que contribui para códigos mais limpos, organizados e curtos e é uma característica das linguagens dinâmicas, ao contrário das linguagens estáticas que necessitam explicitar e declarar suas variáveis. *Optional typing*, que é a capacidade da linguagem declarar ou não os tipos em cada uma das variáveis, permite ao usuário realizar um balanceamento entre alto desempenho e generalização do código. Essa característica aliada ao alto desempenho, facilidade de uso e uma comunidade que produz novos pacotes regularmente, são as características que levam as pessoas a adotarem o seu uso.

Julia é uma linguagem de alto nível, de alto desempenho e de propósito geral. Apesar de ser de alto nível, ela consegue atingir uma velocidade próxima de C se o código for otimizado e com uma simplicidade mais aparente. Ou seja, com um investimento menor de tempo, é possível obter retornos similares aos que se teria em C. Julia tem dentre os seus inúmeros objetivos, ser uma linguagem que seria *user friendly*, ou seja, uma linguagem fácil de se operar e boa para iniciantes ou usuários que não tenham como seu foco a programação mas que precisam de uma linguagem de alto-desempenho para manipulação de grande quantidade de dados para pesquisas ou propósitos similares. Por esse motivo, a linguagem discutida possui vários pacotes externos dedicados a facilitar a implementação de computação científica, fora o apoio já dado pela linguagem em si.

2.2 Julia Sequencial

Para se ter uma linguagem com alto desempenho usando paralelismo é necessário que esta linguagem possua uma estrutura sequencial bem articulada. Isso pode ser adquirido com diversas características que não só auxiliam no desempenho em si mas que também economizam tempo para o programador. Características que visam melhorar a qualidade do código não só acelerando sua velocidade de execução mas que possam prover facilidades para a estruturação do código, tal qual os despachos múltiplos.

A tecnologia de despachos múltiplos, que é utilizada em Julia, confere a capacidade

de discriminar qual método se deve utilizar analisando a natureza dos argumentos e a combinação destes apresentados na chamada. Esses argumentos são declarados junto à função, tal qual: “ *function foo(argument::type)* ”. Julia foi construída tendo como um de seus alicerces o uso de despachos múltiplos já que um dos seus grandes objetivos é facilitar a computação numérica e nesta é necessário muitas vezes diferentes combinações dos mesmos tipos de argumentos. Assim, diferentes métodos de uma mesma função podem ter sua chamada diferenciada por essa mecânica da inspeção de todos os argumentos e os seus respectivos *types* tendo esses vários métodos mais fáceis de usar e organizados.

Application Programming Interface(interface de programação de aplicativos) realiza comunicações entre diferentes programas. Em Julia, isso se traduz na capacidade de utilizar códigos, funções ou métodos em C ou *Python* sem a necessidade de adaptá-los à linguagem Julia, garantindo maior velocidade de desenvolvimento devido ao aproveitamento desses métodos pré-existentes. Isso implica que o programador possa abstrair desse método e criar códigos mais simples e mais limpos.

A linguagem possui o REPL, que é um console que permite interação e compilação em tempo real de linhas alimentadas ao REPL. É uma ferramenta poderosa para testes, *debugs* e para isolar certas partes de um código nas quais é desejado um estudo aprofundado.

Infelizmente, Julia não é uma linguagem tão popular quanto as que a maioria das pessoas usa e, ao mesmo tempo, é uma linguagem nova, tendo sido divulgada oficialmente em 2012. Além disso, Julia sofre *updates* constantes, isso faz com que ela passe por muitas versões diferentes dificultando a busca por informações. Isso ocorre já que, com cada nova versão, os pacotes podem ser alterados e os comandos podem ficar obsoletos ou começar a ter outra função. Ademais, a maioria das discussões sobre Julia é encontrada na língua inglesa devido a ela ainda não ser muito conhecida, tornando pesquisas mais desafiadoras.

Outra característica importante de citar seria que é possível utilizar seções e até mesmo pacotes de outras linguagens, como C ou *Python*, em códigos de Julia. Isso permite o aproveitamento de funções já implementadas sem a necessidade de sua adaptação. Julia também emprega o uso de um tipo de compilação denominado *just-in-time compilation*(compilação sob demanda). Essa forma de compilação implica na compilação do código enquanto ocorre a execução do mesmo, ao invés de antes da execução, como ocorrem nas compilações normais.

Nesta seção também se explica brevemente alguns dos comandos utilizados que possuem importância no projeto mas não pertencem aos pacotes mencionados nas seções abaixo. A documentação sobre Julia v1.1.1 e as rotinas discutidas nessa seção podem ser encontradas em <<https://docs.julialang.org/en/v1.1/>>.

- *Base.map*: Se trata de uma rotina que aplica uma função à escolha do usuário em uma coleção especificada. A coleção pode ser um vetor ou matriz por exemplo.

- *const*: É utilizado para declarar variáveis globais cujos valores não irão mudar. Por motivos de desempenho, é preferível utilizar *const*, se possível, no lugar de *global* pois *global* é difícil de ser otimizado pelo compilador pelo fato de que a variável pode ser alterada em qualquer ponto no código. Com a declaração de *const* no lugar de *global* se está garantindo ao compilador que o valor ali declarado não irá mudar, o que facilita a otimização do código pelo compilador.
- *global*: É utilizado para marcar uma variável. Uma vez marcada, esta mesma variável pode ser utilizada em outro segmento do código se precedida por “*global*”. Desta forma, variáveis globais são importantes dentro de *for* e funções, quando necessário. O *for* não aceita alterar variáveis externas a não ser que sejam marcadas como globais.
- *rand*: É uma função utilizada para produzir um ou mais números aleatórios num alcance especificado pelo usuário. A versão de *rand* em Julia pode receber *seed* tal qual o seu semelhante em C, e é possível usar algum método conhecido, como por exemplo *Mersenne Twister* para dar base aos números aleatórios produzidos.

2.3 Paralelismo em Julia

Julia oferece três categorias principais de paralelismo. Elas são representadas através de: Co-rotinas em Julia (*Greenthreading*), *Multi-Threading* e *Distributed* (Processamento Distribuído). Neste trabalho não será discutido *Multi-Threading* pois em Julia v 1.1.1 este era considerado instável.

É explicado um pouco de cada uma das formas mencionadas de paralelismo para que possa ser aprofundado o conhecimento em seções futuras de forma mais esclarecida.

Usar co-rotinas é a possibilidade de suspender a execução de uma *task* para se rodar outra e retomar a execução da *task* suspendida quando em outro momento mais oportuno. Isso faz com que elas possam dar a um programa uma espécie de paralelismo através da sua característica de poderem trocar a *task* que está rodando com outra que está na espera.

Multi-Threading se trata da capacidade de rodar múltiplas *threads* que compartilham memória. Isso implica que um programa, ou uma seção deste, pode ser executado paralelamente em vários núcleos de processamento de um só computador através do emprego de múltiplas *threads*.

Processamento Distribuído é rodar um programa, ou uma seção deste, em diversos processos, que não compartilham memória. Isso significa poder utilizar vários computadores e seus vários núcleos de processamento para rodar um código ou aproveitar de uma única máquina que tenha múltiplos núcleos. Basicamente, é executar um código, ou uma seção

deste, paralelamente em uma quantidade de processos ao desejo do usuário sem que esses processos compartilhem memória.

2.3.1 Co-rotinas

Quando se trabalha com co-rotinas em Julia, o elemento mais destacado são os *Channels*(canais) e as *tasks*. Esses canais permitem a comunicação entre *tasks* tal qual um *buffer* em um problema de Consumidor e Produtor.

É importante antes de prosseguir para explicar as várias rotinas e macros abordadas, alguns termos chaves para o entendimento.

- *Future* é um parâmetro de retorno. Ele é utilizado para resgatar o retorno de uma computação em outro processo ou um valor armazenado através de *put!*.
- *Macro* é definido por [Metaprogramming \(2020\)](#): “Macros provide a method to include generated code in the final body of a program. A macro maps a tuple of arguments to a returned expression, and the resulting expression is compiled directly rather than requiring a runtime eval call.”¹

Alguns comandos que são utilizados quando se pretende fazer uso das vantagens das co-rotinas são explicados a seguir.

- *Base.@async*: Este macro faz com que a expressão designada seja envolta numa *task* e que seja adicionada a lista de agendamento da máquina local. Lista esta que rotula as *tasks* existentes pelo estado de execução destas. Uma *task* pode estar em uma das seguintes situações: rodando, suspensa, bloqueada, na fila de execução, finalizada com sucesso e finalizada com falha.
- *Base.@sync*: Este macro faz com que seja esperado o término de todas expressões envolvidas pelos comandos *@async*, *@spawn*, *@spawnat* e *@distributed* antes de prosseguir a execução do código.
- *Base.wait*: Esta função, dependendo do argumento passado, executa métodos diferentes. *wait* bloqueia a *task* atual até que um evento que depende do tipo do argumento passado ocorra. *wait* quando o argumento é um *Future*, aguarda até que o valor deste esteja disponível antes de prosseguir no código. *wait* quando o argumento é um *RemoteChannel* ou *Channel* aguarda até que um valor fique disponível no canal especificado antes de prosseguir no código. *wait* quando o argumento é um processo ou uma *task* aguarda o processo ou *task* serem finalizados.

¹ O trecho correspondente na tradução é: “Os macros fornecem um método para incluir o código gerado no corpo final de um programa. Uma macro mapeia uma tupla de argumentos para uma expressão retornada e a expressão resultante é compilada diretamente em vez de exigir uma chamada de avaliação de tempo de execução.”

- *Base.take!:* Remove e retorna um valor de um canal.
- *Base.put!:* Esta função, dependendo do argumento passado, executa métodos diferentes. *put!* quando o primeiro argumento é um *RemoteChannel* armazena o valor passado no segundo argumento no canal. Caso este canal esteja cheio, o canal é bloqueado até que algum espaço seja liberado. *put!* quando o primeiro argumento é um *future* armazena o valor passado no segundo argumento dentro deste *future*.
- *Base.isready!:* Esta função, dependendo do argumento passado, executa métodos diferentes. *isready* quando o argumento é um *RemoteChannel* determina se um canal possui algum valor armazenado. *isready* quando o argumento é um *future* determina se este futuro está disponível.

Pode se perceber que esses comandos ou interagem com um Canal, ou interagem com a execução de *tasks*. Isso ocorre pois administrar as *tasks* e utilizar dos canais para alimentar informação para essas mencionadas *tasks* é a aplicação de co-rotinas no código.

Ambos os comandos *take!* e *put!* mencionados podem ser utilizados por diferentes *tasks* no mesmo canal concomitantemente, permitindo diversos acessos simultâneos a um só canal, sejam os acessos para depósito ou remoção de informações. Além do mais, o canal é uma estrutura de armazenamento de dados do tipo *first in first out*, isso significa que o primeiro dado a ser armazenado através de um *put!* será o primeiro a ser removido quando se usar do comando *fetch!* neste mesmo canal.

O *pipeline* mencionado é estabelecido nas co-rotinas através da utilização dos canais quando na presença de operações de entrada e saída. As co-rotinas em Julia operam de forma que, apesar das *tasks* fazerem sua computação sequencialmente, as suas operações de entrada e saída são paralelas. Isso ocorre devido a essa versão de Julia multiplexar todas as *tasks* numa só *thread*. Isso implica que se uma *task* possuir operações I/O, ela será beneficiada pelo paralelismo, caso contrário, será executada sequencialmente, o que não iria gerar o *pipeline* desejado. Naturalmente, não se terá somente *tasks* para o propósito de I/O, e mesmo estas *tasks* com esse propósito, não são completamente I/O, por isso o nível de paralelismo que conseguimos atingir utilizando exclusivamente de co-rotinas é o de um *pipeline*. Que ocorre mais notavelmente quando a *task* responsável pelo I/O troca com outra *task*, ou seja, cede a ela os seus privilégios computacionais, enquanto espera pedidos de I/O. Caso isso não fosse feito, a *task* esperando pelos pedidos de I/O impediria outras *tasks* de rodarem e não faria nada além de esperar o retorno dos pedidos.

2.3.2 Processamento Distribuído

Este assunto é um dos focos deste trabalho. Sua importância está diretamente ligada ao objetivo do mesmo. O pacote *distributed* é a representação de paralelismo de

memória distribuída em Julia. Nele estão presentes comandos e recursos que permitem a comunicação entre diversos processos.

Devido à sua falta de memória compartilhada entre os processos do processamento distribuído, uma aplicação distribuída simples funciona da seguinte forma: o processo mestre distribui dados a serem computados pelos *workers*. Os processos realizam operações nestes dados e os retornam ao processo mestre já modificados.

Por ser um dos pacotes mais utilizados nesse trabalho, são destrinchados os comandos utilizados no projeto, bem como suas funcionalidades para o entendimento básico dos programas que estão presentes nesta monografia.

- *Distributed.remote_do*: Este método executa uma função, a escolha do usuário, no *worker* número *id* assincronamente. Esse método não produz um *future*, portanto não pode ter seu resultado resgatado através de um comando *fetch*. Esse método também não armazena o resultado da sua computação, o que faz com que seja interessante fazer com que ele altere diretamente os dados desejados.
- *Distributed.@spawnat*: Esta macro cria um fecho em torno de uma expressão e executa esta expressão assincronamente em um processo à escolha do usuário. Este macro retorna um futuro e aceita dois argumentos, o número do processo e uma expressão.
- *Distributed.fetch*: Trata-se de uma macro com o propósito de recuperar o resultado de um *future* que tenha sido ou esteja sendo processado.
- *Distributed.@distributed*: A macro *@distributed* divide e executa de forma assíncrona e local o *loop* declarado entre todos os *workers* disponíveis. É uma implementação de paralelismo de memória distribuída. Existe a opção de se introduzir uma função redutora como “+” ou “*”.
- *Distributed.@everywhere*: Esta macro tem como objetivo executar uma expressão em todos os processos. Para que uma função declarada no processo principal possa ser utilizada nos *workers* é necessário que ela esteja declarada com esta macro.
- *Distributed.addprocs*: A função *addprocs* inicia processos além do principal, ou seja, *workers*, na máquina local, quando usada de forma simplificada.
- *Distributed.nprocs*: Esta função retorna o número de processos disponíveis.
- *Distributed.nworkers*: Esta função retorna o número de *workers* disponíveis.
- *Distributed.myid*: Esta função retorna o número de identificação do processo atual, ou seja, seu *id*.

- *Distributed.pmap*: Se trata de uma função que aplica uma função a escolha do usuário em uma coleção especificada. A coleção pode ser um vetor ou matriz, por exemplo. Além disso, *pmap* utiliza dos *workers* disponíveis e *tasks* para paralelizar sua execução. O tipo de paralelismo empregado por *pmap* é de memória distribuída.
- *Distributed.RemoteChannel*: O *RemoteChannel* é um *type* que é uma estrutura de armazenamento de dados local para um *worker*. É possível um *worker* fazer uma operação *put!*, *take!* ou *isready!* no *RemoteChannel* de outro *worker*. Essa ação pode ser considerada uma troca de mensagem entre estes *workers*. Os dados no *RemoteChannel* são armazenados em forma de uma pilha.
- *Base.put!*: É um método com o objetivo de armazenar um valor dentro de um *RemoteChannel*. O valor a ser armazenado pode ser um vetor, ou até mesmo inúmeros vetores, este método os passa para o *RemoteChannel* na forma de uma tupla.
- *Base.take!*: É um método com o objetivo de recuperar e remover um valor de dentro de um *RemoteChannel*.
- *Base.isready!*: É um método com o objetivo de determinar se um *RemoteChannel* possui algum valor armazenado em si.
- *Distributed.@fetchfrom*: É uma macro que faz o equivalente a utilizar uma macro de *fetch* encapsulando um *@spawnat*. Ou seja: “*@fetchfrom p expr*” equivale a “*fetch(@spawnat p expr)*”, onde “*p*” é o número do *worker* e “*expr*” é a expressão desejada.

2.4 Pacotes Utilizados

Julia possui muitos pacotes externos, dentre eles, *SharedArrays* e *BenchmarkTools* são importantes para este projeto, portanto serão explicados para terem seu propósito esclarecido de forma breve.

O pacote *BenchmarkTools* se trata de uma ferramenta com uma extensa configuração. Ele permite o uso de várias rotinas e macros, como *@benchmarkable* que, se configuradas e utilizadas corretamente, permitem uma medição com ruídos diminuídos através de seus parâmetros como *time_tolerance* e *memory_tolerance*. Estes dois parâmetros filtram os resultados das medições que não atendam os critérios percentuais de tempo e memória ao apresentarem um desvio maior que um certo valor que o usuário atribui a eles. Além disso, é possível estipular a quantidade de medições a serem feitas através do parâmetro *samples*. O motivo de se ter empregado este pacote e usar das suas rotinas ao invés de usar a rotina *@time*, que é parte do Julia básico, é pelo fato que as medições usando as rotinas do *BenchmarkTools* são muito mais customizáveis. O pacote também

possui parâmetros que facilitam a filtragem das medidas de tempo e oferece soluções a problemas que consomem o tempo do usuário. Problemas estes que seriam: fazer uma média de múltiplas medidas executadas num mesmo trecho do código; a impressão em tela do tempo mínimo ou médio de execução dentre as medidas feitas; a possibilidade de utilizar a rotina *tune!*, que automaticamente encontra um balanço entre os parâmetros utilizados para se fazer o *benchmark* do código; acesso aos parâmetros que foram utilizados para fazer o *benchmark*, permitindo que o próprio usuário possa fazer alterações na medição, de forma que seu código seja cautelosamente medido. A documentação do pacote *BenchmarkTools* pode ser encontrada em: <<https://github.com/JuliaCI/BenchmarkTools.jl>>.

O pacote *SharedArrays* permite a criação de um tipo chamado *SharedArray*. Um *SharedArray* se trata de um *array* que é construído em todos os processos, ou em alguns, caso sejam especificados. O interessante desse pacote é que esse *array* criado é o mesmo para todos, isso implica que qualquer alteração feita nele por qualquer um dos processos, irá modificar os dados armazenados no *array* para todos. É um pacote com uma funcionalidade importante pois permite aumentar o desempenho em muitos programas por fazer desnecessário o retorno dos cálculos dos *workers* para o processo mestre. Ao invés disso, eles podem simplesmente ser armazenados em suas determinadas casas num único vetor que está disponível para o acesso de todos. Uma particularidade sua é que os *workers*, apesar de possuírem acesso ao *array* compartilhado, não têm como referenciá-lo, então para utilizá-lo nos *workers* é preciso passar os *SharedArray* por referência de alguma forma. Uma vez feito isso, o *worker* consegue ler e modificar o *SharedArray* através da referência. As modificações feitas por ele são “observadas” na leitura do *SharedArray* pelos outros *workers*. A documentação do pacote *SharedArrays* pode ser encontrada em: <<https://docs.julialang.org/en/v1.1/stdlib/SharedArrays/#SharedArrays.SharedArray>>.

2.5 Exemplos de Julia

Nesta seção, se encontram exemplos com o objetivo de esclarecer algumas rotinas e, de um modo geral, a linguagem Julia.

2.5.1 Exemplo Comparativo Entre Julia, C e Python

Abaixo, são comparadas as três linguagens, C, *Python* e Julia, usando a soma de dois vetores. Estes vetores são populados com números de 1 a 100, os programas somam estes dois vetores e armazenam o resultado em um terceiro vetor.

2.5.1.1 Soma de Dois Vetores Usando Julia Sequencial

Na Listagem 2.1 tem-se o código de Julia completo. Na linha 1,2 e 3 são declarados os usos dos pacotes. O *BenchmarkTools* é utilizado para uma medição mais apurada do

tempo de execução. O *Random* permite realizar o *seed* de *rand*. O *Statistics* dá acesso a *mean*, usada para tirar a média dos valores de tempo medidos.

Na linha 6, a função *popula_vetor* é utilizada para devolver um vetor com inteiros do tipo *Int8* que é mais leve do que *Int64*.

Nas linhas 9 e 10, os vetores são declarados e populados usando da função *popula_vetor*. Bem como *NUM_P* no início do código, os vetores são declarados como *const* para acelerar o desempenho ao facilitar a compilação.

Na linha 11, usa-se da rotina *map* para fazer a soma de todos os elementos do *vetor1* com os de *vetor2*. O tempo médio de execução de *map* é medido usando o macro *@benchmarkable* que permite uma configuração mais extensa através de parâmetros. Parâmetros esses como *time_tolerance* que exige que os valores medidos não desviem mais do que uma porcentagem da média de tempo. Existe também o *memory_tolerance* que exige que os valores medidos não desviem mais do que uma porcentagem da média de memória usada na execução. O *samples* que define quantas vezes a medição será feita. E *seconds* que define o limite de segundos que todas as medições podem tomar.

A linha 12 imprime a média das 20 medidas e roda o segmento encapsulado pela macro *@benchmarkable*.

```

1 using BenchmarkTools
2 using Random
3 using Statistics
4 const NUM_P=1000000
5 Random.seed!(1234)
6 function popula_vetor()::Array{Int8}
7     return rand(1:100, NUM_P)
8 end
9 const vetor1=popula_vetor()
10 const vetor2=popula_vetor()
11 b=@benchmarkable map(+, vetor1, vetor2) memory_tolerance=0.01
    time_tolerance=0.01 seconds=1500 samples=20
12 println(mean(run(b)))

```

Listing 2.1 – exemplo-sequencial.jl

2.5.1.2 Soma de Dois Vetores Usando *Python* Sequencial

Analisando o código de *Python* pode-se perceber que ele e o código de Julia possuem uma quantidade muito baixa de linhas utilizadas em relação ao código em C. Para se conseguir o mesmo que as muitas linhas de C, *Python* só precisa de 19. Naturalmente, essa simplicidade é conseguida às custas de desempenho por *Python* se tratar de uma linguagem interpretada. C, sendo uma linguagem compilada, ganha em desempenho e perde algumas facilidades que *Python* e Julia têm acesso. Julia é também uma linguagem

compilada, mas é uma linguagem compilada *just in time*, isso significa que Julia adquire características de *Python* e mantém um alto desempenho de linguagens compiladas.

Analisando o código na Listagem 2.2, na linha 1 e 2, declara-se a utilização das *libs time* e *random*.

Já na linha 3, declara-se *NUM_P*, para comodidade da alteração deste valor pelo usuário.

Na linha 4 é criada a função *popula_vetor* que, recebendo um vetor como argumento, atribui a ele *NUM_P* números inteiros aleatórios de um a cem.

São declarados os vetores *vetor1*, *vetor2* e *vetorsoma* respectivamente, nas linhas 7, 8 e 9.

Na linha 11 e 12, a função *popula_vetor* é chamada passando como argumentos os vetores *vetor1* e *vetor2*.

Na linha 13 é começa o *loop* que faz a medição da computação 20 vezes.

Na linha 14 começa a medida de tempo que é parada na linha 17. Este intervalo de tempo é medido 20 vezes e é somado em *tempofinal* na linha 18. *tempofinal* é impresso na linha 19 quando dividido por 20, para tirar a média, e multiplicado por 1000, para que ele seja passado para milissegundos.

```
1 import time
2 import random
3 NUM_P=20000000
4 def popula_vetor(x):
5     for _ in range(NUM_P):
6         x.append(random.randint(1,100))
7 vetor1=[]
8 vetor2=[]
9 vetorsoma=[]
10 tempofinal=0.0
11 popula_vetor(vetor1)
12 popula_vetor(vetor2)
13 for _ in range(20):
14     inicio = time.time()
15     for elemvetor1, elemvetor2 in zip(vetor1, vetor2):
16         vetorsoma.append(elemvetor1 + elemvetor2)
17     fim = time.time()
18     tempofinal+=fim-inicio
19 print("Tempo de execucao:", (tempofinal/20)*1000, "ms.")
```

Listing 2.2 – soma-vetor.py

2.5.1.3 Soma de Dois Vetores usando C Sequencial

Em C, fez-se um código que é notavelmente mais complexo do que o seu semelhante em Julia. A quantidade de linhas não reflete a quantidade de instruções passadas pois, para deixar o código mais compacto e facilitar visualização, as declarações foram agrupadas em linhas. O código, como pode ser visto pela quantidade de instruções utilizadas para se obter o mesmo resultado do que em Julia e *Python*, é extenso e bem mais complexo, apesar de permitir ao usuário maior liberdade.

Na Listagem 2.3, são inclusas as *libs* necessárias para o funcionamento do programa nas 3 primeiras linhas. Depois disso, na linha 4, é definido *NUM_P*, permitindo sua alteração com facilidade para outros valores compatíveis. Na linha 5 se declara a função *gera_vetor* que utiliza a rotina *malloc* para retornar alocações de memória do tamanho de *NUM_P* inteiros.

Seguindo adiante, já entrando em *main*, os três vetores utilizados no programa, *vetor1*, *vetor2* e *vetorsoma* são declarados como ponteiros para inteiros nas linhas 8. Nota-se também a utilização de *srand* e da *lib time.h* para criar uma base que permite que os números aleatórios gerados sejam mais independentes do computador.

Na linha 10 os três vetores necessitam da mesma alocação de memória e todos trabalham com o tipo de dado *int*, então utiliza-se a função *gera_vetor* para todos. Permitindo que os vetores *vetor1* e *vetor2* sejam populados através de um *for* e usando *rand* nas linhas 11,12 e 13.

Na linha 17 se dá início ao *loop* que tem o início de sua medição de tempo nas linhas 15 e 16. Esse *loop* é encerrado na linha 21 e ele é executado 20 vezes. Isso faz com que o tempo medido no intervalo começando na linha 16 e terminando na linha 22 meça 20 execuções do *loop while*.

Nas linhas 18 e 19 há um *for* que realiza *NUM_P* iterações. Cada uma dessas iterações atribui a soma de uma casa de *vetor1* com uma casa de *vetor2* a uma casa de *vetorsoma*. A casa do vetor em questão é a mesma para todos e é determinada pelo valor da iteração atual, *i*, sendo que este varia seu valor entre 0 e *NUM_P*-1, fazendo assim todas as operações necessárias desejadas.

Terminando as repetições, o código segue para imprimir em tela a média dos tempos medidos na linha 24. Tempo são 20 intervalos de tempo medidos entre as linhas 16 e 22. Logo, dividindo o valor de Tempo por 20, tem-se a média da medição de tempo.

Na linha 25 é liberada a memória que havia sido alocada pela função *gera_vetor* aos vetores utilizados no programa e, por fim, encerra com *return 0*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
```

```

4 #define NUM_P 50000000
5 int* gera_vetor() {
6     return (int*)malloc(NUM_P*sizeof(int));}
7 int main(int argc, char *argv[]){
8     int* vetor1; int* vetor2; int* vetorsoma;
9     srand(time(NULL));
10    vetor1=gera_vetor(); vetor2=gera_vetor(); vetorsoma=gera_vetor();
11    for(int i=0; i<NUM_P; i++){
12        vetor1[i]=rand() % 100+1;
13        vetor2[i]=rand() % 100+1;}
14    int repeticao=20;
15    clock_t Ticks[2];
16    Ticks[0]=clock();
17    while(repeticao>0){
18        for(int i=0; i<NUM_P; i++){
19            vetorsoma[i]=vetor1[i]+vetor2[i];}
20        repeticao--;
21    }
22    Ticks[1]=clock();
23    double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
24    printf("Tempo de execução médio:%fms\n", Tempo/20);
25    free(vetor1); free(vetor2); free(vetorsoma);
26    return 0;
27 }

```

Listing 2.3 – exemplo-soma-vetor.c

2.5.1.4 Benchmark dos Exemplos Comparativos de Soma de Vetor.

Nestes exemplos vistos nas subseções acima, foram utilizadas quantidades pequenas em *NUM_P*. Agora, utilizando números maiores, é possível perceber a diferença no desempenho variando o valor de *NUM_P* e traçar conclusões a respeito desses dados. A Tabela 1 apresenta 4 variações do valor de *NUM_P* para estudo.

Tabela 1 – Tempos de Execução dos Exemplos Soma de Vetores usando Julia, C e *Python* Sequenciais. O 'n.p.m.' significa que foi impossível a medida de tempo.

Valor de <i>Num_P</i>	Julia	C	<i>Python</i>
1.000.000	2,720 ms	1,489 ms	275,569 ms
10.000.000	31,087 ms	15,605ms	2970,945 ms
20.000.000	57,020 ms	31,341 ms	n.p.m.
50.000.000	143,143 ms	78,382 ms	n.p.m.

As medições são feitas da seguinte forma para as linguagens:

- Para C se tem um *while* que repete 20 vezes o intervalo de código responsável pela computação que se deseja medir. Esse *while* tem o seu tempo de execução medido e

armazenado em uma variável chamada de *Tempo*. Depois do fim do *loop while*, é impresso o valor de *Tempo* dividido por 20, conseguindo-se a média de tempo de uma execução. Além disso, foi utilizado o parâmetro de otimização de *gcc -O3*.

- Para *Python* se tem um *for* que repete 20 vezes a medida de tempo do intervalo de código responsável pela computação que se deseja medir. Esse intervalo de tempo medido vai sendo somado a uma variável chamada *tempofinal* e, ao fim do *for*, é impresso o seu valor dividido por 20 e multiplicado por 1000, conseguindo-se a média em milissegundos.
- Para Julia se utiliza o macro *@benchmarkable* encapsulando a computação que se deseja medir. Usando o parâmetro *samples* se regula o número de medições feitas para 20. Para uma medição com um ruído reduzido, são modificados os parâmetros *memory_tolerance* e *time_tolerance* para 0,01. Esses parâmetros exigem que os valores medidos não desviem mais do que uma certa porcentagem da média de tempo e de memória encontrados nessas 20 repetições. Não foi utilizado o parâmetro de otimização *-O3* pois ele produz aproximadamente o mesmo resultado ou um resultado um pouco mais lento.

Analisando a tabela, Julia mostra um valor de execução aproximadamente 85% mais lento na média de todas as medições quando comparado a C. Julia quando comparado a *Python* mostra um valor de execução aproximadamente 98% mais rápido na média das duas primeiras medições.

Python rapidamente se torna inviável em questão tanto de desempenho quanto da escala das unidades. Chegando no valor de 20 milhões, *Python* não conseguiu retornar um valor de tempo. Devido a grande alocação de memória necessária para o programa, o *OOM killer* (matador de processos por memória insuficiente) do *Ubuntu* encerra o processo e impossibilita a medição da mesma forma que os demais programas. Isso ocorre pois o *Python* assume todas as variáveis como sendo do tipo *double*, o que sobe muito o custo de memória.

2.5.2 Exemplo usando *@spawnat*

Na Figura 1 se tem o uso do REPL de Julia utilizando do pacote *Distributed* para fazer com que o processo dois execute a função “quadrado”, também declarada na figura, no número aleatório “*number*”.

Fazendo uma análise linha a linha, temos na primeira linha de comando representada na Figura 2, após o símbolo de Julia, escrito “*using Distributed*”. Isso inclui o pacote *Distributed* e permite o uso das rotinas que ele contém.

Figura 1 – Implementação de *@spawnat* e *@fetchfrom* utilizando o REPL.

```
bruno@bruno-Z450UA:~$ julia
┌──────────┴──────────┐ Documentation: https://docs.julialang.org
│                  │ │ Type "?" for help, "]"? for pkg help.
│                  │ │ Version 1.1.0 (2019-01-21)
│                  │ │ Official https://julialang.org/ release
└──────────┴──────────┘

julia> using Distributed

julia> addprocs(2)
2-element Array{Int64,1}:
 2
 3

julia> number=rand(1:100)
20

julia> @everywhere function quadrado(x::Int64)
    return x*x
end

julia> fut=@spawnat 2 quadrado(number)
Future{2, 1, 8, nothing}

julia> resultado=fetch(fut)
400

julia> resultado2=@fetchfrom 2 quadrado(number)
400

julia> resultado3=@fetchfrom 3 quadrado(number)
400
```

Figura 2 – Linha que declara uso do pacote *Distributed*.

```
bruno@bruno-Z450UA:~$ julia
┌──────────┴──────────┐ Documentation: https://docs.julialang.org
│                  │ │ Type "?" for help, "]"? for pkg help.
│                  │ │ Version 1.1.0 (2019-01-21)
│                  │ │ Official https://julialang.org/ release
└──────────┴──────────┘

julia> using Distributed
```

Na próxima linha, na Figura 3, se declara “*addprocs(2)*”. Agora, o REPL adiciona dois processos além do que já se tinha, no caso, somente o processo número 1. A resposta dessa linha, são os processos adicionados, o processo número 2 e o processo número 3.

Figura 3 – Adicionando 2 ao número de processos.

```
julia> addprocs(2)
2-element Array{Int64,1}:
 2
 3
```

Na linha seguinte, se declara a variável *number* utilizando a rotina *rand*. Esta rotina gera a quantidade especificada de número aleatórios dentro do intervalo desejado, no caso o intervalo foi entre 1 e 100. A resposta é o número aleatório gerado “20”. Como se pode ver na Figura 4.

A seguir, na Figura 5, empregando a rotina *@everywhere*, é declarada a função *quadrado*, que tem como argumento um único inteiro e retorna a multiplicação deste

Figura 4 – Declarando a variável *number* usando a rotina *rand*.

```
julia> number=rand(1:100)
20
```

inteiro por ele mesmo, em todos os processos.

Figura 5 – Declarando a função *quadrado*.

```
julia> @everywhere function quadrado(x::Int64)
    return x*x
end
```

Aqui na Figura 6 declara-se *fut*, que é a variável que contém o *future* do *@spawnat* declarado em sua atribuição. Esta rotina necessita receber um *fetch* para que se possa resgatar seu resultado. Neste caso, o processo número 2 recebeu a tarefa de executar a função *quadrado* tendo como argumento *number*.

Figura 6 – Utilização de *@spawnat*.

```
julia> fut=@spawnat 2 quadrado(number)
Future{2, 1, 8, nothing}
```

Em seguida, é necessário que o *future* seja resgatado para se obter o valor desejado da função. Para isso, serviria entrar *fetch(fut)* mas aqui, como pode ser observado na Figura 7, se atribui a uma variável chamada “*resultado*” o *fetch* do *future* contido em *fut*.

Figura 7 – Uso de *fetch*.

```
julia> resultado=fetch(fut)
400
```

Na Figura 8, existem duas linhas de comando. A primeira mostra o uso do *fetchfrom*, uma rotina que junta em um só comando as rotinas *@spawnat* e *fetch*. Esta instrução reproduz a mesma coisa contida nas Figuras 6 e 7. Na linha inferior, é feita a mesma coisa, porém no processo número 3.

Figura 8 – Uso da rotina *fetchfrom* nos processos número 2 e 3.

```
julia> resultado2=@fetchfrom 2 quadrado(number)
400

julia> resultado3=@fetchfrom 3 quadrado(number)
400
```

2.5.3 Exemplo usando *@distributed*

Na Figura 9 se tem o uso do REPL de Julia utilizando do pacote *Distributed* e de sua rotina *@distributed* para fazer com que os *workers*, os processos número 2, 3 e 4,

executem a função *quadrado* em números dentro do vetor *numbers* e os armazenem num vetor do tipo *SharedArray*.

Figura 9 – Implementação de *@distributed* junto de *SharedArrays* utilizando o REPL.

```
julia> using Distributed
julia> using SharedArrays

julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4

julia> @everywhere function quadrado(x::Int64)
    return x*x
end

julia> numbers=rand(1:100, 1, 12)
1×12 Array{Int64,2}:
 36 24 81 88 100 1 39 30 4 52 30 18

julia> vetorresultado=SharedArray{Int64}(1, 12)
1×12 SharedArray{Int64,2}:
 0 0 0 0 0 0 0 0 0 0 0 0

julia> futuro=@distributed for i in 1:length(numbers)
    vetorresultado[i]=quadrado(numbers[i])
end
Task (queued) @0x00007f703a765ae0

julia> fetch(futuro)

julia> vetorresultado
1×12 SharedArray{Int64,2}:
1296 576 6561 7744 10000 1 1521 900 16 2704 900 324
```

Similar ao último programa, o da Figura 1, o cabeçalho começa com a declaração dos pacotes que são usados. Neste caso, se pode notar na Figura 10 os pacotes *Distributed* e *SharedArrays* sendo declarados. O *SharedArrays* é útil pois ele simplifica a recuperação dos resultados dos *workers* para o processo número 1. Além disso, se adicionam aqui 3 processos.

Figura 10 – Declaração dos pacotes utilizados e do número de processos adicionados.

```
julia> using Distributed
julia> using SharedArrays

julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4
```

A seguir, na Figura 11, se declara a função *quadrado* mais uma vez e a variável a ser utilizada *numbers*, um vetor de doze casas, populado por inteiros existentes no intervalo de um e cem.

Aqui na Figura 12 o vetor *vetorresultado* tem como seu tipo um *SharedArray* que contém doze números inteiros. Por se tratar de um *SharedArray*, todos os processos compartilham este vetor e seu conteúdo.

A variável *futuro*, que contém o *future* da rotina *@distributed*, está presente na Figura 13. Nesta se pode verificar o uso da rotina *@distributed* no intervalo de 1 até

Figura 11 – A função quadrado e o vetor de inteiros *numbers*.

```
julia> @everywhere function quadrado(x::Int64)
    return x*x
end

julia> numbers=rand(1:100, 1, 12)
1×12 Array{Int64,2}:
 36 24 81 88 100 1 39 30 4 52 30 18
```

Figura 12 – *Vetorresultado* e o tipo *SharedArray*.

```
julia> vetorresultado=SharedArray{Int64}(1, 12)
1×12 SharedArray{Int64,2}:
 0 0 0 0 0 0 0 0 0 0 0 0
```

`length(numbers)`, que é o tamanho do vetor *numbers*, no caso este tamanho é doze pois ele possui doze elementos como declarado na Figura 11. Dentro do `for`, sendo partilhado por `@distributed`, existe a instrução que atribui ao *SharedArray* *vetorresultado* na casa “*i*”, o valor resultado da aplicação da função quadrado na casa “*i*” da variável *numbers*.

Figura 13 – A utilização de `@distributed` gerando um *future*.

```
julia> futuro=@distributed for i in 1:length(numbers)
    vetorresultado[i]=quadrado(numbers[i])
end
Task (queued) @0x00007f703a765ae0
```

Similar a Figura 7, aqui na Figura 14 são recuperados os resultados utilizando a rotina `fetch` usando como seu argumento o *future* que se deseja recuperar. Feito isso, para verificar se está tudo correto, é chamado *vetorresultado* para que possa ser impresso seu conteúdo.

Figura 14 – Recuperação do *future* “*futuro*” e impressão do conteúdo de *vetorresultado*.

```
julia> fetch(futuro)

julia> vetorresultado
1×12 SharedArray{Int64,2}:
1296 576 6561 7744 10000 1 1521 900 16 2704 900 324
```

2.5.4 Exemplo usando *pmap*

Utilizando da rotina `pmap` para cumprir o mesmo objetivo do programa da Figura 9, se percebe que não foi usado o pacote *SharedArrays* pois o `pmap` está retornando o resultado para o processo número 1 para dentro do vetor *vetorresultado*. Esse programa, representado na Figura 15, é notavelmente mais simples que o seu correspondente em `@distributed` mas este possui características aqui não exploradas que validam a sua utilização.

O pacote utilizado no programa e o número de processos adicionados são explicitados na Figura 16.

Figura 15 – Implementação de *pmap* utilizando o REPL.

```
julia> using Distributed

julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4

julia> @everywhere function quadrado(x::Int64)
    x*x
end

julia> numbers=rand(1:100, 1, 12)
1×12 Array{Int64,2}:
 68  5  36  27  9  66  52  88  17  30  33  5

julia> vetorresultado= pmap(quadrado, numbers)
1×12 Array{Int64,2}:
4624  25 1296  729  81 4356 2704 7744 289 900 1089 25
```

Figura 16 – Pacote *Distributed* e adição ao número de processos.

```
julia> using Distributed

julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4
```

Similar aos antecessores, aqui na Figura 17 se declara a função *quadrado* e o vetor *numbers*.

Figura 17 – Declaração de *quadrado* e do vetor *numbers*.

```
julia> @everywhere function quadrado(x::Int64)
    x*x
end

julia> numbers=rand(1:100, 1, 12)
1×12 Array{Int64,2}:
 68  5  36  27  9  66  52  88  17  30  33  5
```

Na Figura 18, se atribui ao *vetorresultado* a rotina *pmap*. Esta tendo como argumentos a função *quadrado* e os números inteiros contidos no vetor *numbers*, resultam na resposta desejada.

Figura 18 – Atribuindo *pmap* com seus argumentos ao *vetorresultado*.

```
julia> vetorresultado= pmap(quadrado, numbers)
1×12 Array{Int64,2}:
4624  25 1296  729  81 4356 2704 7744 289 900 1089 25
```

3 Mecanismos de Comunicação em Sistemas Distribuídos

3.1 Introdução a MPI

De acordo com [Gropp e Lusk \(2014, p. 13\)](#), “MPI is a library, not a language. It specifies [...] the functions to be called from C programs.”¹

MPI é um pacote da implementação de paralelismo distribuído que pode ser utilizado em C. A sigla MPI significa *Message-Passing Interface* (Interface de Passagem de Mensagens). A versão utilizada neste projeto foi MPICH, especificamente 3.3a2, uma versão mais conhecida entre suas concorrentes e, portanto, mais familiar.

Esclarecendo MPICH através de [MPICH... \(2020\)](#), “MPICH is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard [...]”².

Diferentemente do pacote de Julia Distributed, um programa MPI executa todas as instruções do programa em todos os processos até que o programa seja encerrado. Portanto, para se obter o resultado desejado, o MPI necessita que todas as instruções sejam esclarecidas pelo usuário sobre se são para serem executadas por todos ou por um grupo específico processos. Isso pode ser feito através da utilização de comandos *if*. No MPI, cada processo é diferenciado do outro pelo seu número de identificação. Assim como no *distributed* em Julia que possui o *id* para diferenciar os seus processos, o MPI em C possui *rank*, como é comumente declarado pelos usuários. Através de um *if* excluindo os *ranks* indesejados de entrarem dentro de sua seção, é possível produzir um programa com várias seções que selecionam os processos que irão executar as instruções dentro de si.

MPI possui uma tabela utilizada para definir o rótulo que deve ser inserido que depende do tipo de dado a ser enviado ou recebido. Como, por exemplo uma variável com tipo de dado *double* em C seria representada em MPI como *MPI_DOUBLE* na troca de mensagens. Esse rótulo é diferente dependendo do tipo de dado C da variável que se deseja passar para que possa ser feito o cálculo da quantidade de *bytes* que serão enviados ou recebidos na mensagem.

As chamadas operações de comunicação coletiva de MPI são uma grande vantagem pois diminuem o tempo de execução, facilitam a leitura do programa por deixá-lo mais

¹ O trecho correspondente na tradução é: “MPI é uma biblioteca, não é uma linguagem. Este especifica [...] as funções para serem chamadas por programas de C.”

² O trecho correspondente na tradução é: “MPICH é uma implementação de alto desempenho e com alta portabilidade do padrão MPI [...]”

compacto e permitem flexibilidade. Comandos estes tais quais *MPI_Bcast* e *MPI_Reduce*, que estão explicados abaixo, são executados por todos os processos e com os mesmos parâmetros, não requerendo a utilização de *if* para filtrar os processos que o executariam. Importante ressaltar que o que estes comandos fazem pode ser feito utilizando os dois comandos de comunicação ponto-a-ponto básica, *MPI_Send* e *MPI_Recv*, eles são rotinas que possuem funcionalidades muito utilizadas.

MPI lida unicamente com a troca de mensagens entre diferentes processos. Isso implica que as entidades que trocam mensagens não compartilham memória. O MPI pode ser utilizado para trocar mensagens de processos do mesmo computador ou entre processos de computadores de um mesmo *cluster* ou outra forma de estrutura de redes. Tendo isso em mente, às vezes, é importante que estes processos estejam sincronizados entre si. Existem ferramentas que podem ser utilizadas para fazer comunicações bloqueantes ou não bloqueantes. Algumas podem fazer bloqueios no meio do código impedindo os processos de continuarem seguindo até que todos estejam no mesmo ponto de execução.

Na implementação de troca de mensagens, a comunicação possui duas vertentes, a comunicação síncrona e assíncrona. A comunicação síncrona é atingida através de comandos bloqueantes, a maioria utilizada neste trabalho é de natureza síncrona. A comunicação assíncrona utiliza de rotinas não-bloqueantes e requer um cuidado maior porém permite a obtenção de maiores ganhos por possibilitar que o processo não esteja ativamente esperando o *MPI_Send*, ao invés disso, ele pode fazer computações enquanto checa constantemente a chegada do *MPI_Send*.

Existe outro argumento dentro das rotinas de comunicação que não é mencionado abaixo. Este argumento é o *MPI_TAG*. Ele permite um discernimento maior na troca de mensagens por poder ser utilizado para passagem de alguma informação secundária acerca do dado que foi passado.

3.2 Comandos de MPI

Abaixo serão esclarecidos os propósitos dos comandos pertinentes a MPI que foram utilizados no trabalho. Para mais informações sobre as rotinas abaixo descritas, referir-se a <http://www.mpich.org/static/docs/latest/>.

- *MPI_INIT*: Esta rotina serve para inicializar o ambiente MPI.
- *MPI_Comm_size*: Esta rotina retorna o número de processos que estão sendo executados no programa. Possui como parâmetros o *MPI_COMM_WORLD* que define o ambiente de comunicação utilizado e uma variável que é declarada pelo usuário para receber a quantidade de processos existentes. Nos programas deste

trabalho, esta variável foi declarada como *size*. Portanto, representa o número de processos executando o programa MPI.

- *MPI_Comm_rank*: Esta rotina retorna o número de identificação do processo que a expediu. Possui como parâmetros o *MPI_COMM_WORLD* e uma variável que é declarada pelo usuário para receber o número de identificação do processo que executou esta instrução. Nos programas deste trabalho, essa variável foi *rank*. Portanto, *rank* representa o número de identificação do processo.
- *MPI_Finalize*: Esta rotina serve para encerrar o ambiente MPI.
- *MPI_Send*: Esta rotina é utilizada para fazer o envio de mensagem por um processo remetente. Possui como seus parâmetros, em ordem, a variável que deseja ser enviada; a quantidade a ser enviada; o *MPI_Type* compatível com o tipo da variável em questão; o número de identificação do processo destinatário(o *rank* do processo destinatário); um rótulo para identificação de qual mensagem está sendo enviada e *MPI_COMM_WORLD*.
- *MPI_Recv*: Esta rotina é utilizada para fazer o recebimento de mensagem por um processo destinatário. Possui como seus parâmetros, em ordem, a variável a ser recebida; a quantidade a ser recebida; o *MPI_Type* compatível com o tipo da variável em questão; o número de identificação do processo remetente(o *rank* do processo remetente); um rótulo para identificação de qual mensagem está sendo enviada; *MPI_COMM_WORLD*; e uma variável de erro, *status*. Neste trabalho, esta variável de erro foi declarada como *status*.
- *MPI_Bcast*: Trata-se de uma rotina de comunicação coletiva que envia dados de um processo aos demais. Possui como seus parâmetros, em ordem, o endereço da variável; a quantidade a ser enviada; o *MPI_Type* compatível com o tipo da variável em questão; o *rank* do processo remetente e *MPI_COMM_WORLD*.
- *MPI_Reduce*: Trata-se de uma rotina de comunicação coletiva que reduz todos os valores em todos os processos para um único valor utilizando de operações à escolha do usuário, tratando-se de vetores, cada casa seria reduzida com as casas correspondentes dos outros processos. Dentre as operações, encontram-se: máximo, mínimo, soma e produto. Possui como seus parâmetros, em ordem, o endereço da variável sujeita a operação de *MPI_Reduce*; o endereço da variável que receberá o resultado da operação de *MPI_Reduce*; a quantidade a ser enviada; o *MPI_Type* compatível com o tipo da variável em questão; a operação de redução a ser aplicada; o *rank* do processo que receberá o resultado e *MPI_COMM_WORLD*.
- *MPI_Scatter*: Trata-se de uma rotina de comunicação coletiva que envia dados de um processo aos demais. Possui como seus parâmetros, em ordem, o endereço

da variável a ser enviada; a quantidade a ser enviada; o *MPI_Type* compatível com o tipo da variável em questão(que envia os dados); o endereço da variável que recebe os dados; a quantidade a ser recebida; o *MPI_Type* compatível com o tipo da variável em questão(que recebe os dados); o *rank* do processo remetente e *MPI_COMM_WORLD*.

- *MPI_Gather*: Trata-se de uma rotina de comunicação coletiva que junta dados de um grupo de processos em um processo escolhido. Possui como seus parâmetros, em ordem, o endereço da variável a ser enviada; a quantidade a ser enviada; o *MPI_Type* compatível com o tipo da variável em questão(que envia os dados); o endereço da variável que recebe os dados; a quantidade a ser recebida; o *MPI_Type* compatível com o tipo da variável em questão(que recebe os dados); o *rank* do processo destinatário e *MPI_COMM_WORLD*.
- *MPI_Barrier*: Esta rotina bloqueia o processo que a executou até que todos os processos a tenham executado.

Sobre algumas das operações que podem ser utilizadas em *MPI_Reduce*:

- *MPI_MAX*: Retorna o máximo entre valores.
- *MPI_MIN*: Retorna o mínimo entre os valores.
- *MPI_PROD*: Retorna o produto entre os valores.
- *MPI_SUM*: Retorna a soma entre os valores.

Além desses comandos, existem tipos de dados específicos nas rotinas de troca de mensagem para MPI. A Tabela 2 representa o tipo de dado MPI com seu semelhante em C. Os tipos de dados devem ser iguais nos envios e recebimentos, a única exceção se faz em *MPI_PACKED*, que é o tipo utilizado para manipular tipos de dados mistos. A documentação do MPI sobre constantes na Tabela 2 pode ser encontrada em: <<https://www.mpich.org/static/docs/latest/www3/Constants.html>>.

Tabela 2 – Tabela de Conversão entre tipo de dado de MPI e tipo de dado de C.

<i>MPI_Type</i>	Tipo de dado C
<i>MPI_CHAR</i>	signed char
<i>MPI_SHORT</i>	signed short int
<i>MPI_INT</i>	signed int
<i>MPI_LONG</i>	signed short int
<i>MPI_UNSIGNED_CHAR</i>	unsigned char
<i>MPI_UNSIGNED_SHORT</i>	unsigned short int
<i>MPI_UNSIGNED</i>	unsigned int
<i>MPI_UNSIGNED_LONG</i>	unsigned long int
<i>MPI_FLOAT</i>	float
<i>MPI_DOUBLE</i>	double
<i>MPI_LONG_DOUBLE</i>	long double
<i>MPI_BYTE</i>	
<i>MPI_PACKED</i>	

3.3 Exemplos de Códigos usando MPI

3.3.1 A Estrutura de um Código C com MPI

A estrutura de um programa básico de MPI é similar às demais estruturas em C. Tem-se a inclusão de *libs*, a declaração de funções, o *Main* e, dentro deste, tem o início do paralelismo com a rotina *MPI_Init*. A partir desta rotina, os demais processos executam todas as instruções presentes no código salvo a presença de um *if* que filtre esses processos ou o fim do ambiente MPI, externado por meio da rotina *MPI_Finalize*. Entre essas duas rotinas que indicam o início e o fim do ambiente MPI é onde se faz uso das rotinas apresentadas tais quais: *MPI_Send*, *MPI_Recv* e *MPI_Barrier*, além das demais formas de comunicação coletiva disponíveis para uso.

Na Figura 19, percebe-se que a estrutura de um programa C com MPI é bem similar a de um programa C. Aonde a continuidade do programa é representada por reticências, poderia-se ter qualquer uma das rotinas discutidas.

Para mostrar o uso de uma rotina *MPI_Send*, utiliza-se um exemplo, representado na Figura 20, no qual o objetivo é fazer o envio de uma variável chamada *number*. Seguindo a ordem dos argumentos, declara-se primeiro o nome da variável a ser passada, a quantidade de dados que é desejada passar, o tipo de dado MPI, o número do processo que receberá estes dados, um rótulo e o ambiente MPI que está sendo utilizado.

No exemplo da Figura 20, o objetivo é a passagem de um inteiro oriundo da variável

Figura 19 – Estrutura de um código MPI.

```
#include <mpi.h>

int main(int argc, char *argv[]) {

    // Inicializando o ambiente MPI.
    MPI_Init(&argc, &argv);
    MPI_Status status;

    // Declarando rank e size.
    int rank, size;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    .
    .
    .
    MPI_Finalize();
}
```

number para o processo 1 dentro do ambiente *MPI_Comm_World*.

Figura 20 – Rotina *MPI_Send*.

```
MPI_Send(
    /* dados          = */ &number,
    /* quantidade     = */ 1,
    /* MPI Type       = */ MPI_INT,
    /* rank destinatário = */ 1,
    /* tag(rótulo)     = */ 0,
    /* ambiente        = */ MPI_COMM_WORLD);
```

A rotina da Figura 20 precisa ser acompanhada por um *MPI_Recv* executada pelo processo 1, que é para quem se deseja passar os dados. Para isso, seguindo a ordem dos argumentos, como se pode observar na Figura 21, declaramos o nome da variável a ser recebida, a quantidade de dados que será recebida, o tipo de dado MPI, o número do processo que enviará estes dados, o rótulo, o ambiente MPI que está sendo utilizado e o *MPI_Status*, que se declarou como *status* para encurtar. Este possui a funcionalidade de fornecer dados sobre o remetente caso esteja habilitado. O *status* pode fornecer o número do processo, o rótulo e a quantidade de elementos que foram passados. Isso pode ser útil quando se estiver trabalhando com um *MPI_Recv* que esteja apto a receber dados de qualquer um dos processos, neste caso, aonde se declara o número do processo que enviará os dados, se insere *MPI_ANY_SOURCE*.

Além da comunicação ponto a ponto, tem-se a comunicação coletiva. Esta é mais utilizada pois suas rotinas são mais eficientes e abrangentes, permitindo facilidade de implementação em relação à comunicação ponto a ponto. Desta, analisaremos as rotinas *MPI_Bcast* e *MPI_Reduce*.

O *MPI_Bcast* tem por objetivo enviar para todos os demais processos de um ambiente MPI uma quantidade de dados. Seus argumentos são, em ordem, o nome da

Figura 21 – Rotina *MPI_Recv*.

```
MPI_Recv(  
    /* dados          = */ &number,  
    /* quantidade     = */ 1,  
    /* MPI Type       = */ MPI_INT,  
    /* rank remetente = */ 0,  
    /* tag            = */ 0,  
    /* ambiente       = */ MPI_COMM_WORLD,  
    /* status         = */ MPI_STATUS_IGNORE);
```

variável a ser enviada, a quantidade de dados a ser passada, o tipo de dado MPI, o número do processo que fará o envio e o ambiente MPI. A Figura 22 ilustra a declaração apropriada para uma rotina *MPI_Bcast* com o objetivo de enviar um inteiro dentro de *numbers* para os demais processos.

Figura 22 – Rotina *MPI_Bcast*.

```
MPI_Bcast(  
    /* dados          = */ &number,  
    /* quantidade     = */ 1,  
    /* MPI Type       = */ MPI_INT,  
    /* rank remetente = */ 0,  
    /* ambiente       = */ MPI_COMM_WORLD);
```

O *MPI_Reduce* faz a redução dos elementos para um só. Caso estejamos lidando com vetores, utilizando da operação de redução de *MPI_MAX*, que seleciona o valor máximo dentre os analisados para permanecer, se tratando de um vetor chamado *numbers* com dez casas populadas por números inteiros em quatro processos que possuem valores diferentes para este vetor, o resultado da operação de *MPI_Reduce* para este vetor seria um vetor de dez casas no qual cada uma dessas casas possui o valor máximo encontrado entre esses quatro processos na devida posição. Armazenando o resultado desta redução em um vetor chamado *vetorresultado*, podemos verificar essa exemplificação acima, na Figura 23.

Figura 23 – Rotina *MPI_Reduce*.

```
MPI_Reduce(  
    /* variável a ser reduzida          = */ &numbers,  
    /* variável que receberá o resultado da redução = */ &vetorresultado,  
    /* a quantidade de dados a ser enviados = */ 10,  
    /* MPI Type                         = */ MPI_INT,  
    /* operação de redução              = */ MPI_MAX,  
    /* rank remetente                   = */ 0,  
    /* ambiente                         = */ MPI_COMM_WORLD);
```

3.3.2 Exemplo Soma de Vetores

Como todo programa C, se começa com as declarações de *libs* utilizadas no programa, funções que serão empregadas e mais, como mostra a Figura 24.

Figura 24 – Vetor Soma C MPI - Declarações.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <mpi.h>
5 #include <time.h>
6 #define NUM_P 10
7
8 double func_soma(int x, int y){
9     return x+y;
10 }
11
```

Entrando em *Main*, existem muitas variáveis que são úteis para facilitar a vida do usuário quando declaradas num programa MPI. Na linha 15, da Figura 25, está a declaração de cinco variáveis das quais *rank*, *size* e *type* são usadas para guardar informações relacionadas diretamente ao MPI, como será demonstrado na Figura 27. Aqui também se declara *dif*, *ta* e *tata*, variáveis de apoio utilizadas para permitir que o programa receba qualquer número em *NUM_P*, não importando se ele é divisível pelo número de processos. Além disso, para que C gere números aleatórios de natureza menos determinística, se usa a rotina *srand* usando do tempo atual para dar *seed* nos *rands* utilizados.

Figura 25 – Vetor Soma C MPI - Entrando em *Main*.

```
12
13 int main(int argc, char *argv[])
14 {
15     int dif=0, ta, rank, size, type=99;
16     double tata;
17
18     srand(time(NULL));
19
```

Nas linhas 20 e 21, vistas na Figura 26, são declarados os vetores *vetor1* e *vetor2* que são do tipo *int* e possuem *NUM_P* elementos. Seguindo na linha 23, o *for* tem a função de popular cada casa de *vetor1* e *vetor2* com números aleatórios que variam de um a cem. Existe também a declaração de *vetorsoma*, o vetor que recebe os resultados finais da soma de *vetor1* com *vetor2*.

Figura 26 – Vetor Soma C MPI - Populando *Vetor1* e *Vetor2* com Números Aleatórios.

```
20     int vetor1[NUM_P];
21     int vetor2[NUM_P];
22
23     for(int i=0; i<NUM_P; i++){
24         vetor1[i]=rand() % 100+1;
25         vetor2[i]=rand() % 100+1;
26     }
27
28     double vetorsoma[NUM_P];
```

A inicialização do ambiente MPI ocorre na linha 30 da Figura 27. Abaixo dela, existem outras declarações destinadas a facilitar a manipulação de informações com o canal de MPI formado. Na linha 32 e 33, atribui-se a *size* e *rank* os valores de número de processos que estão sendo utilizados no ambiente de MPI e o número de identificação do processo dentro deste ambiente MPI, respectivamente. Isso significa que se todos os processos imprimirem *size* e *rank*, supondo que se tenham quatro processos no total, seria impresso em tela por todos o valor de *size* como 4 e o valor de *rank* revelaria os valores de 0, 1, 2 e 3 que seriam os números de identificação dos processos.

Figura 27 – Vetor Soma C MPI - Início do Ambiente de MPI.

```
29
30 MPI_Init(&argc,&argv);
31 MPI_Status status;
32 MPI_Comm_size(MPI_COMM_WORLD, &size);
33 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
34
35
```

Da linha 36 até a linha 39, da Figura 28 se trata de uma mensagem de erro feita para evitar que o usuário coloque um número de processos maior do que *NUM_P*.

Figura 28 – Vetor Soma C MPI - Mensagem de Erro.

```
35
36 if (NUM_P<(size)) {
37     printf("Erro: Foi entrado um valor de dados NUM_P inferior ao numero de workers.\n");
38     return 0;
39 }
```

Nem sempre o valor entrado pelo usuário como *NUM_P* possui divisão exata pelo número de processos *size*. Levando isso em consideração, as operações representadas na Figura 29 se tornam necessárias para permitir uma partilha adequada sem deixar restos não processados. Por exemplo, caso o usuário esteja trabalhando com quatro processos e queira um valor de *NUM_P* igual a 201, essa parte do código faz com que *ta*, a variável que representa a quantidade de duplas de inteiros que cada processo, incluindo o processo 0, deve passar pela função, assuma o valor de 50. Sendo assim, sobraria 1 dupla de inteiros intocada. As linhas 44 a 46, na Figura 29, atribuem o número que sobraria a uma variável chamada *dif* que é utilizada mais abaixo no código, evitando assim que não sobrem restos.

Figura 29 – Vetor Soma C MPI - Lidando com Valores de *NUM_P* Indivisíveis por *Size*.

```
41 tata=(double)NUM_P/size;
42 ta=(int)tata;
43
44 if (tata-(double)ta!=0 && rank==0) {
45     dif=NUM_P-ta*size;
46 }
47
48 double vetorprovisorio[ta];
```

Nas linhas 50 e 51, na Figura 30, se começa a medir o tempo de execução do programa. Abaixo, na linha 53 até 58 da Figura 30, o segmento contido no *if* é executado

somente pelo processo 0. Neste segmento ocorre o envio da parte que cada *worker* deverá passar pela função e depois retornar. Os inteiros contidos em *vetor1* e *vetor2* são enviados pelo processo 0 em ordem. O processo 1 recebe o primeiro intervalo *ta* de *vetor1* e *vetor2*, o processo 2 recebe o segundo intervalo e assim vai. O último intervalo *ta* e a sobra *dif* são processados pelo processo 0, como pode ser visto na Figura 32. Da linha 59 a 62, na Figura 30, todos os processos exceto o processo 0 executam as rotinas de *MPI_Recv* recebendo *ta* valores em cada um dos *MPI_Recv*.

Figura 30 – Vetor Soma C MPI - Passagem de Dados por MPI usando de *MPI_Send* e *MPI_Recv*.

```

49
50     clock_t Ticks[2];
51     Ticks[0]=clock();
52
53     if (rank==0){
54         for (int i=1; i<size; i++){
55             MPI_Send(&vetor2[(i-1)*ta], ta, MPI_INT, i, type, MPI_COMM_WORLD);
56             MPI_Send(&vetor1[(i-1)*ta], ta, MPI_INT, i, type, MPI_COMM_WORLD);
57         }
58     }
59     else{
60         MPI_Recv(&vetor2, ta, MPI_INT, 0, type, MPI_COMM_WORLD, &status);
61         MPI_Recv(&vetor1, ta, MPI_INT, 0, type, MPI_COMM_WORLD, &status);
62     }

```

Na Figura 31, existem instruções que são executadas somente pelos *workers*. Na linha 66, atribui-se a casa correspondente do vetor *vetorprovisorio* o resultado da passagem das mesmas casas dos vetores *vetor1* e *vetor2* pela função *func_soma*. Enquanto na linha 71, *ta* valores são enviados ao processo 0 através de um *MPI_Send* por cada um dos *workers*.

Figura 31 – Vetor Soma C MPI - *Workers* Processando os Valores Recebidos e os Retornando ao Processo 0.

```

64     if (rank!=0) {
65         for(int i=0; i<ta; i++){
66             vetorprovisorio[i]=func_soma(vetor1[i],vetor2[i]);
67         }
68     }
69
70     if (rank!=0) {
71         MPI_Send(&vetorprovisorio, ta, MPI_DOUBLE, 0, type, MPI_COMM_WORLD);
72     }

```

Enquanto os *workers* executam as instruções contidas na Figura 31, o processo 0 passa a sua partilha *ta* dos valores e, se houver, também o restante *dif* pela função *func_soma*, como se pode ver na Figura 32 na linha 75. Na linha 78, o processo 0 recebe dos *workers* os valores processados por eles e os armazena diretamente nas casas apropriadas dentro do *vetorsoma*. Na linha 82, na Figura 32, o ambiente MPI é encerrado.

Nas linhas 86 e 87, na Figura 33, é parado a medição e é atribuído à variável “*Tempo*”, o tempo medido desde o início na Figura 30, em milissegundos. Em 88, este valor é feito com que seja impresso em tela.

Figura 32 – Vetor Soma C MPI - Processo 0 Fazendo sua Parte e Recebendo os Valores dos *Workers*.

```

73     else{
74         for (int i = 0; i<ta+dif; i++) {
75             vetorsoma[NUM_P-ta-dif+i]=func_soma(vetor1[NUM_P-ta-dif+i],vetor2[NUM_P-ta-dif+i]);
76         }
77         for (int i = 1; i < size; i++) {
78             MPI_Recv(&vetorsoma[(i-1)*ta], ta, MPI_DOUBLE, i, type, MPI_COMM_WORLD, &status);
79         }
80     }
81
82     MPI_Finalize();
83
84

```

Figura 33 – Vetor Soma C MPI - Fim da Medida do Tempo de Execução.

```

85     if (rank==0) {
86         Ticks[1]=clock();
87         double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
88         printf("Tempo de execução:%fms\n", Tempo);
89

```

Para que ocorra a impressão em tela similar ao dos outros programas de soma de vetores, as linhas 90 a 106 na Figura 34 fazem com que sejam impressos os vetores *vetor1*, *vetor2* e *vetorsoma* em tela. Resultados estes que podem ser vistos na Figura 35, junto ao tempo de execução.

Figura 34 – Vetor Soma C MPI - Fim da Medida do Tempo de Execução.

```

89
90     printf("vetor1=");
91     for (int i = 0; i < NUM_P; i++) {
92         printf("%d ", vetor1[i]);
93     }
94     printf("\n");
95
96     printf("vetor2=");
97     for (int i = 0; i < NUM_P; i++) {
98         printf("%d ", vetor2[i]);
99     }
100    printf("\n");
101
102    printf("vetorsoma=");
103    for (int i = 0; i < NUM_P; i++) {
104        printf("%d ", (int)vetorsoma[i]);
105    }
106    printf("\n");
107
108    }
109    return 0;
110 }

```

Figura 35 – Vetor Soma C MPI - Fim da Medida do Tempo de Execução.

```

Tempo de execução:0.644000ms
vetor1=73 71 66 26 72 62 5 80 9 54
vetor2=30 93 48 40 53 95 7 75 43 4
vetorsoma=103 164 114 66 125 157 12 155 52 58

```


4 Avaliação de Desempenho

4.1 Ambiente de Execução

O ambiente de execução utilizado nos experimentos consiste de um processador Intel Core i3-6100U, 4 GB de memória RAM DDR3 com 1600MHz de frequência e 500 GB de disco rígido.

Para os testes realizados, as seguintes configurações foram utilizadas:

- Memória RAM SODIMM DDR3 1600 MHz de 4 *gigabytes*
- Processador Intel Core i3-6100U @ 2.30GHz com 3 MB Intel Smart Cache. Este processador possui 2 *cores* físicos e trabalha com 4 *threads* simultaneamente.
- Disco rígido TOSHIBA MQ01ABF0 de 456 Gib (500 GB)
- O sistema operacional: Ubuntu 18.04.02 LTS 64-bit
- A versão do *MPICH*: *MPICH* 3.3a2
- A versão de Julia: Julia v1.1.1
- A versão do *Python*: *Python* 2.7.17

O processador utilizado, Intel Core i3-6100U, possui a tecnologia de *hyperthreading*. Com esta característica, é possível se ter dois *cores* lógicos além dos dois físicos já presentes. Com isso, as medidas feitas nesse trabalho se restringem a, no máximo, quatro processos simultâneos, salvo alguns casos que serão explicados, pois tendo em vista que trabalhando com mais do que quatro processos, não se obteria ganho de desempenho.

Alguns programas de Julia possuem um desempenho melhor utilizando de um processo a mais. Esses programas possuem uma característica que parece ser uma norma em Julia que é a de computação ser feita somente pelos *workers*. O *distributed.jl* por exemplo, devido a arquitetura da rotina *@distributed*, não utiliza o processo 1 para fazer computação, ele utiliza exclusivamente os *workers*. Isso, quando comparado a um programa MPI C, que utiliza todos, o colocaria em desvantagem caso fosse fixado o número de processos a serem utilizados, portanto utilizou-se 1 processo a mais para permitir o uso de todos os *cores* físicos e lógicos na computação. Para a medida dos programas que utilizam de 2 processos em Julia, no qual foram usados 3 processos, foram desligados os *cores* lógicos para processamento e, então feitas as medidas do tempo de execução com 3 processos no total.

4.2 Programas de Teste

As medições são feitas da seguinte forma para as linguagens:

- Para C se tem um *while* que repete 20 vezes o intervalo de código responsável pela computação que se deseja medir. Esse *while* tem o seu tempo de execução medido e armazenado em uma variável chamada de Tempo. Depois do fim do *loop while*, é impresso o valor de Tempo dividido por 20, conseguindo-se a média de tempo de uma execução. Além disso, foi utilizado o parâmetro de otimização de *gcc -O3*.
- Para Julia se utiliza o macro *@benchmarkable* encapsulando a computação que se deseja medir. Usando o parâmetro *samples* se regula o número de medições feitas para 20. Para uma medição com um ruído reduzido, são modificados os parâmetros *memory_tolerance* e *time_tolerance* para 0,01. Esses parâmetros exigem que os valores medidos não desviem mais do que uma certa porcentagem da média de tempo e de memória encontrados nessas 20 repetições. Não foi utilizado o parâmetro de otimização -O3 pois ele produz aproximadamente o mesmo resultado ou um resultado um pouco mais lento. A forma de medição em Julia também varia dependendo do programa. Alguns deles obtêm ganho de desempenho ao empregar cinco processos.

4.2.1 Programas de Julia

4.2.1.1 Análise do Código.

O que o programa faz.

Este programa faz a distribuição de *NUM_P* elementos dos vetores *vetorg* e *vetorp* para um número qualquer de *workers* utilizando *@spawnat*. Esses elementos são passados por uma função nesses *workers* e são retornados ao processo 1.

A razão de ter sido feito este programa:

A meta foi uma forma de resolução da proposta com o comando mais simples o possível, tal qual o *Send/Recv*.

Analisando o Código.

Analisando o código da Listagem 4.1. Na linha 6, é declarada a constante *NUM_P*. Na linha 8, a função *popula_vetor* é declarada. Esta função foi utilizada para poder declarar o *array* retornado por ela como do tipo *Array{Int8}*, caso fosse declarado sem ser usada uma função, o *rand*, para retornar um *array* de números aleatórios do intervalo de 1 a 100, retorna ao usuário o tipo *Array{Int64}*, que ocupa muito mais espaço na alocação de memória. Na linha 11, a constante *ta* é declarada. *ta* é a divisão sem restos de *NUM_P* por *nworkers* que é o número de *workers*. Na linha 12, a função *calcpint* é declarada usando o macro *@everywhere*, permitindo que os *workers* tenham acesso a essa função.

Da linha 15 a 18, caso NUM_P dividido por $nworkers$ seja uma divisão inexata, a constante dif assume o valor do resto desta divisão para que o processo 1 possa computar estes elementos mais abaixo no código. Nas linhas 19 e 20, os vetores $vetorg$ e $vetorp$ são populados através da função `popula_vetor`.

O intervalo do código entre as linhas 22 e 34 é medido utilizando a rotina `@benchmarkable`. Na linha 24 é declarado R e $R[1]$ até $R[nworkers()]$ que são *futures* que podem ser resgatados. Por exemplo, chamando $R[1][1]$ é acessado o primeiro *future* disponível no vetor R e, dentro dele, o primeiro elemento do primeiro vetor. Fazendo $R[1][1:ta]$ é acessado o primeiro vetor do seu primeiro elemento até o último elemento do intervalo em que o *worker* trabalhou. Nas linhas 25 e 26, caso a divisão de NUM_P por $nworkers$ seja inexata, o resto é computado aqui pelo processo 1 e armazenado em *buffer*. Nas linhas 29 e 30, o processo 1 recupera os valores já computados dos *workers* e os armazena, em ordem, em *vetorresults*. Na linha 32, os resultados da computação dos restos armazenados em *buffer* são colocados no fim do vetor *vetorresults*. Na linha 35, o programa se encerra.

```

1 using BenchmarkTools
2 using Distributed
3 using Statistics
4 using Random
5 addprocs(4)
6 const NUM_P=1000*10000
7 Random.seed!(1234)
8 function popula_vetor()::Array{Int8}
9     return rand(1:100, NUM_P)
10 end
11 const ta = floor{Int32, NUM_P/nworkers()}
12 @everywhere function calcpint(x,y::Int8)::Float64
13     return (sin(cos(x))*(tan(sin(cos(y)))))
14 end
15 if (NUM_P/nworkers()-floor{Int32, NUM_P/nworkers()}!=0)
16     const dif=NUM_P-ta*nworkers()
17 else const dif=0
18 end
19 vetorp=popula_vetor()
20 vetorg=popula_vetor()
21
22 b= @benchmarkable begin
23     buffer=zeros(0)
24     R=[@spawnat i map(calcpint, vetorp[ta*(myid()-2)+1:ta*(myid()-1)],
25         vetorg[ta*(myid()-2)+1:ta*(myid()-1)]) for i=2:nprocs()]
26     if dif!=0
27         append!(buffer, (map(calcpint, vetorp[NUM_P-dif+1:NUM_P], vetorg[NUM_P-
28             -dif+1:NUM_P])))
29     end
30     vetorresults=zeros{Float64, 0}

```

```

29 for i in 2:nprocs()
30     append!(vetorresults, R[i-1][1:ta])
31 end
32 append!(vetorresults, buffer)
33     return vetorresults
34 end seconds=1500 samples=10 time_tolerance=0.01 memory_tolerance=0.01
35 println(mean(run(b)))

```

Listing 4.1 – @spawnat.jl

Comentários Adicionais.

Neste programa foram utilizados 5 processos no total. Isso ocorre pois o código distribui os elementos somente para os *workers* e somente estes fazem computação mais dispendiosa. Com 5 processos, 4 deles sendo *workers*, o processo mestre executa as instruções e distribui os elementos a serem computados e depois, ao invés de ficar ocupando um core com uma espera ativa, é substituído por outro processo que passa a fazer sua parte da computação. Com *NUM_P* valendo 10,000,000, usando 5 processos tomou 120ms a menos para executar do que com 4.

4.2.1.2 Análise do Código *pmap.jl*

O que o programa faz.

Este programa faz a distribuição de *NUM_P* elementos dos arrays *vetorg* e *vetorp* para um número qualquer de *workers* utilizando *pmap*. Esses elementos são processados e retornados ao processo 1.

A razão de ter sido feito este programa:

Usar a rotina *pmap* para fazer uma versão de comunicação coletiva do código *@spawnat.jl*, bem como servir como comparativo.

Analisando o Código.

Analisando o Código da Listagem 4.2. Na linha 17, o código distribui os elementos de *vetorg* e *vetorp*, passa estes elementos pela função *calcpint* e os retorna para o processo 1. A rotina *benchmarkable* serve para marcar o intervalo de medição de tempo que, no caso, é só uma linha.

```

1 using BenchmarkTools
2 using Distributed
3 using Statistics
4 using Random
5 addprocs(3)
6 const NUM_P=1000*1000
7 Random.seed!(1234)
8 function popula_vetor()::Array{Int8}

```

```
9     return rand(1:100, NUM_P)
10 end
11 @everywhere function calcpint(x,y::Int8)::Float64
12     return (sin(cos(x))*(tan(sin(cos(y)))))
13 end
14 const vetorp=popula_vetor()
15 const vetorg=popula_vetor()
16
17 b=@benchmarkable pmap(calcpint, vetorg, vetorp) seconds=1500 samples=20
    time_tolerance=0.01 memory_tolerance=0.01
18 println(mean(run(b)))
```

Listing 4.2 – pmap.jl

Comentários Adicionais.

pmap é estruturado para o caso em que cada chamada de função faz uma quantia enorme de trabalho, não sendo esse o caso desse programa, já que são feitas várias chamadas para quantias pequenas de trabalho, *pmap* parece notavelmente inferior aos demais, ao menos nesta forma de trabalho. Notando os valores altos resultantes da medida de *pmap*, alterou-se o número de *workers* disponíveis. O tempo de execução medido com 1 único processo foi o melhor tempo obtido. O tempo de execução medido com 2 processos, 1 processo mestre e 1 *worker* aumentou o tempo de execução substancialmente. O tempo de execução medido com 3, 4 e 5 processos foi quase igual. Devido a essas medidas, acredita-se que o *overhead* de distribuição de valores é muito alto. O *pmap* gerencia os *workers* que ele utiliza, o que torna interessante o fato do tempo de execução com 2, 3 e 4 *workers* ser quase igual, pois pode significar que o *pmap* calculou que seria desvantajoso ter que distribuir para ainda mais 1 *worker*.

Mais um experimento, dessa vez com os elementos armazenados em *SharedArrays*, provou que a ideia de um grande *overhead* na distribuição de valores estar incorreta, pois mesmo tendo os valores de *vetorg* e *vetorp* já disponíveis, o tempo de execução foi o mesmo, mostrando que a arquitetura de *pmap* simplesmente não é adequada para este tipo de problema.

4.2.1.3 Análise do Código *distributed-sharedarrays.jl*

O que o programa faz.

Este programa faz a distribuição de *NUM_P* elementos dos *arrays* *vetorg* e *vetorp* para um número qualquer de *workers* utilizando *@distributed*. Esses elementos são processados e retornados ao processo 1, aonde são todos somados e reduzidos a um único valor.

A razão de ter sido feito este programa:

A meta foi a de utilizar a rotina *@distributed* para fazer computação paralela e testar os retornos obtidos com os operadores que ele possui.

Analisando o Código.

Analisando o Código da Listagem 4.3. Na linha 4, é declarado o uso de *SharedArrays* no programa. Na linha 17, o vetor *vetorresults* é declarado como do tipo *SharedArray* com tamanho *NUM_P*. Utiliza-se *SharedArrays* para que possa ser armazenado o resultado final em um *SharedArray* que é *vetorresults*, permitindo que sejam feitas alterações diretamente nesse vetor dentro dos *workers*. Isso faz com que não seja necessário voltar com os valores para o processo 1 com a finalidade de colocá-los num vetor de resultados.

Na linha 19, o tempo começa a ser medido por *@benchmarkable*. Na linha 20, *@distributed* partilha o *for* entre os *workers*, incluindo os dados que eles necessitam. A rotina *@distributed*, neste caso, é utilizada com o operador de redução “+”, fazendo com que o somatório dos elementos computados seja retornado ao usuário. O *@sync* serve para esperar as *tasks* independentes terminarem antes de prosseguir, permitindo uma medida de tempo mais fiel.

```

1 using BenchmarkTools
2 using Distributed
3 using Statistics
4 using SharedArrays
5 using Random
6 Random.seed!(1234)
7 addprocs(4)
8 const NUM_P=1000*10000
9 function popula_vetor()::Array{Int8}
10     return rand(1:100, NUM_P)
11 end
12 @everywhere function calcpint(x,y::Int8)::Float64
13     return (sin(cos(x))*(tan(sin(cos(y)))))
14 end
15 const vetorp=popula_vetor()
16 const vetorg=popula_vetor()
17 vetorresults=SharedArray{Float64}(NUM_P)
18
19 b=@benchmarkable begin
20     @sync @distributed (+) for i in 1:NUM_P
21         vetorresults[i]=calcpint(vetorp[i],vetorg[i])
22     end
23 end memory_tolerance=0.01 time_tolerance=0.01 seconds=1500 samples=20
24 println(median(run(b)))

```

Listing 4.3 – distributed-sharedarrays.jl

Comentários Adicionais.

@distributed aliado a *SharedArrays* acabou sendo uma das implementações mais simples que mantiveram um tempo de execução baixo em Julia. Como o *@distributed* é programado para utilizar somente os *workers* na execução, utilizou-se 5 processos no total, 4 deles *workers*. Assim evitando uma espera ativa por parte do processo mestre, já que este ocupa um dos cores. Esse código foi simples e apresentou um desempenho superior ao código de *@spawnat*, que teve implementação mais convoluta.

4.2.1.4 Análise do Código *remote-do-sharedarrays.jl*

O que o programa faz.

Este programa disponibiliza os vetores de dados *vetorg* e *vetorp* para os *workers* através do uso de *SharedArrays* e, dependendo se o *myid* do processo, executa uma operação específica. No final, soma todos os resultados através de *map* para um único *vetorresultado* no processo 1.

A razão de ter sido feito este programa:

O objetivo foi o de utilizar *RemoteChannels* para armazenar informação, usar *SharedArrays* para compartilhar dados e testar a viabilidade do uso de uma variável global como recipiente do resultado final. Além disso, utilizar a rotina *remote_do* e encontrar formas diferentes de se recuperar seu resultado, já que, pela arquitetura desta rotina, ela não retorna um *future* que pode ser recuperado ou o resultado da computação.

Analisando o Código.

Analisando o Código da Listagem 4.4. As mesmas *libs* foram utilizadas, foram adicionados 3 *workers* e declarada a constante *NUM_P*. Na linha 9, declara-se *vetorresults* como uma variável global, permitindo sua alteração dentro de *loops for*. Nas linhas 11 e 12, são abertos um *RemoteChannel* para cada um dos processos utilizados, para permitindo o armazenamento dos resultados depois de serem passados os dados pelas funções.

As funções declaradas nas linhas 17 e 20 são as funções pelas quais os dados serão passados.

```

1 using BenchmarkTools
2 using Distributed
3 addprocs(4)
4 using Random
5 using Statistics
6 Random.seed!(1234)
7 using SharedArrays
8 const NUM_P=1000*10000
9 global vetorresults
10 const vetordezeros=zeros(Float64, NUM_P)
11 for p in procs()
12     @fetchfrom p global inbox = RemoteChannel{()}->Channel{()}(1)
13 end
14 function popula_vetor()::Array{Int8}
15     return rand(1:100, NUM_P)
16 end
17 @everywhere function calc_real_p_par(x,y::Int8)::Float64
18     return (sin(cos(x))*(tan(sin(cos(y))))))
19 end
20 @everywhere function calc_real_p_impar(x,y::Int8)::Float64

```

```

21 return (sin(cos(x))*(tan(sin(cos(y))))*1/tan(sin(cos(x*y)))
22 end

```

Listing 4.4 – remote-do-sharedarrays.jl

Analisando agora a Listagem 4.5. A função declarada na linha 1 serve ao propósito de encapsular as funções *calc_real_p_par* e *calc_real_p_impar*, enquanto permite a passagem dos argumentos *X* e *Y*, que, apesar de estarem disponíveis nos *workers* por se tratarem de *SharedArrays*, não é possível acessá-los sem tê-los passado como argumentos, já que os *workers* não possuem os rótulos dos *SharedArrays* em questão. Aqui também, dependendo do *myid* do processo, executa uma das duas funções.

Devido a dificuldades na medição de tempo correta decorrentes da natureza do programa, tornou-se necessário utilizar a rotina *@benchmarkable* que funciona melhor tendo uma só função para gerenciar. Por esse motivo, foi usada mais uma camada de função para encapsular as operações necessárias, essa função foi declarada na linha 8. Nas linhas 9 e 10, ocorrem as chamadas aos processos para passar *X* e *Y* pelas funções. É preciso utilizar de *RemoteChannels* para guardar os resultados da computação, pois *remote_do* não retorna nenhum valor ou *future*. Devido a esta característica, não é preciso fazer uma operação de *fetch* ou esperar o valor ser retornado, mas também impede que o valor seja recuperado destas formas. Portanto, uma das soluções para este problema foi a adoção do uso de *RemoteChannels*. É utilizado o pacote *SharedArrays* para declarar *X* e *Y* pois todos os processos utilizam seus valores. Na linha 11, são recuperados os vetores resultados da aplicação das funções que estavam armazenados nos *RemoteChannels* em cada um dos *processos*, permitindo o uso do vetor *vetorresults* para somar os resultados de todos os *procs* em um só vetor.

```

1 @everywhere function work(X, Y)
2   if (myid()/2)-floor(myid()/2)!=0
3     put!(inbox, map(calc_real_p_impar, X, Y))
4   else
5     put!(inbox, map(calc_real_p_par, X, Y))
6   end
7 end
8 function paralelizar(x,y::SharedArray)
9   for p in procs()
10    @async remote_do(work, p, x, y)
11    g=@fetchfrom p take!(inbox)
12    global vetorresults=map(+, vetorresults, g)
13  end
14 end

```

Listing 4.5 – remote-do-sharedarrays.jl

Analisando agora a Listagem 4.6. Nas linhas 1 e 2, são iniciados *X* e *Y* como *SharedArrays*, disponibilizando-os em todos os processos. Nas linha 3 e 4, os vetores *vetorg*

e *vetorp* são populados com números aleatórios usando a função *popula_vetor*. Na linha 6, a rotina *@benchmarkable* se tornou necessária para fazer a medida de tempo, pois para que os valores de *vetorresults* não fossem modificados com cada uma das *samples* feitas pelo pacote *BenchmarkTools* para medir melhor o tempo, se torna preciso voltar com os valores dentro do vetor para 0, caso contrário o resultado de *vetorresults* iria se acumulando. O valor de *vetorresults* é mantido correto devido ao *setup* que atribui seu valor para o mesmo de *vetordezeros*, que, como o nome implica, é um vetor populado por *NUM_P* zeros. Entre as linhas 7 e 10 são feitas as passagens de valores dos vetores *vetorg* e *vetorp* para os *SharedArrays* *X* e *Y* através do uso da rotina *@distributed*. A função paralelizar é chamada na linha 11 e na linha 12 é encerrada medição de tempo do *@benchmarkable*.

```

1 X=SharedArray{Int8}(1, NUM_P)
2 Y=SharedArray{Int8}(1, NUM_P)
3 vetorp=popula_vetor()
4 vetorg=popula_vetor()
5
6 b=@benchmarkable begin setup=(global vetorresults=vetordezeros)
7   @distributed for i=1:NUM_P
8     X[1,i]=vetorp[i]
9     Y[1,i]=vetorg[i]
10  end
11  paralelizar(X,Y)
12 end memory_tolerance=0.01 time_tolerance=0.01 seconds=1500 samples=20
13 println(mean(run(b)))

```

Listing 4.6 – remote-do-sharedarrays.jl

Comentários Adicionais.

Este código, similar ao código *bcast-reduce.c* de C, lida com *procs * NUM_P* cálculos devido à sua natureza. Como todos os processos devem fazer seus próprios cálculos com todas as variáveis disponíveis, o tempo de execução desse código é melhor avaliado quando comparado com seu semelhante em C, *bcast-reduce.c*, pois a computação feita pelo programa muda dependendo do número de processos. Utilizou-se *@distributed* nesse código somente para distribuir os valores de *vetorg* e *vetorp* para os *SharedArrays* *X* e *Y*.

4.2.1.5 Análise do Código *@spawnat-remotechannels.jl*

O que o programa faz.

O programa gera um número *NUM_P* de vetores de *N* casas populadas com variáveis aleatórias para cada um dos dois vetores: *vetorg* e *vetorp*. Esses vetores são gerados no processo 1 e são recuperados pelos demais processos através do *take!* de *RemoteChannels*, sendo extraídos em forma de uma tupla. Os *workers* passam os vetores pela função *calcpint* usando a rotina *map*. Após terem sido passados pelo *work*, os valores encontrados são colocados num segundo *RemoteChannel*, chamado *results*, com objetivo de serem recuperados pelo processo 1 para armazenamento no vetor de resultados *vetorresults*.

A razão de ter sido feito este programa:

Este programa tem como objetivo utilizar *RemoteChannels* como principal fonte de passagem de informação. Ele usa de *RemoteChannels* e *@spawnat* juntos para gerar um pipeline. Utilizando tanto de co-rotinas quanto de paralelismo distribuído para descobrir a viabilidade da utilização de ambos como um conjunto. Averiguar a viabilidade da produção de dados concorrente ao seu consumo com essas ferramentas.

Analisando o Código.

Analisando o código da Listagem 4.7. Nas primeiras linhas estão as declarações de pacotes e constantes. São declaradas as funções já estabelecidas de *calcpint* e *popula_vetor*. Além de serem iniciados os *RemoteChannels* nas linhas 10 e 11.

```

1 using BenchmarkTools
2 using Distributed
3 using Statistics
4 using Random
5 addprocs(4)
6 Random.seed!(1234)
7 const NUM_P=10000
8 const N=1000
9 const ta = floor{Int32, NUM_P/nworkers()}
10 const inbox = RemoteChannel{()->Channel{}}(300)
11 const results = RemoteChannel{()->Channel{}}(300)
12 function popula_vetor()::Array{Int8}
13     return rand(1:100, N)
14 end
15 @everywhere function calcpint(x,y::Int8)::Float64
16     return (sin(cos(x))*(tan(sin(cos(y)))))
17 end

```

Listing 4.7 – *@spawnat-remotechannels.jl*

Analisando o código da Listagem 4.8. A função *work*, que tem como objetivo servir como um encapsulamento para os argumentos e funções envolvidas, é explicitada na linha

1. Nas linhas 2 e 3, o *while* e o *wait* utilizados fazem com que os *workers* não realizem uma operação de *take!* se o canal remoto *inbox* estiver vazio. Chamadas *take!* com o canal alvo vazio resultam em erros, portanto são evitadas dessa forma. Na linha 5, após verificar que o canal remoto não está vazio, o *worker* pega uma parcela de trabalho com o *take!*. Feito isso, na linha 6, o *worker* passa esta parcela obtida pela função *calcpint* e a armazena, junto com o número de ordenação do vetor, no caso *b[3]*, no canal remoto *results*.

```

1 @everywhere function work(inbox, results)
2   while !isready(inbox)
3     wait(inbox)
4   end
5   b=take!(inbox)
6   put!(results, (map(calcpint, b[1],b[2]),b[3]))
7 end

```

Listing 4.8 – @spawnat-remotechannels.jl

Analisando o código da Listagem 4.9. A função *englobe* é declarada na linha 1. Esta é mais um encapsulamento para facilitar a medida do tempo usando o pacote *BenchmarkTools*. Ela possui duas partes principais, a primeira, trata dos consumidores e vai da linha 2 até a linha 13. Nesta seção, como pode ser visto nas linhas 2, 3 e 4, o processo 1 faz com que os *workers* iniciem o consumo de dados através da função *work*. Isso é iniciado antes da produção, caso contrário, se teria um *deadlock* com o processo 1, pois este chegaria ao limite do canal remoto *inbox* e ficaria esperando um *take!* que nunca aconteceria para liberar espaço. Entre o intervalo das linhas 7 e 10, caso a divisão de *NUM_P* por *workers* seja inexata, o último *worker* passa o resto da divisão por *calcpint* e o armazena em *results*.

Ainda dentro da função *englobe*, agora dentro da seção que trata da produção e coleta de dados feitas pelo processo 1, da linha 14 a 27, são populados os vetores *vetorg* e *vetorp*, bem como o seu armazenamento no canal remoto *inbox* como uma tupla, onde estes aguardam para serem pegos pelos *workers* via a rotina *take!*. O motivo do *i* deste *loop for* ser de *0:NUM_P-1* se faz pela necessidade matemática da linha 25. Para que os valores mantenham a ordem depois de serem passados aos *workers*, o *i* de produção destes precisa ser enviado junto, como pode ser visto na linha 17. Depois de passados pela função *calcpint* e recuperados pelo processo 1, eles precisam ser colocados no intervalo de casas correto do vetor *vetorresults*. Para isso, é utilizado o *i*, na linha 25 representado por *buffer[2]* por estar lá armazenado. Das linhas 19 a 27, o processo 1 recupera os valores computados pelos *workers* armazenados no canal remoto *results* para colocá-los em ordem no vetor *vetorresults*. Nas linhas 20 e 21, o processo 1 aguarda a disponibilidade de valores dentro do *RemoteChannel results*. Na linha 23, estes valores são pegos do canal e utilizados no *loop for* nas linhas 24 e 25. Aonde são colocados em suas posições corretas de acordo com o *i* e *N*.

```

1 function englobe(inbox, results)
2   for i in 1:ta
3     for p in workers()
4       R=[@spawnat p work(inbox, results)]
5     end
6   end
7   if dif!=0
8     @spawnat nprocs() begin
9       for i in 1:dif
10        work(inbox, results)
11      end
12    end
13  end
14  for i in 0:NUM_P-1
15    vetorp=popula_vetor()
16    vetorg=popula_vetor()
17    put!(inbox, (vetorp, vetorg, i))
18  end
19  for i in 1:NUM_P
20    while !isready(results)
21      wait(results)
22    end
23    buffer=take!(results)
24    for j in 1:N
25      vetorresults [buffer[2]*N+j]=buffer[1][j]
26    end
27  end
28 end

```

Listing 4.9 – @spawnat-remotechannels.jl

Analisando o código da Listagem 4.10. Nas linhas 1, 2 e 3, se a divisão de NUM_P por $nworkers$ for inexata, a constante declarada *dif* recebe o valor do resto dessa divisão. Na linha 5, o vetor *vetorresults* é populado por zeros e possui as dimensões de NUM_P multiplicado por N . Na linha 7, é atribuído a *b* o *@benchmarkable* da função *englobe* que é executado na linha 8.

```

1 if (NUM_P/nworkers()-floor(Int32, NUM_P/nworkers()))!=0)
2   const dif=NUM_P-ta*nworkers()
3 else const dif=0
4 end
5 vetorresults=zeros(Float64, NUM_P*N)
6
7 b=@benchmarkable englobe(inbox, results) memory_tolerance=0.01
   time_tolerance=0.01 seconds=1500 samples=20
8 println(mean(run(b)))

```

Listing 4.10 – @spawnat-remotechannels.jl

Comentários Adicionais.

Este código é bem diferenciado dos demais apresentados neste trabalho, com exceção do *simulacao-remotechannels.c* de C que teve como objetivo ser uma representação deste código em outra linguagem, por ser o único que gera os valores ao mesmo tempo que os processa. É interessante notar também que este programa utilizou 5 processos por ter mostrado um desempenho um pouco melhor. Este programa foi testado variando o N e o NUM_P .

A comparação destes resultados indica que a alteração do valor de N mostrou um desempenho bem superior. Quanto maior o valor de N , menor o número de vezes que o processo mestre precisa se comunicar com o *worker* para passar os vetores e menor o número de vezes que este *worker* precisa se comunicar com o processo mestre para entregar os resultados da sua computação.

4.2.2 Programas de C

4.2.2.1 Análise do Código *send-receive.c*

O que o programa faz.

Este programa faz a distribuição de NUM_P elementos dos vetores *vetorg* e *vetorp* para *size* processos utilizando *MPI_Send* e *MPI_Recv*. Esses elementos são passados pela função *calc_real_p* por cada dos processos e são então retornados ao processo mestre através do *MPI_Send* e *MPI_Recv*.

A razão de ter sido feito este programa:

A meta foi a de demonstrar as dificuldades de se fazer um programa MPI com processamento paralelo utilizando dos comandos básicos de MPI, *MPI_Send* e *MPI_Recv*. Possuir uma comparação entre comunicação MPI ponto a ponto com comunicação MPI coletiva e compará-lo com o código com objetivo semelhante de Julia, *@spawnat.jl*.

Analisando o Código.

Analisando o código da Listagem 4.11. As *libs* utilizadas são adicionadas e a função *calc_real_p* é definida. Essa função faz um cálculo qualquer, ela poderia ser substituída por uma outra função que use também dois inteiros aleatórios e não haveria problema. São declaradas as variáveis utilizadas no programa e aloca-se memória para os ponteiros dos vetores *vetorg*, *vetorp* e *vetorresultado*. Iniciado o ambiente MPI, seguindo para a atribuição das variáveis auxiliares recebendo os dados MPI, tem-se, nas linhas 21, 22 e 23, a população dos vetores *vetorg* e *vetorp* através do uso de um *for* e da rotina *rand*.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <mpi.h>
5 #include <time.h>
6 #define NUM_P 1000*1000
7 double calc_real_p(short int x, short int y){
8     return (sin(cos(x))*(tan(sin(cos(y)))));}
9 int main(int argc, char** argv){
10     int dif=0, ta, rank, size, type=99;
11     double tata;
12     srand(time(NULL));
13     short int *vetorp=(short int*)malloc(NUM_P*sizeof(short int));
14     short int *vetorg=(short int*)malloc(NUM_P*sizeof(short int));
15     double *vetorresultado=(double*)malloc(NUM_P*sizeof(double));
16     MPI_Init(&argc, &argv);
17     MPI_Status status;
18     MPI_Comm_size(MPI_COMM_WORLD, &size);
19     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20     if (rank==0) {

```

```

21  for(int i=0; i<NUM_P; i++){
22      vetorp[i]=rand() % 100+1;
23      vetorg[i]=rand() % 100+1;}
24  }

```

Listing 4.11 – send-receive.c

Analisando o código da Listagem 4.12. Utilizando de *MPI_Send* e *MPI_Recv* é necessário tomar cuidado com a quantidade de variáveis que se está distribuindo. O devido cuidado é tomado da linha 5 a 8, onde é analisado o valor resultado da divisão de *NUM_P* por *size*. Caso este valor não seja um inteiro, é necessário fazer algumas operações para que o resto da divisão seja propriamente processado. É começada a medida de tempo nas linhas 11 e 12, antes do início do *loop while*.

```

1  if (NUM_P<(size)) {
2      printf("Erro: Foi entrado um valor de dados NUM_P inferior ao numero
          de workers.\n");
3      return 0;}
4
5  tata=(double)NUM_P/size;
6  ta=(int)tata;
7  if (tata-(double)ta!=0 && rank==0) {
8      dif=NUM_P-ta*size;}
9  double *vetorprovisorio=(double*)malloc(ta*sizeof(double));
10 int repeticao=20;
11 clock_t Ticks[2];
12 Ticks[0]=clock();

```

Listing 4.12 – send-receive.c

Analisando o código da Listagem 4.13. Na linha 1, o *loop while* que é o intervalo de medição começa, este *loop* é executado 20 vezes para medição de tempo mais fiel e é reproduzido em todos os programas de C. Nas linhas 6 e 7, o processo 0 faz o envio das parcelas de dados dos demais processos usando a rotina *MPI_Send* dentro de um *for*. Nas linhas 10 e 11, os demais processos recebem individualmente tais parcelas através da rotina *MPI_Recv*. Na linha 14, dentro de um *for*, os *workers* passam os valores recebidos por dentro da função *calc_real_p* e armazenam os resultados desses cálculos dentro da mesma casa de um vetor chamado *vetorprovisorio*. O processo 0 calcula e computa a sua parcela de dados nas linhas 19 e 20. Caso a divisão de *NUM_P* por *size* não seja um inteiro, ele calcula este restante e armazena o resultado de tudo em *vetorresultado*. Na linha 17 os *workers* enviam o conteúdo de seus vetores *vetorprovisorio* para o processo 0, que, na linha 22, os recebe cada um em seu devido lugar no *vetorresultado*, agora completo.

```

1  while (repeticao>0) {
2      repeticao-=1;
3

```

```

4  if (rank==0){
5      for (int i=1; i<size; i++){
6          MPI_Send(&vetorg[(i-1)*ta], ta, MPI_SHORT, i, type, MPI_COMM_WORLD);
7          MPI_Send(&vetorp[(i-1)*ta], ta, MPI_SHORT, i, type, MPI_COMM_WORLD)
            ;}
8      }
9      else{
10         MPI_Recv(vetorg, ta, MPI_SHORT, 0, type, MPI_COMM_WORLD, &status);
11         MPI_Recv(vetorp, ta, MPI_SHORT, 0, type, MPI_COMM_WORLD, &status);}
12     if (rank!=0) {
13         for(int i=0; i<ta; i++){
14             vetorprovisorio[i]=calc_real_p(vetorp[i],vetorg[i]);}
15     }
16     if (rank!=0) {
17         MPI_Send(vetorprovisorio, ta, MPI_DOUBLE, 0, type, MPI_COMM_WORLD);}
18     else{
19         for (int i = 0; i<ta+dif; i++) {
20             vetorresultado[NUM_P-ta-dif+i]=calc_real_p(vetorp[NUM_P-ta-dif+i],
21                 vetorg[NUM_P-ta-dif+i]);}
22         for (int i = 1; i < size; i++) {
23             MPI_Recv(&vetorresultado[(i-1)*ta], ta, MPI_DOUBLE, i, type,
24                 MPI_COMM_WORLD, &status);}
25     }
26 }

```

Listing 4.13 – send-receive.c

Analisando o código da Listagem 4.14. Na linha 1, o ambiente MPI é encerrado. Na linha 2 e 3, a medida de tempo é terminada. Na linha 5, o tempo de execução dividido por 20 é impresso em tela. Na linha 6 há o *return*, encerrando o programa.

```

1  MPI_Finalize();
2  Ticks[1]=clock();
3  double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
4  if (rank==0) {
5      printf("Tempo de execução médio:%fms\n", Tempo/20);}
6  return 0;
7  }

```

Listing 4.14 – send-receive.c

Comentários Adicionais.

Este programa foi um dos mais complexos por não ter adotado nenhuma forma de comunicação coletiva, que era seu objetivo. Ele mostra com sucesso os desafios e complicações de se abster do uso das rotinas de comunicação coletiva bem como a recompensa, que são tempos de execução melhores, apesar de que a diferença é pequena em relação ao *scatter-gather.c*.

4.2.2.2 Análise do Código *scatter-gather.c*

O que o programa faz.

Este programa faz a distribuição de NUM_P elementos dos vetores *vetorg* e *vetorp* para *size* processos utilizando *MPI_Scatter*. Esses elementos são processados e retornados ao processo mestre através do *MPI_Gather*.

A razão de ter sido feito este programa:

Utilizar a rotina *MPI_Gather* como forma de retornar os valores passados pelas funções ao processo mestre junto ao *MPI_Scatter*. Ter seu tempo de execução utilizado para comparar com o código de *Send/Recv* só que utilizando rotinas de comunicação coletiva ao invés de ponto a ponto, e poder ser comparado com o código com funcionalidade similar a ele em Julia, *pmap.jl*.

Analisando o Código.

Analisando o código da Listagem 4.15. As *libs*, declarações de variáveis auxiliares e função utilizadas são as mesmas do código *Send/Recv*. A declaração de vetores deste código adiciona dois novos vetores, *vetorgg* e *vetorpp*, declarados nas linhas 1 e 2.

```

1  short int *vetorpp=(short int*)malloc((ta)*sizeof(short int));
2  short int *vetorgg=(short int*)malloc((ta)*sizeof(short int));
3  double *vetorresultado=(double*)malloc(NUM_P*sizeof(double));
4  int repeticao=20;
5  clock_t Ticks[2];
6  Ticks[0]=clock();

```

Listing 4.15 – *scatter-gather.c*

Analisando o código da Listagem 4.16. Já com a medida de tempo contando e dentro do *loop while* de medição, nas linhas 3 e 4, o processo *size-1* utiliza a rotina *MPI_Scatter* para enviar as parcelas de dados *ta* referentes a cada um dos processos, inclusive para si mesmo, que são recebidas nos vetores *vetorgg* e *vetorpp*. No caso, *vetorgg* recebe os *ta* elementos de *vetorg* enquanto *vetorpp* recebe os *ta* elementos de *vetorp*. Nas linhas 5 e 6, os valores agora em *vetorgg* e *vetorpp* são passados pela função *calc_real_p* e armazenados nos vetores *vetorprovisorio* de cada processo. A necessidade da existência de *vetorgg* e *vetorpp* se deve ao fato de que se o processo mestre utilizasse do *vetorg* e *vetorp* como destino bem como origem da rotina *MPI_Scatter* isso seria uma operação ilegal.

Nas linhas 8 e 9, são tratados dos restos da divisão de NUM_P por *size-1*. Estes restos(*dif*) são passados pela função *calc_real_p* e diretamente armazenados em *vetorresultado* já na sua posição correta. Na linha 11, a rotina *MPI_Gather* preenche o restante do vetor *vetorresultado* em ordem com os elementos tirados dos vetores *vetorprovisorio* de cada um dos processos no processo *size-1*. Na linha 13, o ambiente MPI é finalizado e na linha seguinte a medição de tempo é parada. Na linha 17 o tempo de execução é dividido

por 20 para se ter o tempo de execução médio de uma iteração.

```

1  while (repeticao>0) {
2      repeticao--;
3      MPI_Scatter(vetorg, ta, MPI_SHORT, vetorgg, ta, MPI_SHORT, size-1,
4                  MPI_COMM_WORLD);
5      MPI_Scatter(vetorp, ta, MPI_SHORT, vetorpp, ta, MPI_SHORT, size-1,
6                  MPI_COMM_WORLD);
7      for(int i=0; i<ta; i++){
8          vetorprovisorio[i]=calc_real_p(vetorpp[i], vetorgg[i]);
9          if (rank==size-1 && dif!=0) {
10             for (int i = 0; i < dif; i++) {
11                 vetorresultado[NUM_P-dif+i]=calc_real_p(vetorp[NUM_P-dif+i], vetorg[
12                     NUM_P-dif+i]);}
13             }
14             MPI_Gather(vetorprovisorio, ta, MPI_DOUBLE, vetorresultado, ta,
15                         MPI_DOUBLE, size-1, MPI_COMM_WORLD);
16         }
17         MPI_Finalize();
18         Ticks[1]=clock();
19         double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
20         if (rank==size-1) {
21             printf("Tempo de execução:%fms\n", Tempo/20);}
22         return 0;
23     }

```

Listing 4.16 – scatter-gather.c

Comentários Adicionais.

Com o mesmo objetivo que o código *send-receive.c*, este programa utiliza de rotinas de comunicação coletiva que facilitam o implemento da solução. Apesar de não ser tão eficiente quanto *send-receive.c*, as diferenças dos tempos de execução medidos, não são tão diferentes.

4.2.2.3 Análise do Código *bcast-reduce.c*

O que o programa faz.

Este programa distribui os vetores de dados *vetorg* e *vetorp* para os *workers* através de *MPI_Bcast* e, dependendo se o *rank* do processo for par ou ímpar, executa uma operação específica. No final, soma todos os resultados através de *MPI_Reduce* para um único *vetorresultado* no processo mestre.

A razão de ter sido feito este programa:

O objetivo foi o de exemplificar formas de uso das rotinas de comunicação coletiva *MPI_Bcast* e *MPI_Reduce*, bem como gerar um comparativo ao código em Julia com funcionalidade similar a este, *remote-do-sharedarray.jl*.

Analisando o Código.

Analisando o código da Listagem 4.17. As *libs* utilizadas por este código são as mesmas já mostradas. As variáveis de apoio declaradas são as mesmas, exceto que neste código, devido a natureza de *bcast* e do próprio objetivo do código, não há necessidade em ter cautela com a divisão de *NUM_P* por *size* já que todos os processos recebem todos os valores. Por este motivo, as variáveis que lidam com isso que estavam presentes no código de *Send/Recv* não são necessárias aqui. Existe uma mudança na declaração de funções, pois este código usa duas, como pode ser visto nas linhas 1 e 3.

```

1 double calc_real_p_par(int x, int y){
2   return (sin(cos(x))*(tan(sin(cos(y)))));}
3 double calc_real_p_impar(int x, int y){
4   return (sin(cos(x))*(tan(sin(cos(y))))*1/tan(sin(cos(x*y))));}

```

Listing 4.17 – bcast-reduce.c

Analisando o código da Listagem 4.18. Os envios dos vetores são feitos nas linhas 3 e 4. No caso deste programa, o processo mestre é o último processo adicionado, por isso o *size-1* como remetente dentro dos argumentos das rotinas. A medida de tempo, como sempre, foi iniciada antes da entrada no *loop while*.

Esta parte do código diferencia entre os *ranks* pares e ímpares atribuindo uma das duas funções *calc_real_p_par* e *calc_real_p_impar* ao processo em questão. Nas linhas 6 e 7, os processos pares preenchem seus vetores *vetorprovisorio* com os resultados da passagem de seus vetores *vetorg* e *vetorp* pela função *calc_real_p_par*. Nas linhas 10 e 11, os processos ímpares preenchem seus vetores *vetorprovisorio* com os resultados da passagem de seus vetores *vetorg* e *vetorp* pela função *calc_real_p_impar*.

A redução através de soma feita pela rotina *MPI_Reduce* ocorre na linha 13. Aqui os vetores *vetorprovisorio* de todos os processos têm o valor contido em suas casas somado e este resultado é armazenado nas respectivas casas de *vetorresultado* do processo *size-1*.

```

1 while (repeticao>0) {
2   repeticao--;
3   MPI_Bcast(vetorg, NUM_P, MPI_SHORT, size-1, MPI_COMM_WORLD);
4   MPI_Bcast(vetorp, NUM_P, MPI_SHORT, size-1, MPI_COMM_WORLD);
5   if (rank%2==0) {
6     for(int i=0; i<NUM_P; i++){
7       vetorprovisorio[i]=calc_real_p_par(vetorp[i], vetorg[i]);}
8   }
9   else{
10    for(int i=0; i<NUM_P; i++){
11      vetorprovisorio[i]=calc_real_p_impar(vetorp[i], vetorg[i]);}
12  }
13  MPI_Reduce(vetorprovisorio, vetorresultado, NUM_P, MPI_DOUBLE, MPI_SUM
    , size-1, MPI_COMM_WORLD);

```

14 }

Listing 4.18 – bcast-reduce.c

Analisando o código da Listagem 4.19. Na linha 1, o ambiente MPI é encerrado. A medida de tempo é encerrada e impressa, depois de dividida por 20, em tela pelo processo *size-1*. O código se encerra com o *return* na linha 6.

```

1 MPI_Finalize();
2 Ticks[1]=clock();
3 double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
4 if (rank==size-1) {
5     printf("Tempo de execução médio:%fms\n", Tempo/20);}
6 return 0;
7 }
```

Listing 4.19 – bcast-reduce.c

Comentários Adicionais.

Este código, similar ao código *remote-do-sharedarrays.jl* de Julia, é peculiar porque o número de elementos que ele irá processar depende do número de processos utilizados. Por este motivo, o tempo de execução entre 4 e 2 processos deste programa não devem ser comparados, já que com 4 processos o número de elementos computados seria de $4 * NUM_P$, enquanto o de 2 processos computaria $2 * NUM_P$ elementos.

4.2.2.4 Análise do Código *scatter-reduce.c*

O que o programa faz.

Faz uma operação de NUM_P elementos, que são distribuídos entre *size* processos através do comando *MPI_Scatter*. Reduz todos os elementos para um só valor, armazenado em *global_sum*, através de *MPI_Reduce*.

A razão de ter sido feito este programa:

Utilizar das rotinas *MPI_Scatter* e *MPI_Reduce* como alternativa de comunicação coletiva para produzir um único valor e criar um comparativo para o código similar a esse em Julia, *distributed-sharredarrays.jl*.

Analisando o Código.

Analisando o código da Listagem 4.20. Este código requer declarações similares aos do código *Scatter/Gather*, mas não foi necessário um *vetorresultado* ou um *vetorprovisorio*, declarou-se duas variáveis do tipo *double* que possuem os nomes de *local_sum* e *global_sum* e atribuiu-se a ambas o valor zero. Além disso, é necessário ter cautela com a divisão de NUM_P por *size*, portanto as variáveis e medidas tomadas nos outros códigos com este problema também se aplicam.

Já com o medidor de tempo iniciado e os elementos de *vetorg* e *vetorp* já distribuídos para *vetorgg* e *vetorpp* para todos os processos através do *MPI_Scatter* nas linhas 3 e 4, cada processo passa os seus valores pela função *calc_real_p* e os soma ao valor já existente de *local_sum*, que inicialmente é zero. Nas linhas 8 e 9, o processo mestre *size-1* trata dos restos não distribuídos por *MPI_Scatter*. Ele os passa pela função e os soma diretamente ao valor de *local_sum*, que inicialmente é zero.

Na linha 11, a rotina *MPI_Reduce* é utilizada para somar todos os *local_sum* de todos os processos no *global_sum* do processo mestre *size-1*. É finalizado o ambiente MPI na linha 13. Na linha 14, é parado medidor de tempo. Nas linhas 17 e 18, o tempo medido médio é impresso em tela e o valor de *global_sum* também. O programa se encerra na linha 19.

```

1  while(repeticao>0){
2      repeticao--;
3      MPI_Scatter(vetorg, ta, MPI_SHORT, vetorgg, ta, MPI_SHORT, size-1,
4                  MPI_COMM_WORLD);
5      MPI_Scatter(vetorp, ta, MPI_SHORT, vetorpp, ta, MPI_SHORT, size-1,
6                  MPI_COMM_WORLD);
7      for(int i=0; i<ta; i++){
8          local_sum+=calc_real_p(vetorpp[i], vetorgg[i]);
9          if (rank==size-1 && dif!=0) {
10             for (int i = 0; i < dif; i++) {
11                 local_sum+=calc_real_p(vetorp[NUM_P-dif+i], vetorg[NUM_P-dif+i]);
12             }
13         }
14         MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, size-1,
15                   MPI_COMM_WORLD);
16     }
17     MPI_Finalize();
18     Ticks[1]=clock();
19     double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
20     if (rank==size-1) {
21         printf("Tempo de execução médio:%fms\n", Tempo/20);
22         printf("Valor da soma global= %f\n", global_sum);
23     }
24     return 0;
25 }
```

Listing 4.20 – scatter-reduce.c

Comentários Adicionais.

Este programa possuiu um desempenho bem similar ao de *distributed-sharredarrays.jl*, seu semelhante em Julia, sua implementação, por outro lado, foi mais complexa.

4.2.2.5 Análise do Código *simulacao-remote-channels.c*

O que o programa faz.

Esse programa tem como objetivo simular e utilizar *RemoteChannels* para gerar um *pipeline*. O programa gera um número *NUM_P* de vetores de *N* casas populadas com variáveis aleatórias para cada um dos dois vetores: *vetorg* e *vetorp*. Esses vetores são gerados no processo 0 e são recuperados pelos demais processos através da troca de mensagens por MPI. Nos *workers*, os vetores *vetorg* e *vetorp* têm cada um de seus elementos passados pela função *calc_real_p*. Após isso, os valores encontrados são devolvidos ao processo 0.

A razão de ter sido feito este programa:

Simular um *RemoteChannel* de Julia em C para averiguar a sua viabilidade e também para que o código de Julia, *@spawnat-remotechannels.jl*, pudesse ser propriamente comparado já que não havia nenhum outro código que tivesse aquela mesma funcionalidade.

Analisando o Código.

São listadas as *libs* utilizadas, declaradas as constantes *NUM_P* e *N*, além da função *calc_real_p*. São feitas as declarações das variáveis de apoio, a inicialização do ambiente MPI, a alocação de memória de *vetorresultado* e o início da medida de tempo. Com isso feito, entrando no *loop while* analisa-se a parte do código referida ao Produtor.

Analisando o código da Listagem 4.21. Na linha 1, é indicado que somente o processo mestre executa deste ponto em diante. Esta parte do código tem como objetivo produzir os dados, distribuí-los para os *workers*, recebe-los já processados e armazená-los em *vetorresultado*. Na linha 5, é recebido o vetor *temp* com os valores já passados por *calc_real_p*. Caso seja a primeira vez do consumidor entrando em contato com o produtor, essa mensagem serve somente para conectá-los e, após a primeira comunicação, passa a ter dados úteis até o consumidor ser desligado.

Na linha 12, o *status.MPI_TAG* é utilizado para diferenciar se uma mensagem recebida pelo processo mestre é o primeiro contato entre ele e o *worker* em questão ou não. Esta mensagem serve para passagem de dados e para estabelecer um canal entre o produtor e consumidor (um dos *workers*). Caso a mensagem recebida seja o primeiro contato, a variável *status.MPI_TAG* possui o valor 0. Sendo assim, a linha 14 que faz com que o conteúdo de *temp* seja armazenado em *vetorresultado* só é executada quando não se trata da primeira comunicação entre o processo 0 e o *worker*. Nas linhas 16 e 17, o processo mestre envia os vetores populadas *vetorg* e *vetorp* para o *worker* em questão.

```

1  if (rank == 0) {
2      srand(time(NULL));
3      short int *vetorp=(short int*)malloc(N*sizeof(short int));
4      short int *vetorg=(short int*)malloc(N*sizeof(short int));
5      double *temp=(double*)malloc((N)*sizeof(double));
6      for (int j = 1; j <= NUM_P; ++j) {
7          MPI_Status status;

```

```

8   for (int i = 0; i<N; i++) {
9       vetorg[i]=rand()%100+1;
10      vetorp[i]=rand()%100+1;}
11      MPI_Recv(temp, N, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
12      if (status.MPI_TAG > 0) {
13          for (int i=0; i<N; i++) {
14              vetorresultado[((status.MPI_TAG-1)*N)+i]=temp[i];}
15          }
16      MPI_Send((vetorg), N, MPI_SHORT, status.MPI_SOURCE, j,
MPI_COMM_WORLD);
17      MPI_Send((vetorp), N, MPI_SHORT, status.MPI_SOURCE, j,
MPI_COMM_WORLD);
18  }

```

Listing 4.21 – simulacao-remote-channels.c

Analisando o código da Listagem 4.22. No intervalo entre as linhas 1 e 11, o produtor lida com a última comunicação e envia um sinal de encerramento que acaba com o canal estabelecido entre ele e o *worker* com o qual ele estiver se comunicando. Na linha 4, o processo 0 recebe os elementos já processados do consumidor, os armazena no *vetorresultado* e, nas linhas 9 e 10, os *MPI_Send* com o argumento de *MPI_Tag* igual a 0 são passados, servindo como um sinal de desligamento para o consumidor. Na linha 12, acabam as instruções do processo mestre como produtor.

```

1   int num_terminated = 0;
2   for (int num_terminated = 0; num_terminated < size-1; num_terminated
++) {
3       MPI_Status status;
4       MPI_Recv(temp, N, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
5       if (status.MPI_TAG > 0) {
6           for (int i=0; i<N; i++) {
7               vetorresultado[((status.MPI_TAG-1)*N)+i]=temp[i];}
8           }
9       MPI_Send(vetorg, N, MPI_SHORT, status.MPI_SOURCE, 0, MPI_COMM_WORLD)
;
10      MPI_Send(vetorp, N, MPI_SHORT, status.MPI_SOURCE, 0, MPI_COMM_WORLD)
;
11  }
12  }

```

Listing 4.22 – simulacao-remote-channels.c

Analisando o código da Listagem 4.23. Na linha 1, é indicado que somente os *workers* executam deste ponto em diante. Esta parte do código tem como objetivo receber os dados distribuídos pelo produtor, passá-los pela função *calc_real_p* e enviar este

resultado de volta ao produtor. Na linha 5, o consumidor estabelece um canal com o produtor através da rotina *MPI_Send* com o argumento de *MPI_TAG* igual a 0. Na linha 6, a variável *terminated* que indica que o consumidor pode encerrar suas atividades é marcado como falso.

Na linha 7, os consumidores executam este *loop* até que a variável *terminated* seja marcada como verdadeiro, como indicado na linha 18. Nas linhas 9 e 10, o consumidor recebe do produtor os elementos a serem computados, bem como o valor de *MPI_TAG* que pode significar seu desligamento. Na linha 11, é verificado se o consumidor recebeu o sinal de desligamento. Caso tenha recebido, a linha 12 marca *terminated* como verdadeiro e o consumidor se encerra. Caso não tenha recebido, ou seja, *MPI_TAG* é diferente de 0, o consumidor, nas linhas 14 e 15, passa os elementos pela função *calc_real_p* e os armazenam em *temp*, que é enviado para o produtor na linha 17.

```

1  else {
2      short int *vetorp=(short int*)malloc(N*sizeof(short int));
3      short int *vetorg=(short int*)malloc(N*sizeof(short int));
4      double *temp=(double*)malloc((N)*sizeof(double));
5      MPI_Send(temp, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
6      bool terminated = false;
7      do {
8          MPI_Status status;
9          MPI_Recv(vetorg, N, MPI_SHORT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
10             status);
11          MPI_Recv(vetorp, N, MPI_SHORT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
12             status);
13          if (status.MPI_TAG == 0) {
14              terminated = true;}
15          else {
16              for (int i=0; i<N; i++) {
17                  temp[i]=calc_real_p(vetorg[i], vetorp[i]);
18              }
19              MPI_Send(temp, N, MPI_DOUBLE, 0, status.MPI_TAG, MPI_COMM_WORLD);}
20      } while (!terminated);
21  }

```

Listing 4.23 – simulacao-remote-channels.c

Analisando o código da Listagem 4.24. Na linha 1, o ambiente MPI é encerrado. Na linha 2, a medida de tempo é encerrada. O tempo de execução é impresso em tela na linha 5 depois de ser dividido por 20. Na linha 6, acaba o programa.

```

1  MPI_Finalize();
2  Ticks[1]=clock();
3  double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
4  if (rank == 0) {
5      printf("Tempo de execução médio:%fms\n", Tempo/20);}
6

```


6 }

Listing 4.24 – simulacao-remote-channels.c

Comentários Adicionais.

Variando o valor de N ou NUM_P não resultou em uma diferença de tempo de execução entre os dois, portanto variou-se o valor de NUM_P e, este foi o único resultado referido nas tabelas quanto a canais remotos em C, diferente do código de Julia, *@spawnat-remotechannels.jl*.

4.3 Avaliação de Desempenho

As tabelas apresentadas nesta seção servem ao propósito de comparar os tempos de execução medidos entre programas de Julia e os programas de C com o objetivo de avaliar o desempenho da linguagem Julia relativo a C empregando MPI.

As Tabelas 3 e 4 são referentes aos tempos de execução dos programas de C que não envolvem canais remotos.

Tabela 3 – Tempos de Execução dos Programas de C Utilizando 4 Processos.

Valor de <i>NUM_P</i>	<i>Send/Recv</i>	<i>Scatter/Gather</i>	<i>Bcast/Reduce</i>	<i>Scatter/Reduce</i>
1.000.000	59,029 ms	61,896 ms	396,048 ms	58,950 ms
5.000.000	292,809 ms	306,278 ms	2,001 s	292,552 ms
10.000.000	583,573 ms	620,451 ms	3,963 s	584,553 ms

Tabela 4 – Tempos de Execução dos Programas de C Utilizando 2 Processos.

Valor de <i>NUM_P</i>	<i>Send/Recv</i>	<i>Scatter/Gather</i>	<i>Bcast/Reduce</i>	<i>Scatter/Reduce</i>
1.000.000	82,445 ms	84,773 ms	274,937 ms	82,834 ms
5.000.000	409,957 ms	421,306 ms	1,379 s	411,681 ms
10.000.000	819,111 ms	841,041 ms	2,762 s	823,077 ms

Referente às Tabelas 3 e 4, pode-se perceber quanto aos códigos analisados:

- *Send/Recv*: Dentre os códigos analisados, este esteve entre os mais rápidos. Aumentando os números além de 10.000.000, a sua velocidade em relação ao *@spawnat* ficaria mais aparente. Apesar desse desempenho superior, os outros métodos que envolvem rotinas de comunicação coletiva se mostram mais simples e com pouca perda de desempenho. Quando comparado com seu semelhante em Julia, *@spawnat*, quanto a sua simplicidade do código, *@spawnat* é mais simples, já quanto ao desempenho, *send-receive* fica com um resultado melhor.
- *Scatter/Gather*: A implementação do objetivo de *Send/Recv* usando de rotinas de comunicação coletiva. Ficou mais simples do que *Send/Recv* com pouca perda de desempenho. Quando comparado ao seu semelhante em Julia, *pmap*, ganha em questão de desempenho e perde em questão de simplicidade.
- *Bcast/Reduce*: Um código com um objetivo diferente dos demais, *Bcast/Reduce* só pode ser comparado com o seu semelhante de Julia, *remote-do*. Quando comparados, fica claro que *Bcast/Reduce* possuiu um tempo de execução melhor e também ficou um código mais simples e claro.
- *Scatter/Reduce*: Este programa ficou com uma implementação simples, porém *distributed.jl* ficou ainda mais simples. *Scatter/Reduce* ficou com uma vantagem no tempo de execução.

As Tabelas 5 e 6 são referentes aos tempos de execução dos programas de Julia que não envolvem canais remotos.

Tabela 5 – Tempos de Execução dos Programas de Julia Utilizando 4 ou 5 Processos. Nomes marcados com um * utilizaram de 5 processos.

Valor de <i>NUM_P</i>	<i>@spawnat</i> *	<i>pmap</i>	<i>remote-do</i>	<i>distributed</i> *
1.000.000	220,618 ms	67,542 s	1,383 s	79,609 ms
5.000.000	504,926 ms	349,209 s	6,433 s	337,603 ms
10.000.000	860,962 ms	682,957 s	12,465 s	644,375 ms

Tabela 6 – Tempos de Execução dos Programas de Julia Utilizando 2 ou 3 Processos. Nomes marcados com um * utilizaram de 3 processos.

Valor de <i>NUM_P</i>	<i>@spawnat</i> *	<i>pmap</i>	<i>remote-do</i>	<i>distributed</i> *
1.000.000	317,067 ms	89,615 s	601,125 ms	100,480 ms
5.000.000	756,460 ms	459,721 s	2,472 s	465,917 ms
10.000.000	1,361 s	902,363 s	5,923 s	918,334 ms

Referente às Tabelas 5 e 6, pode-se perceber quanto aos códigos analisados:

- *@spawnat*: Este programa foi relativamente simples, quando comparado ao *send-receive.c*, e não utiliza de rotinas de comunicação coletiva de Julia. Quando comparado ao código de *send-receive.c*, ele fica com um tempo de execução claramente maior.
- *pmap*: Comparado com os demais, claramente mostrou o pior tempo de execução. A diferença entre a execução com 4 e 5 processos foi pequena, mas a de 4 processos foi melhor. Isso é interessante pois o *pmap*, similarmente ao código de *distributed*, trabalha usando todos os *workers* disponíveis, logo seria intuitivo deduzir que a execução com 4 *workers*, 1 para cada *core* disponível, seria a ideal, entretanto a arquitetura de *pmap* para atingir este objetivo específico parece ser ineficaz ao ponto em que a execução com 3 *workers* possui um tempo de execução ligeiramente menor do que com 4. Apesar de também ter sido o código mais curto e fácil de implementar, o tempo de execução medido reflete mal nesta rotina. Como já explicado na seção sobre *pmap*, esta rotina possui uma arquitetura feita para tratar com poucas chamadas de função que fazem uma computação pesada. O objetivo de *pmap* teria sido melhor cumprido usando de *@distributed*, que não possui essa mesma arquitetura mencionada.
- *remote-do*: A rotina *remote-do* na situação apresentada e da forma que foi utilizada, fica subaproveitada e, se empregada de forma e em uma situação que utilize melhor do seu potencial, teria resultados superiores.

- *distributed*: Este código possui um bom desempenho. Mesmo aumentando o número de elementos a serem computados ele consegue acompanhar o tempo de execução *scatter-reduce.c*.

Os tempos medidos nas Tabelas 7 e 8 são referentes aos programas de C e Julia de Canais Remotos.

Tabela 7 – Tempos de Execução dos Programas de C e Julia de Canais Remotos Utilizando 4 ou 5 Processos. Nomes marcados com um * utilizaram de 5 processos.

Elementos Computados	1.000.000	5.000.000	10.000.000
@spawnat-remotechannels* <i>NUM_P</i> variando	4,216 s	70,901 s	267,585 s
@spawnat-remotechannels* <i>N</i> variando	4,216 s	7,133 s	10,662 s
<i>simulacao-remote-channels</i>	83,616 ms	434,541 ms	1,079 s

Tabela 8 – Tempos de Execução dos Programas de C e Julia de Canais Remotos Utilizando 2 ou 3 Processos. Nomes marcados com um * utilizaram de 3 processos.

Elementos Computados	1.000.000	5.000.000	10.000.000
@spawnat-remotechannels* <i>NUM_P</i> variando	3,536 s	59,585 s	231,458 s
@spawnat-remotechannels* <i>N</i> variando	3,536 s	6,671 s	10,198 s
<i>simulacao-remote-channels</i>	166,854 ms	825,517 ms	1,642 s

Referente às Tabelas 7 e 8, pode-se perceber quanto aos códigos analisados:

- *Simulacao-Remote-Channels*: Comparado ao código de Julia, que já possui *RemoteChannels*, este programa fica mais complexo. Quanto a forma de implementação, este programa é de mais fácil compreensão. O estabelecimento de canais remotos utilizando de comunicação ponto a ponto de MPI foi a melhor forma encontrada de simular estes canais remotos. Seu tempo de execução é bem mais rápido do que o de *@spawnat-remotechannels*.
- *@spawnat-remotechannels*: Apesar de ter os canais remotos já embutidos em sua arquitetura, este programa fica com o seu tempo de execução bem alto. Isso é minimizado alterando o valor de *N* para aumentar o número de elementos enquanto se fixa o valor de *NUM_P*. Isso significa que está sendo mantido fixo a quantidade de envios de mensagens, o que está sendo alterado é o seu conteúdo, que está sendo aumentado. Logo, para melhorar o desempenho, aumentou-se a quantidade de variáveis passadas por mensagem enquanto se diminuiu a quantidade de mensagens passadas. Neste programa, como já mencionado, a meta foi a de se utilizar de canais remotos para tentar fazer a maior parte da comunicação. Depois dos resultados obtidos, os *RemoteChannels* parecem servir melhor como *buffers* acessíveis por todos os processadores.

5 Conclusões

Este trabalho apresentou uma comparação do uso das linguagens C e Julia no que se refere a estruturas de programação paralela, mais especificamente no modelo de programação paralela com troca de mensagens. As estruturas dessas linguagens foram comparadas no que concerne: simplicidade e clareza do código; desempenho; pacotes e *libs* disponíveis.

Em Julia foram estudadas as seguintes estruturas paralelas para processos distribuídos: *@spawnat*, *RemoteChannels*, *remote-do*, *@distributed*, *pmap* e *SharedArrays*. Já em C, empregou-se as rotinas oriundas da biblioteca de MPI: *MPI_Send*, *MPI_Recv*, *MPI_Scatter*, *MPI_Gather*, *MPI_Reduce* e *MPI_Bcast*.

Na comparação entre as duas estruturas, os resultados mostraram que, quanto ao desempenho, Julia não é uma linguagem tão rápida quanto C. Julia mostrou desempenho com tempos comparáveis a C em 2 dos seus 5 programas estudados, *@spawnat* e *distributed*. Quanto a comparação entre C e Julia a respeito de simplicidade e clareza de código, os programas de Julia ficaram com complexidade inferiores aos códigos de C, apontando-a como uma alternativa superior neste caso. Os pacotes utilizados em Julia são modernos e atualizados constantemente devido a linguagem ainda estar em sua infância quando relacionada a linguagens tal como C que foi criada em 1972 contra Julia que foi criada em 2012. Isso é uma vantagem e desvantagem para Julia, pois com modernizações constantes se tem a tecnologia mais atual, mas também se tem inconstância e instabilidade.

Concluí-se que Julia é uma boa alternativa para C em geral, devido às suas características de linguagem compilada ao mesmo tempo que permite as facilidades de uma linguagem interpretada, como *Python*. Porém quanto ao desempenho, Julia não mostrou ser tão rápida quanto C. As características possuídas por Julia que C não possui, como *optional typing*, despacho múltiplo, entre outras, fazem de Julia uma opção a ser considerada quando for necessária uma linguagem com bom desempenho aliado a essas características modernas. É importante ressaltar que apesar de somente 2 dos 5 programas de Julia analisados ficarem com tempos comparáveis aos de C, os programas foram feitos com intenção de explorar as rotinas oferecidas por Julia, testá-las e compará-las com C. Alguns programas, *pmap* se sobressaindo neste quesito, não foram utilizados de forma que fosse aproveitado seu potencial por completo.

Julia pode vir a se tornar uma ameaça para a hegemonia de *Python* no futuro de acordo com [Moutafis \(2020\)](#), que diz:

Julia is a very new language that competes head-on with Python. It fills the gap of large-scale technical computations: Usually, one would have used Python or Matlab, and patched the whole thing up with C++

libraries, which are necessary at a large scale. Now, one can use Julia instead of juggling with two languages.¹

Com isso, se pode inferir alguma perda de mercado de *Python* em prol de Julia, visto que esta possui as mesmas características que levaram a alta propagação do uso de *Python* só que acrescido do desempenho superior apresentado. *Python* e C já estão estabelecidos no mercado há muito tempo, por esse motivo, essas linguagens possuem comunidades fortes, o que facilita usuários novos adotarem a linguagem, e significa um número maior de pacotes ou *libs* disponíveis. O que falta para Julia ser mais utilizada é tempo para que a linguagem possa se assentar.

Futuramente, este trabalho pode ser estendido através do estudo de estruturas de *deep learning* em C e em Julia. Seria interessante também uma comparação entre Julia, *Python* e R.

¹ O trecho correspondente na tradução é: “Julia é uma linguagem muito nova que compete de frente com *Python*. Ela preenche o vazio de computações técnicas de grande escala: Normalmente, se utilizaria *Python* ou *Matlab*, e remendado isso tudo com *libs* de C++, que são necessárias em grande escala. Agora, pode-se utilizar Julia ao invés de se utilizar duas linguagens.”

Referências

BEZANSON, J. et al. *Julia: A Fresh Approach to Numerical Computing*. 2020. Disponível em: <<https://epubs.siam.org/doi/10.1137/141000671>>. Acesso em: 25 maio 2020. Nenhuma citação no texto.

BEZANSON, J. et al. *Why We Created Julia*. 2020. Disponível em: <<https://julialang.org/blog/2012/02/why-we-created-julia/>>. Acesso em: 24 maio 2020. Citado na página 15.

DOCUMENTAÇÃO de Julia v1.1.1. 2020. Disponível em: <<https://docs.julialang.org/en/v1.1/>>. Acesso em: 23 abril 2020. Nenhuma citação no texto.

DOCUMENTAÇÃO de MPICH. 2020. Disponível em: <<http://www.mpich.org/static/docs/latest/>>. Acesso em: 24 abril 2020. Nenhuma citação no texto.

GROPP, W.; LUSK, E. *Scientific and Engineering Computation*. Cambridge, Massachusetts: The MIT Press, 2014. Citado na página 35.

METAPROGRAMMING. 2020. Disponível em: <<https://docs.julialang.org/en/v1.1/manual/metaprogramming/>>. Acesso em: 10 junho 2020. Citado na página 20.

MOUTAFIS, R. *Why Python is not the programming language of the future*. 2020. Disponível em: <<https://towardsdatascience.com/why-python-is-not-the-programming-language-of-the-future-30ddc5339b66>>. Acesso em: 20 maio 2020. Citado na página 77.

MPI Routines and Constants. 2020. Disponível em: <<http://www.mpich.org/static/docs/v3.2.x/>>. Acesso em: 25 maio 2020. Nenhuma citação no texto.

MPICH Overview. 2020. Disponível em: <<http://www.mpich.org/about/overview/>>. Acesso em: 25 abril 2020. Citado na página 35.

REVELS, J. *Benchmark Tools*. 2020. Disponível em: <<https://github.com/JuliaCI/BenchmarkTools.jl>>. Acesso em: 08 abril 2020. Nenhuma citação no texto.

Apêndices

APÊNDICE A – Códigos de Julia

A.0.1 exemplo-sequencial.jl

```
using BenchmarkTools
using Random
using Statistics
const NUM_P=1000000
Random.seed!(1234)
function popula_vetor()::Array{Int8}
    return rand(1:100, NUM_P)
end
const vetor1=popula_vetor()
const vetor2=popula_vetor()
b=@benchmarkable map(+, vetor1, vetor2) memory_tolerance=0.01
    time_tolerance=0.01 seconds=1500 samples=20
println(mean(run(b)))
```

Listing A.1 – exemplo-sequencial.jl

A.0.2 @spawnat.jl

```
#=
Este programa faz a distribuição de NUM_P elementos dos arrays vetorg e
vetorp para n processos
utilizando @spawnat. Esses elementos são passados por uma função nos
workers e retornam ao
processo 1.
=#
using BenchmarkTools
using Distributed
using Statistics
using Random

#Aqui são utilizados 5 processos no total. Este programa foi feito de
forma que somente os workers
#processam os dados distribuídos, sendo assim, tendo só 3 processos
trabalhando simultaneamente é
#subaproveitar o processador com 4 cores.
addprocs(4)
#Utilizamos algo similar a um #define em C para deixar o código mais
limpo e compreensível
#NUM_P se refere ao número de iterações que serão feitas
const NUM_P=1000*10000

Random.seed!(1234)
```

```

function popula_vetor()::Array{Int8}
    return rand(1:100, NUM_P)
end

#utilizamos ta para encontrar o valor do intervalo do vetor que cada
#worker deverá computar
const ta = floor{Int32, NUM_P/nworkers()}

#essa é a função aleatória que usaremos para teste
@everywhere function calcpint(x,y::Int8)::Float64
    return (sin(cos(x))*(tan(sin(cos(y)))))
end

#aqui determinamos se dif=0 ou não. se NUM_P/nworkers() não for um
#inteiro, precisamos compensar isso
#para podermos fazer o paralelismo adequado sem deixar de fazer o número
#exato de repetições NUM_P
if (NUM_P/nworkers()-floor{Int32, NUM_P/nworkers()}!=0)
    const dif=NUM_P-ta*nworkers()
else const dif=0
end

#declaramos e populamos vetorp e vetorg com NUM_P números aleatórios de
#1 a 100
vetorp=popula_vetor()
vetorg=popula_vetor()

#aqui começamos a medir o tempo gasto
b= @benchmarkable begin
    #buffer é um vetor necessário, caso NUM_P/nworkers() seja uma divisão
    #inexata
    buffer=zeros(0)
    #aqui declaramos R, R[1]:R[nworkers()] são futuros que podem ser
    #resgatados
    #fazendo R[1][1] acessamos ao primeiro futuro disponível no vetor R e
    #acessamos dentro dele,
    #o primeiro elemento do primeiro vetor. fazendo R[1][1:ta] estamos
    #pegando
    #o primeiro vetor do seu primeiro elemento até o último elemento do
    #intervalo em que ele trabalhou
    R=[@spawnat i map(calcpint, vetorp[ta*(myid()-2)+1:ta*(myid()-1)],
        vetorg[ta*(myid()-2)+1:ta*(myid()-1)]) for i=2:nprocs()]

    #aqui, caso NUM_P não seja divisível por nworkers(), o resto, dif, é
    #processado pelo processo 1
    #e são guardados num buffer os valores encontrados por ele

```

```

if dif!=0
    append!(buffer, (map(calcpint, vetorp[NUM_P-dif+1:NUM_P], vetorg[NUM_P
        -dif+1:NUM_P])))
end

#é declarado o vetorresults que será o vetor de armazenamento dos
    resultados calculados
vetorresults=zeros(Float64, 0)
#neste for, que abrange todos os workers, estamos pegando os valores
    encontrados e, conforme
#explicado mais acima, são armazenados os intervalos de cada R[i]
    atrelado a seu worker.
for i in 2:nprocs()
    append!(vetorresults, R[i-1][1:ta])
end
#aqui, se dif for diferente de 0, são armazenados os valores "resto"
    calculados pelo processo 0
append!(vetorresults, buffer)
    #println("Tamanho do vetor=", length(vetorresults))
    #println(vetorresults)
    return vetorresults
end seconds=1500 samples=20 time_tolerance=0.01 memory_tolerance=0.01
println(mean(run(b)))

```

Listing A.2 – @spawnat.jl

A.0.3 pmap.jl

```

#=
Este programa faz a distribuição de NUM_P elementos dos arrays vetorg e
    vetorp para n processos
utilizando pmap. Esses elementos são processados e retornados ao
    processo 1.
Pmap é estruturado para o caso em que cada chamada de função faz uma
    quantia enorme de trabalho, não sendo
esse o caso desse programa, pmap parece subótimo em comparação aos
    demais.
=#
using BenchmarkTools
using Distributed
using Statistics
using Random
addprocs(3)
#Utilizamos algo similar a um #define em C para deixar o código mais
    limpo e compreensível
#NUM_P se refere ao número de iterações que serão feitas
const NUM_P=1000*1000

```

```

Random.seed!(1234)

function popula_vetor()::Array{Int8}
    return rand(1:100, NUM_P)
end

@everywhere function calcpint(x,y::Int8)::Float64
    return (sin(cos(x))*(tan(sin(cos(y))))))
end

const vetorp=popula_vetor()
const vetorg=popula_vetor()

b=@benchmarkable pmap(calcpint, vetorg, vetorp) seconds=1500 samples=20
    time_tolerance=0.01 memory_tolerance=0.01
println(mean(run(b)))

```

Listing A.3 – pmap.jl

A.0.4 remote-do.jl

```

#=
Este programa disponibiliza os vetores de dados vetorg e vetorp para os
workers através do uso de
SharedArrays e, dependendo se o myid() do processo for par ou ímpar,
executa uma operação específica.
No final, soma todos os resultados através de map para um único
vetorresultado no processo 1.
=#

using BenchmarkTools
using Distributed
addprocs(4)
using SharedArrays
using Statistics
using Random
Random.seed!(1234)
#Utilizamos algo similar a um #define em C para deixar o código mais
limpo e compreensível
#NUM_P se refere ao número de iterações que serão feitas
const NUM_P=1000*1000

#O motivo do global ser utilizado na inicialização de vetorresults é
para permitir seu uso dentro do for
global vetorresults
#O vetordezeros é utilizado para zerar o valor de vetorresults na
medição de tempo
const vetordezeros=zeros(Float64, NUM_P)

```

```

#Aqui declaramos um RemoteChannel para cada um dos processos que
    utilizaremos para permitir o
#armazenamento depois de passar os dados pelas funções.
for p in procs()
    @fetchfrom p global inbox = RemoteChannel{()>Channel{}}(1)
end

function popula_vetor()::Array{Int8}
    return rand(1:100, NUM_P)
end

#as funções a seguir representam uma caixa preta. é definida por
    qualquer procedimento simples
#pelo qual se queira passar os dados
#@everywhere incluído permitindo aos workers utilizar as funções.
@everywhere function calc_real_p_par(x,y::Int8)::Float64
    return (sin(cos(x))*(tan(sin(cos(y)))))
end

@everywhere function calc_real_p_impar(x,y::Int8)::Float64
    return (sin(cos(x))*(tan(sin(cos(y))))) * 1/tan(sin(cos(x*y)))
end

#Essa função serve ao propósito de encapsular as funções calc_real_p_par
    e calc_real_p_impar,
#enquanto permite a passagem dos argumentos X e Y, que, apesar de
    estarem disponíveis nos workers,
#não é possível acessá-los sem tê-los passado como argumentos, já que os
    workers não possuem os
#rótulos dos SharedArrays.
@everywhere function work(X, Y)

    if (myid()/2)-floor(myid()/2)!=0
        put!(inbox, map(calc_real_p_impar, X, Y))
    else
        put!(inbox, map(calc_real_p_par, X, Y))
    end
end

#Devido a dificuldades na medição de tempo correta decorrentes da
    natureza do programa, tornou-se
#necessário utilizar a rotina @benchmarkable que funciona melhor tendo
    uma só função para gerenciar.
function paralelizar(x,y::SharedArray)
    #aqui ocorre a chamada aos processos para passar X e Y pelas funções. é
        preciso utilizarmos
    #RemoteChannels para permitirmos um paralelismo mais "solto", pois

```



```

    remote_do() não retorna nenhum
#valor. assim não temos de fazer uma operação de fetch ou esperarmos o
    valor ser
#retornado, mas também impede que recuperemos o valor sem utilizarmos
    do RemoteChannel ou
#colocarmos o resultado num SharedArray.
#É utilizado o SharedArrays para declarar X e Y pois todos os processos
    utilizam o valor.
for p in procs()
    @async remote_do(work, p, x, y)

    #aqui recuperamos os vetores resultados da aplicação das funções
        calc_real_p_par ou
    #calc_real_p_impar, que estavam armazenados nos RemoteChannels, em
        cada um dos p workers,
    #permitindo que façamos uma operação similar a MPI_Reduce usando do
        operador MPI_SUM.
    #for p in workers()
        g=@fetchfrom p take!(inbox)
    global vetorresults=map(+, vetorresults, g)
end
end

#Aqui declaramos X e Y como SharedArrays e os disponibilizamos em todos
    os processos.
X=SharedArray{Int8}(1, NUM_P)
Y=SharedArray{Int8}(1, NUM_P)

#os vetores a seguir definem nossos dados iniciais que serão processados
    .
vetorp=popula_vetor()
vetorg=popula_vetor()

#A rotina @benchmarkable se tornou necessária utilizar para que o valor
    de vetorresults não
#fosse modificado com cada uma das samples que ele roda. O valor de
    vetorresults é mantido
#uniforme devido ao setup que atribui seu valor para o mesmo de
    vetordezeros, que, como o nome implica,
#é um vetor populado por NUM_P zeros.
b=@benchmarkable begin setup=(global vetorresults=vetordezeros)
    #passagem de vetorg e vetorp para o SharedArray X e SharedArray Y
    @distributed for i=1:NUM_P
        X[1,i]=vetorp[i]
        Y[1,i]=vetorg[i]
    end
    paralelizar(X,Y)

```

```
end memory_tolerance=0.01 time_tolerance=0.01 seconds=1500 samples=20
println(mean(run(b)))
```

Listing A.4 – remote-do.jl

A.0.5 distributed.jl

```
#=
Este programa faz a distribuição de NUM_P elementos dos arrays vetorg e
vetorp para n processos
utilizando @distributed. Esses elementos são processados e retornados ao
processo 1.
=#
using BenchmarkTools
using Distributed
using Statistics
using Random
#Aqui utilizamos 5 processos no total. 4 workers são utilizados porque a
rotina @distributed funciona
#de forma que esta somente utiliza os workers para paralelizar a
computação. Se usássemos 4 processos,
#3 workers, somente 3 processos seriam utilizados enquanto o processo 1
ficaria subaproveitado.
#Essa mudança, utilizando NUM_P=1000000, já dá uma diferença de 100ms. O
programa usando 5 processos
#termina 100ms mais rápido.
addprocs(4)
Random.seed!(1234)
#armazenamos nosso resultado final em um SharedArray que é vetorresults
, permitindo que façamos
#alterações diretamente nesse vetor dentro dos workers, ou seja, sem
precisarmos voltar com os valores
#para o processo 1 com a finalidade de colocá-los num vetor de
resultados.
using SharedArrays
#Utilizamos algo similar a um #define em C para deixar o código mais
limpo e compreensível
#NUM_P se refere ao número de iterações que serão feitas
const NUM_P=1000*10000

function popula_vetor()::Array{Int8}
    return rand(1:100, NUM_P)
end

#essa é a função aleatória que usaremos para teste
@everywhere function calcpint(x,y::Int8)::Float64
    return (sin(cos(x))*(tan(sin(cos(y)))))
end
```

```

#os vetores a seguir definem nossos dados iniciais que serão
    processados.
const vetorp=popula_vetor()
const vetorg=popula_vetor()

#declaramos o SharedArray vetorresults que será o local de
    armazenamento dos resultados
vetorresults=SharedArray{Float64}(NUM_P)

#aqui começamos a medir o tempo gasto
b=@benchmarkable begin
    #aqui atribuímos a casa i de vetorresults, o valor do cálculo
        resultante da casa i de vetorp
    #com a casa i de vetorg passados na caixa-preta, calcpint.
    #como já foi mencionado, os valores são setados diretamente no
        vetorresults pois este é um
    #SharedArray que está incluso em todos os processos.
    #a função @distributed partilha automaticamente as tarefas entre os
        workers.
    @sync @distributed (+) for i in 1:NUM_P
        vetorresults[i]=calcpint(vetorp[i],vetorg[i])
    end
end memory_tolerance=0.01 time_tolerance=0.01 seconds=1500 samples=20
println(median(run(b)))

```

Listing A.5 – distributed.jl

A.0.6 @spawnat-remotechannels.jl

```

#=
Esse programa tem como objetivo utilizar RemoteChannels e @spawnat
    juntos para gerar um pipeline
Utilizando assim tanto de corotinas e do pacote Distributed.
O programa gera um número NUM_P de vetores de N casas populados com
    variáveis aleatórias
para cada um dos dois vetores: vetorg e vetorp.
Esses vetores são gerados no processo 1 e são recuperados pelos demais
    processos
através do take!() de RemoteChannels, sendo extraídos em forma de uma
    tupla.
Nos workers, os vetores são passados pela função calcpint através do
    comando map.
Após terem sido passados pelo work(), os valores encontrados são
    colocados num segundo RemoteChannel,
results, com objetivo de serem recuperados pelo processo 1 para
    armazenamento.
=#

```

```

using BenchmarkTools
using Distributed
using Statistics
using Random
Random.seed!(1234)
#=
Aqui utilizamos 5 processos no total. 4 workers são utilizados porque o
    desempenho entre utilizar 5 e 4 foi comparado e
o melhor resultado foi o que utilizava 5 processos, apesar da diferença
    ter sido pouca.
=#
addprocs(4)
#Utilizamos algo similar a um #define em C para deixar o código mais
    limpo e compreensível
#NUM_P se refere ao número de iterações que serão feitas
const NUM_P=1000
#N se refere ao tamanho dos vetores vetorg e vetorp, tanto quanto de
    vetorresultados
const N=1000
const ta = floor{Int32, NUM_P/nworkers()}
#=
Utilizando os valores de NUM_P e N, é possível ajustar o código para
    algo orientado a ter mais
ou menos trocas de mensagens. Pode ser mais interessante ter poucas
    trocas de mensagens
mas um pipeline mais atrasado. Aumentando N e diminuindo NUM_P, se
    consegue poucas trocas
de mensagens e a mesma quantidade de elementos, porém, um atraso na
    saída.
=#

#aqui são declarados os RemoteChannels que serão usados
#seu propósito é permitir uma comunicação facilitada entre processos
    permitindo paralelismo
#I/O sem problemas
const inbox = RemoteChannel{()>Channel{}}(300)

const results = RemoteChannel{()>Channel{}}(300)

function popula_vetor()::Array{Int8}
    return rand(1:100, N)
end

@everywhere function calcpint(x,y::Int8)::Float64
    return (sin(cos(x))*(tan(sin(cos(y)))))
end

```

```

#esta função tem como objetivo servir como um encapsulamento para os
    argumentos e funções envolvidos
@everywhere function work(inbox, results)
    #um while que serve para ter certeza que inbox possui alguma coisa
    while !isready(inbox)
        wait(inbox)
    end
    b=take!(inbox)
    put!(results, (map(calcpint, b[1],b[2]),b[3]))
end

#Esta função é mais um encapsulamento para facilitar a medida do tempo
    usando o pacote BenchmarkTools
function englobe(inbox, results)

    #aqui, os workers passam os valores de vetorg e vetorp pela função
        calcpint.
    #iniciamos isso antes da produção, caso contrário, teríamos um deadlock
        com o processo 1
    #chegando ao limite do remotechannel e depois, esperando um take!, que
        nunca aconteceria.
    for i in 1:ta
        for p in workers()
            R=[@spawnat p work(inbox, results)]
        end
    end

    #aqui, caso NUM_P não seja divisível por nworkers(), é processado o
        resto(dif) pelo
    #último processo adicionado, o processo número nprocs().
    if dif!=0
        @spawnat nprocs() begin
            for i in 1:dif
                work(inbox, results)
            end
        end
    end

    #Parte do código referida ao processo 1 que lida com a população de
        vetorg e vetorp com
    #números aleatórios entre 1 e 100. A dimensão dos vetores é determinada
        por N.
    #Neste código, o processo 1 é o produtor enquanto os workers são os
        consumidores.

    for i in 0:NUM_P-1
        vetorp=popula_vetor()
    end

```

```

vetorg=popula_vetor()
#envia os valores de vetorp e vetorg gerados de tamanho N para o canal
  inbox
#o 'i' também é enviado, permitindo que o vetorresults não perca sua
  ordenação.
#O motivo de i variar entre 0 e NUM_P-1, é para a ordem do vetor ser
  mantida
#por razões matemáticas.
put!(inbox, (vetorp, vetorg, i))
end

#Parte do código referida ao processo 1. Responsável por recuperar os
  resultados calculados
#e armazená-los num vetor.
for i in 1:NUM_P
  while !isready(results)
    wait(results)
  end

  buffer=take!(results)
  #buffer[2] é o 'i' armazenado anteriormente. Ele é utilizado para
    determinar a ordenação do vetorresults.
  for j in 1:N
    vetorresults[buffer[2]*N+j]=buffer[1][j]
  end
end
end

#aqui determinamos se dif=0 ou não. se NUM_P/nworkers() não for um
  inteiro, precisamos compensar isso
#para podermos fazer o paralelismo adequado sem deixar de fazer o número
  exato de repetições NUM_P
if (NUM_P/nworkers()-floor(Int32, NUM_P/nworkers()))!=0)
  const dif=NUM_P-ta*nworkers()
else const dif=0
end

vetorresults=zeros(Float64, NUM_P*N)

b=@benchmarkable englobe(inbox, results) memory_tolerance=0.01
  time_tolerance=0.01 seconds=1500 samples=20
println(mean(run(b)))

```

Listing A.6 – @spawnat-remotechannels.jl

APÊNDICE B – Códigos de C

B.0.1 exemplo-soma-vetor.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define NUM_P 50000000
int* gera_vetor() {
    return (int*)malloc(NUM_P*sizeof(int));
}
int main(int argc, char *argv[]){
    int* vetor1; int* vetor2; int* vetorsoma;
    srand(time(NULL));
    vetor1=gera_vetor(); vetor2=gera_vetor(); vetorsoma=gera_vetor();
    for(int i=0; i<NUM_P; i++){
        vetor1[i]=rand() % 100+1;
        vetor2[i]=rand() % 100+1;}
    int repeticao=20;
    clock_t Ticks[2];
    Ticks[0]=clock();
    while(repeticao>0){
        for(int i=0; i<NUM_P; i++){
            vetorsoma[i]=vetor1[i]+vetor2[i];}
        repeticao-=1;
    }
    Ticks[1]=clock();
    double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
    printf("Tempo de execução médio:%fms\n", Tempo/20);
    free(vetor1); free(vetor2); free(vetorsoma);
    return 0;
}
```

Listing B.1 – exemplo-soma-vetor.c

B.0.2 exemplo-send-receive.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <time.h>
#define NUM_P 10

double func_soma(int x, int y){
    return x+y;
}
```



```
int main(int argc, char *argv[])
{
    int dif=0, ta, rank, size, type=99;
    double tata;

    srand(time(NULL));

    int vetor1[NUM_P];
    int vetor2[NUM_P];

    for(int i=0; i<NUM_P; i++){
        vetor1[i]=rand() % 100+1;
        vetor2[i]=rand() % 100+1;
    }

    double vetorsoma[NUM_P];

    MPI_Init(&argc,&argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (NUM_P<(size)) {
        printf("Erro: Foi entrado um valor de dados NUM_P inferior ao numero
            de workers.\n");
        return 0;
    }

    tata=(double)NUM_P/size;
    ta=(int)tata;

    if (tata-(double)ta!=0 && rank==0) {
        dif=NUM_P-ta*size;
    }

    double vetorprovisorio[ta];

    clock_t Ticks[2];
    Ticks[0]=clock();

    if (rank==0){
        for (int i=1; i<size; i++){
            MPI_Send(&vetor2[(i-1)*ta], ta, MPI_INT, i, type, MPI_COMM_WORLD);
            MPI_Send(&vetor1[(i-1)*ta], ta, MPI_INT, i, type, MPI_COMM_WORLD);
```

```

    }
}
else{
    MPI_Recv(&vetor2, ta, MPI_INT, 0, type, MPI_COMM_WORLD, &status);
    MPI_Recv(&vetor1, ta, MPI_INT, 0, type, MPI_COMM_WORLD, &status);
}

if (rank!=0) {
    for(int i=0; i<ta; i++){
        vetorprovisorio[i]=func_soma(vetor1[i],vetor2[i]);
    }
}

if (rank!=0) {
    MPI_Send(&vetorprovisorio, ta, MPI_DOUBLE, 0, type, MPI_COMM_WORLD);
}
else{
    for (int i = 0; i<ta+dif; i++) {
        vetorsoma[NUM_P-ta-dif+i]=func_soma(vetor1[NUM_P-ta-dif+i],vetor2[
        NUM_P-ta-dif+i]);
    }
    for (int i = 1; i < size; i++) {
        MPI_Recv(&vetorsoma[(i-1)*ta], ta, MPI_DOUBLE, i, type,
        MPI_COMM_WORLD, &status);
    }
}

MPI_Finalize();

if (rank==0) {
    Ticks[1]=clock();
    double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
    printf("Tempo de execução:%fms\n", Tempo);

    printf("vetor1=");
    for (int i = 0; i < NUM_P; i++) {
        printf("%d ", vetor1[i]);
    }
    printf("\n");

    printf("vetor2=");
    for (int i = 0; i < NUM_P; i++) {
        printf("%d ", vetor2[i]);
    }
    printf("\n");
}

```

```
printf("vetorsoma=");
for (int i = 0; i < NUM_P; i++) {
    printf("%d ", (int)vetorsoma[i]);
}
printf("\n");

}
return 0;
}
```

Listing B.2 – exemplo-send-receive.c

B.0.3 exemplo-comunicacao-ponto-a-ponto.c

```
// Exemplo básico de comunicação ponto-a-ponto. O processo 0 utiliza a
// rotina
// MPI_Send para enviar um número ao processo 1.

#include <mpi.h>

int main(int argc, char *argv[]) {

    // Inicializando o ambiente MPI.
    MPI_Init(&argc, &argv);
    MPI_Status status;

    // Declarando rank e size.
    int rank, size;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // We are assuming at least 2 processes for this task
    if (size < 2) {
        fprintf(stderr, "World size must be greater than 1 for %s\n", argv
[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int number;

    // Somente o processo 0 entra dentro desse if.
    if (rank == 0) {
        number = 10;
        // O processo 0 envia os dados para o processo 1.
        MPI_Send(
            /* data          = */ &number,
            /* count         = */ 1,
```

```

    /* datatype      = */ MPI_INT ,
    /* destination   = */ 1,
    /* tag           = */ 0,
    /* communicator = */ MPI_COMM_WORLD);
}

// Somente o processo 1 entra dentro desse if.
else if (rank == 1) {
    MPI_Recv(
        /* data      = */ &number,
        /* count     = */ 1,
        /* datatype   = */ MPI_INT,
        /* source     = */ 0,
        /* tag        = */ 0,
        /* communicator = */ MPI_COMM_WORLD,
        /* status     = */ MPI_STATUS_IGNORE);

    printf("Process 1 received number %d from process 0\n", number);
}

MPI_Finalize();
}

```

Listing B.3 – exemplo-comunicacao-ponto-a-ponto.c

B.0.4 send-receive.c

```

//Este programa faz a distribuição de NUM_P elementos dos arrays vetorg
e vetorp para size processos
//utilizando MPI_Send e MPI_Recv. Esses elementos são processados e
retornados ao processo mestre através
//do MPI_Send e MPI_Recv.
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <time.h>
#define NUM_P 1000*1000

double calc_real_p(short int x, short int y){
    return (sin(cos(x))*(tan(sin(cos(y)))));
}

int main(int argc, char** argv)
{
    //declaração de variáveis auxiliares
    int dif=0, ta, rank, size, type=99;
    double tata;

```

```
//criando uma seed para o random
srand(time(NULL));

//os vetores a seguir definem nossos dados iniciais que serão
    processados.
short int *vetorp=(short int*)malloc(NUM_P*sizeof(short int));
short int *vetorg=(short int*)malloc(NUM_P*sizeof(short int));

double *vetorresultado=(double*)malloc(NUM_P*sizeof(double));

//declarações de MPI
MPI_Init(&argc, &argv);
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

//Populamos os vetores através de loops simples.
//Para termos os mesmo números que o random contraparte em Julia,
    restringimos o valor que eles
//podem tomar para [1..100]
//Utilizamos um if antes de popularmos os vetores. O processo 0 é o
    único que possui esses valores,
//garantindo que SE E SOMENTE SE ocorrer o envio de mensagem pelo
    processo 0 aos workers, eles terão acesso
//aos valores produzidos.
if (rank==0) {
    for(int i=0; i<NUM_P; i++){
        vetorp[i]=rand() % 100+1;
        vetorg[i]=rand() % 100+1;
    }
}

if (NUM_P<(size)) {
    printf("Erro: Foi entrado um valor de dados NUM_P inferior ao numero
        de workers.\n");
    return 0;
}

//Encontramos o tamanho para podermos dividir os dados do vetor para
    que possam ser
//enviados para os workers.
//Utilizamos tata que é double caso a divisão NUM_P/size seja inexata
//Passamos a ta o valor inteiro de tata para podermos compara-los mais
    a frente
tata=(double)NUM_P/size;
```

```

ta=(int)tata;

//Verificando se sobrou resto na divisão NUM_P/size, caso tenha sobrado
//signifca que o processo 0 deverá pegar a porção dele de dados normal
//'ta' e mais um adicional que será dado por 'dif'
if (tata-(double)ta!=0 && rank==0) {
    dif=NUM_P-ta*size;
}

double *vetorprovisorio=(double*)malloc(ta*sizeof(double));

int repeticao=20;
//Iniciando a medição do tempo de execução
clock_t Ticks[2];
Ticks[0]=clock();
while (repeticao>0) {
    repeticao--;

    //Enviando vetorg e vetorp para os workers através de mensagens simples
    if (rank==0){
        for (int i=1; i<size; i++){
            MPI_Send(&vetorg[(i-1)*ta], ta, MPI_SHORT, i, type, MPI_COMM_WORLD);
            MPI_Send(&vetorp[(i-1)*ta], ta, MPI_SHORT, i, type, MPI_COMM_WORLD);
        }
    }
    else{
        //os demais processos recebem do processo 0 suas porções dos dados
        MPI_Recv(vetorg, ta, MPI_SHORT, 0, type, MPI_COMM_WORLD, &status);
        MPI_Recv(vetorp, ta, MPI_SHORT, 0, type, MPI_COMM_WORLD, &status);
    }

    //Aqui executamos a caixa preta em cada um dos workers
    if (rank!=0) {
        for(int i=0; i<ta; i++){
            vetorprovisorio[i]=calc_real_p(vetorp[i],vetorg[i]);
        }
    }

    //Os workers retornam os valores encontrados para o processo 0
    if (rank!=0) {
        MPI_Send(vetorprovisorio, ta, MPI_DOUBLE, 0, type, MPI_COMM_WORLD);
    }
    else{
        //executamos a caixa preta no processo 0 e o resultado do calculo vai
        //ser armazenado direto no vetorresultado
        for (int i = 0; i<ta+dif; i++) {
            vetorresultado[NUM_P-ta-dif+i]=calc_real_p(vetorp[NUM_P-ta-dif+i],

```

```

    vetorg[NUM_P-ta-dif+i]);
}
//o processo 0 recebe os valores encontrados nos workers
for (int i = 1; i < size; i++) {
    MPI_Recv(&vetorresultado[(i-1)*ta], ta, MPI_DOUBLE, i, type,
    MPI_COMM_WORLD, &status);
}
}

MPI_Finalize();
//Paramos o relógio e imprimimos o tempo de execução
Ticks[1]=clock();
double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
if (rank==0) {
    printf("Tempo de execução médio:%fms\n", Tempo/20);
}
return 0;
}

```

Listing B.4 – send-receive.c

B.0.5 scatter-gather.c

```

//Este programa faz a distribuição de NUM_P elementos dos arrays vetorg
    e vetorp para size processos
//utilizando MPI_Scatter. Esses elementos são processados e retornados
    ao processo mestre através
//do MPI_Gather.
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <time.h>
#define NUM_P 1000*1000
double calc_real_p(int x, int y){
    return (sin(cos(x))*(tan(sin(cos(y)))));}
int main(int argc, char *argv[]){
    //declaração de variáveis auxiliares
    int dif=0, ta, rank, size, type=99;
    double tata;
    //criando uma seed para o random
    srand(time(NULL));
    //declarações de MPI
    MPI_Init(&argc,&argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
}

```

```

if (NUM_P<(size)) {
    printf("Erro: Foi entrado um valor de dados NUM_P inferior ao numero
        de workers.\n");
    return 0;}
//Encontramos o tamanho para podermos dividir os dados do vetor para
    que possam ser
//enviados para os workers.
//Utilizamos tata que é double caso a divisão NUM_P/size seja inexata
//Passamos a ta o valor inteiro de tata para podermos compara-los mais
    a frente
tata=(double)NUM_P/size;
ta=(int)tata;
//Verificando se sobrou resto na divisão NUM_P/size, caso tenha sobrado
//significa que o processo 0 deverá pegar a porção dele de dados normal
//'ta' e mais um adicional que será dado por 'dif'
if (tata-(double)ta!=0 && rank==size-1) {
    dif=NUM_P-ta*size;}
//os vetores a seguir definem nossos dados iniciais que serão
    processados.
//Populamos os vetores através de loops simples.
//Para termos os mesmo números que o random contraparte em Julia,
    restringimos o valor que eles
//podem tomar para [1..100]
double *vetorprovisorio=(double*)malloc(ta*sizeof(double));
short int *vetorp=(short int*)malloc(NUM_P*sizeof(short int));
short int *vetorg=(short int*)malloc(NUM_P*sizeof(short int));
if (rank==size-1) {
    for(int i=0; i<NUM_P; i++){
        vetorp[i]=rand() % 100+1;
        vetorg[i]=rand() % 100+1;}
}
//Iniciando vetores de apoio para os quais o scatter irá passar a
    quantidade ta
//de elementos tanto de vetorg para vetorgg quanto de vetorp para
    vetorpp
short int *vetorpp=(short int*)malloc((ta)*sizeof(short int));
short int *vetorgg=(short int*)malloc((ta)*sizeof(short int));
double *vetorresultado=(double*)malloc(NUM_P*sizeof(double));
int repeticao=20;
//Iniciando a medição do tempo de execução
clock_t Ticks[2];
Ticks[0]=clock();
while (repeticao>0) {
    repeticao-=1;
    //Enviando vetorg e vetorp para os workers através de MPI_Scatter
    //O envio ocorre de um vetorg para o vetorgg e de um vetorp para o
        vetorpp;

```



```

//isso ocorre porque o processo mestre também envia para a sua própria
//instância do vetor
//escolhido a receber os dados. Logo o processo mestre fazendo o
//scatter em vetorg
//enviaria do próprio vetorg do processo mestre para o vetorg do
//processo mestre,
//sendo o vetorg do processo mestre então o vetor que provê os dados e
//o vetor que os receberia.
//Por esses motivos, é uma operação ilegal no sistema.
MPI_Scatter(vetorg, ta, MPI_SHORT, vetorgg, ta, MPI_SHORT, size-1,
MPI_COMM_WORLD);
MPI_Scatter(vetorp, ta, MPI_SHORT, vetorpp, ta, MPI_SHORT, size-1,
MPI_COMM_WORLD);
//Aqui executamos a caixa preta em cada um dos workers
for(int i=0; i<ta; i++){
    vetorprovisorio[i]=calc_real_p(vetorpp[i], vetorgg[i]);}
//O processo mestre executa a caixa preta nos elementos que não foram
//distribuídos por
//terem ficado como um resto na divisão inteira para calcular 'ta'
if (rank==size-1 && dif!=0) {
    for (int i = 0; i < dif; i++) {
        vetorresultado[NUM_P-dif+i]=calc_real_p(vetorp[NUM_P-dif+i], vetorg[
NUM_P-dif+i]);}
}

//Após o cálculo dos valores, voltamos com todos os valores para o
//processo mestre
MPI_Gather(vetorprovisorio, ta, MPI_DOUBLE, vetorresultado, ta,
MPI_DOUBLE, size-1, MPI_COMM_WORLD);
}
MPI_Finalize();
//Paramos o relógio e imprimimos o tempo de execução
Ticks[1]=clock();
double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
if (rank==size-1) {
    printf("Tempo de execução:%fms\n", Tempo/20);}
return 0;
}

```

Listing B.5 – scatter-gather.c

B.0.6 bcast-reduce.c

```

//Este programa distribui os vetores de dados vetorg e vetorp para os
//workers através de
//MPI_Bcast e, dependendo se o rank do processo for par ou ímpar,
//executa uma operação específica.

```

```

//No final, soma todos os resultados através de MPI_Reduce para um único
    vetorresultado no processo
//mestre.
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <time.h>
#define NUM_P 1000*20000
double calc_real_p_par(int x, int y){
    return (sin(cos(x))*(tan(sin(cos(y)))));}
double calc_real_p_impar(int x, int y){
    return (sin(cos(x))*(tan(sin(cos(y))))*1/tan(sin(cos(x*y)));}

int main(int argc, char *argv[])
{
    //declaração de variáveis auxiliares
    int rank, size, type=99;
    //criando uma seed para o random
    srand(time(NULL));
    //os vetores a seguir definem nossos dados iniciais que serão
        processados.
    //Populamos os vetores através de loops simples.
    //Para termos os mesmo números que o random contraparte em Julia,
        restringimos o valor que eles
    //podem tomar para [1..100]

    //declarações de MPI
    MPI_Init(&argc,&argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    //Encontramos o tamanho para podermos dividir os dados do vetor para
        que possam ser

    double *vetorprovisorio=(double*)malloc(NUM_P*sizeof(double));

    short int *vetorp=(short int*)malloc(NUM_P*sizeof(short int));
    short int *vetorg=(short int*)malloc(NUM_P*sizeof(short int));

    if (rank==size-1) {
        for(int i=0; i<NUM_P; i++){
            vetorp[i]=rand() % 100+1;
            vetorg[i]=rand() % 100+1;
        }
    }
}

```

```

double *vetorresultado=(double*)malloc(NUM_P*sizeof(double));

int repeticao=20;
//Iniciando a medição do tempo de execução
clock_t Ticks[2];
Ticks[0]=clock();
while (repeticao>0) {
    repeticao-=1;

    //Enviando vetorg e vetorp para os workers através de MPI_Bcast
    MPI_Bcast(vetorg, NUM_P, MPI_SHORT, size-1, MPI_COMM_WORLD);
    MPI_Bcast(vetorp, NUM_P, MPI_SHORT, size-1, MPI_COMM_WORLD);

    //Aqui executamos a caixa preta em cada um dos processos dependendo do
    //seu número do rank
    //ser par ou ímpar.

    if (rank%2==0) {
        for(int i=0; i<NUM_P; i++){
            //printf("rank[%d] -> Vetorp[%d]=%d e Vetorg[%d]=%d\n", rank , i,
            //vetorp[i], i, vetorg[i]);
            vetorprovisorio[i]=calc_real_p_par(vetorp[i], vetorg[i]);
            //printf("rank[%d] -> vetorprovisorio[%d]=%f\n", rank , i,
            //vetorprovisorio[i]);
        }
    }
    else{
        for(int i=0; i<NUM_P; i++){
            //printf("rank[%d] -> Vetorp[%d]=%d e Vetorg[%d]=%d\n", rank , i,
            //vetorp[i], i, vetorg[i]);
            vetorprovisorio[i]=calc_real_p_impair(vetorp[i], vetorg[i]);
            //printf("rank[%d] -> vetorprovisorio[%d]=%f\n", rank , i,
            //vetorprovisorio[i]);
        }
    }

    //Após o cálculo dos valores, voltamos com todos os valores para o
    //processo mestre
    MPI_Reduce(vetorprovisorio, vetorresultado, NUM_P, MPI_DOUBLE, MPI_SUM,
        size-1, MPI_COMM_WORLD);
}
MPI_Finalize();
Ticks[1]=clock();
double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
if (rank==size-1) {
    //Paramos o relógio e imprimimos o tempo de execução

```

```

    printf("Tempo de execução:%fms\n", Tempo/20);
}
return 0;
}

```

Listing B.6 – bcst-reduce.c

B.0.7 scatter-reduce.c

```

//Faz uma operação complexa de NUM_P elementos, que são distribuídos
    entre size processos através
//do comando MPI_Scatter.
//Reduz todos os valores para um só valor, armazenado em global_sum,
    através de MPI_Reduce.
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <time.h>
#define NUM_P 1000*1000
double calc_real_p(short int x, short int y){
    return (sin(cos(x))*(tan(sin(cos(y)))));}
int main(int argc, char *argv[]){
    //declaração de variáveis auxiliares
    int dif=0, ta, rank, size, type=99;
    double tata;
    //criando uma seed para o random
    srand(time(NULL));
    //declarações de MPI
    MPI_Init(&argc,&argv);
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (NUM_P<(size)) {
        printf("Erro: Foi entrado um valor de dados NUM_P inferior ao numero
            de workers.\n");
        return 0;}
    //Encontramos o tamanho para podermos dividir os dados do vetor para
        que possam ser
    //enviados para os workers.
    //Utilizamos tata que é double caso a divisão NUM_P/size seja inexata
    //Passamos a ta o valor inteiro de tata para podermos compara-los mais
        a frente
    tata=(double)NUM_P/size;
    ta=(int)tata;
    //Verificando se sobrou resto na divisão NUM_P/size, caso tenha sobrado
    //significa que o processo 0 deverá pegar a porção dele de dados normal
    //'ta' e mais um adicional que será dado por 'dif'

```

```

if (tata-(double)ta!=0 && rank==size-1) {
    dif=NUM_P-ta*size;}
double local_sum=0;
double global_sum=0;
//os vetores a seguir definem nossos dados iniciais que serão
    processados.
//Populamos os vetores através de loops simples.
//Para termos os mesmo números que o random contraparte em Julia,
    restringimos o valor que eles
//podem tomar para [1..100]
short int *vetorp=(short int*)malloc(NUM_P*sizeof(short int));
short int *vetorg=(short int*)malloc(NUM_P*sizeof(short int));
if (rank==size-1) {
    for(int i=0; i<NUM_P; i++){
        vetorp[i]=rand() % 100+1;
        vetorg[i]=rand() % 100+1;}
}
//Iniciando vetores de apoio para os quais o scatter irá passar a
    quantidade ta
//de elementos tanto de vetorg para vetorgg quanto de vetorp para
    vetorpp
short int *vetorpp=(short int*)malloc(ta*sizeof(short int));
short int *vetorgg=(short int*)malloc(ta*sizeof(short int));
int repeticao=20;
//Iniciando a medição do tempo de execução
clock_t Ticks[2];
Ticks[0]=clock();
while(repeticao>0){
    repeticao-=1;
    //Enviando vetorg e vetorp para os workers através de MPI_Scatter
    //O envio ocorre de um vetorg para o vetorgg e de um vetorp para o
        vetorpp;
    //isso ocorre porque o processo mestre também envia para a sua própria
        instância do vetor
    //escolhido a receber os dados. Logo o processo mestre fazendo o
        scatter em vetorg
    //enviaria do próprio vetorg do processo mestre para o vetorg do
        processo mestre,
    //sendo o vetorg do processo mestre então o vetor que provê os dados e
        o vetor que os receberia.
    //Por esses motivos, é uma operação ilegal no sistema.
MPI_Scatter(vetorg, ta, MPI_SHORT, vetorgg, ta, MPI_SHORT, size-1,
    MPI_COMM_WORLD);
MPI_Scatter(vetorp, ta, MPI_SHORT, vetorpp, ta, MPI_SHORT, size-1,
    MPI_COMM_WORLD);
//Aqui executamos a caixa preta em cada um dos processos
for(int i=0; i<ta; i++){

```

```

    local_sum+=calc_real_p(vetorpp[i], vetorgg[i]);}
    //0 processo mestre executa a caixa preta nos elementos que não foram
    distribuídos por
    //terem ficado como um resto na divisão inteira para calcular 'ta'.
    if (rank==size-1 && dif!=0) {
        for (int i = 0; i < dif; i++) {
            local_sum+=calc_real_p(vetorp[NUM_P-dif+i], vetorg[NUM_P-dif+i]);}
        }
    //Após o cálculo dos valores, usamos o MPI_Reduce para encontrarmos o
    valor final de global_sum
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, size-1,
        MPI_COMM_WORLD);
}
MPI_Finalize();
//Paramos o relógio e imprimimos o tempo de execução
Ticks[1]=clock();
double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
if (rank==size-1) {
    printf("Tempo de execução médio:%fms\n", Tempo/20);
    printf("Valor da soma global= %f\n", global_sum);}
return 0;
}

```

Listing B.7 – scatter-reduce.c

B.0.8 simulacao-remote-channels.c

```

/*
Esse programa tem como objetivo simular e utilizar RemoteChannels para
gerar um pipeline.
O programa gera um número NUM_P de vetores de N casas populados com
variáveis aleatórias
para cada um dos dois vetores: vetorg e vetorp.
Esses vetores são gerados no processo '0' e são recuperados pelos demais
processos através da troca de
mensagens por MPI.
Nos processos 1:n, os vetores vetorg e vetorp têm cada um de seus
elementos passados pela função
calc_real_p.
Após terem sido passados pela função, os valores encontrados são
devolvidos ao processo 0.
*/
#include <stdbool.h>
#include <stdio.h>
#include <mpi.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

```

```

#include <time.h>
#define NUM_P 1000
#define N 10000
double calc_real_p(int x, int y){
    return (sin(cos(x))*(tan(sin(cos(y)))));}
int main(int argc, char** argv) {
    //declaração de variáveis auxiliares
    int rank, size;
    //declarações de MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    double *vetorresultado=(double*)malloc((N*NUM_P)*sizeof(double));
    int repeticao=20;
    //Iniciando a medição do tempo de execução
    clock_t Ticks[2];
    Ticks[0]=clock();
    while(repeticao>0){
        repeticao--;
        //Produtor
        if (rank == 0) {
            srand(time(NULL));
            short int *vetorp=(short int*)malloc(N*sizeof(short int));
            short int *vetorg=(short int*)malloc(N*sizeof(short int));
            double *temp=(double*)malloc((N)*sizeof(double));
            for (int j = 1; j <= NUM_P; ++j) {
                MPI_Status status;
                //populando vetorg e vetorp para ser passado a um dos demais
                //processos.
                for (int i = 0; i<N; i++) {
                    vetorg[i]=rand()%100+1;
                    vetorp[i]=rand()%100+1;}
                //Aqui recebe-se o vetor temp com os valores já passados por
                //calc_real_p.
                //Caso seja a primeira vez do consumidor entrando em contato com o
                //produtor,
                //essa mensagem serve somente para ligá-los e, após a primeira
                //comunicação,
                //passa a ter dados reais até o consumidor ser desligado.
                MPI_Recv(temp, N, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
                //Na primeira comunicação, o TAG virá como 0, este if serve para
                //separar a primeira comunicação que não deverá ter seu valor cometido
                //ao vetorresultado das demais.
                if (status.MPI_TAG > 0) {
                    for (int i=0; i<N; i++) {
                        vetorresultado[((status.MPI_TAG-1)*N)+i]=temp[i];}

```

```

}
//Aqui enviamos os vetores vetorg e vetorp para os demais processos.
MPI_Send((vetorg), N, MPI_SHORT, status.MPI_SOURCE, j,
MPI_COMM_WORLD);
MPI_Send((vetorp), N, MPI_SHORT, status.MPI_SOURCE, j,
MPI_COMM_WORLD);
}
//Aqui são rodadas as últimas execuções recebendo os últimos vetores
e "encerrando o canal" com os demais processos.
int num_terminated = 0;
for (int num_terminated = 0; num_terminated < size-1; num_terminated
++) {
    MPI_Status status;
    MPI_Recv(temp, N, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
    if (status.MPI_TAG > 0) {
        for (int i=0; i<N; i++) {
            vetorresultado[((status.MPI_TAG-1)*N)+i]=temp[i];
        }
        //Aqui enviamos os sinais com tag 0 para indicar desligamento dos
processos
        MPI_Send(vetorg, N, MPI_SHORT, status.MPI_SOURCE, 0, MPI_COMM_WORLD)
;
        MPI_Send(vetorp, N, MPI_SHORT, status.MPI_SOURCE, 0, MPI_COMM_WORLD)
;
    }
}
//Consumidores
else {
    short int *vetorp=(short int*)malloc(N*sizeof(short int));
    short int *vetorg=(short int*)malloc(N*sizeof(short int));
    double *temp=(double*)malloc((N)*sizeof(double));
    //0 sinal abaixo funciona como um greeting para o produtor. Serve
para que eles possam estabelecer um canal entre eles.
    MPI_Send(temp, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    bool terminated = false;
    do {
        MPI_Status status;
        MPI_Recv(vetorg, N, MPI_SHORT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
status);
        MPI_Recv(vetorp, N, MPI_SHORT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
status);
        //Aqui, caso TAG=0, isso indica o canal sendo fechado e o fim do
loop.
        if (status.MPI_TAG == 0) {
            terminated = true;
        }
        else {

```



```
//Aqui armazenamos em temp os valores passados pela função
calc_real_p
for (int i=0; i<N; i++) {
    temp[i]=calc_real_p(vetorg[i], vetorp[i]);
}
MPI_Send(temp, N, MPI_DOUBLE, 0, status.MPI_TAG, MPI_COMM_WORLD);}
} while (!terminated);
}
}
MPI_Finalize();
if (rank == 0) {
    Ticks[1]=clock();
    double Tempo = (Ticks[1]-Ticks[0]) * 1000.0 / CLOCKS_PER_SEC;
    printf("Tempo de execução médio:%fms\n", Tempo/20);}
}
```

Listing B.8 – simulacao-remote-channels.c

APÊNDICE C – Código de Python

C.0.1 soma-vetor.py

```
import time
import random
NUM_P=20000000
def popula_vetor(x):
    for _ in range(NUM_P):
        x.append(random.randint(1,100))
vetor1=[]
vetor2=[]
vetorsoma=[]
tempofinal=0.0
popula_vetor(vetor1)
popula_vetor(vetor2)
for _ in range(20):
    inicio = time.time()
    for elemvetor1, elemvetor2 in zip(vetor1, vetor2):
        vetorsoma.append(elemvetor1 + elemvetor2)
    fim = time.time()
    tempofinal+=fim-inicio
print("Tempo de execucao:", (tempofinal/20)*1000,"ms.")
```

Listing C.1 – soma-vetor.py