Atividade 6a - Programando em Paralelo: MPI aplicado em processamento de dados

Gabriel Thiago Henrique Dos Santos <ra107774@uem.br>

***Abstract*—O estudo de paralelismo tende a ser importante para diversos cenários, como em momentos nos quais há a necessidade e possibilidade de executar o mesmo serviço com diferentes cenários, no nosso caso é utilizado para o processamento de tarefas e o paralelismo para arquiteturas com memória distribuída. Para a análise dos experimentos, fora desenvolvido cinco arquivos de entrada e juntamente analisado tanto para o código sequencial quanto com MPI o uso de trabalhadores, no caso de dois a quatro trabalhadores. Para melhores análises foram utilizadas da ferramenta *Tau* para uma melhor consulta.**

***Index Terms*—Latex template, IEEE, Latin America Transactions, guidelines for authors.**

I. Introdução

EM diversas aplicações, o uso de execuções simultâneas podem ser aplicados, e com isso se tem o objetivo principal de melhorar o desempenho e aproveitamento do hardware [1].

Para este trabalho implementamos para execução do problema o código para memória distribuída, utilizando a biblioteca MPI, com auxilio da documentação e exemplos em [2] e os tutoriais de [3] e [4], estes processos que serão executados em memória distribuída devem estar interligado entre alguma rede (cabo, *switch*, *cluster*...) em que cada processo então tera sua própria memória. É iniciada o primeiro processo na máquina e este dispara novos processos em outros máquinas, sabendo já quantas máquinas existem e as mandaria, no caso do nosso trabalho os processos foram realizado na mesma máquina.

O seguinte relatório está organizado por sessões, no qual a II descreve sobre o problema e seus resultados esperados, em seguida é abordado sobre o desenvolvimento e suas devidas explicações. Em III há a abordagem realizada para em seguida houve a análise sobre os resultados. Ao final, temos a conclusão seguido pelas referencias utilizadas para o desenvolvimento do trabalho.

II. Objetivos

O principal objetivo deste trabalho é ganhar experiência com paralelismo para arquitetura com memória distribuída, incluindo exposição a conceitos de passagem de mensagem, utilizando a biblioteca MPI. Como objetivos secundários, têm-se o conhecimento e experiência na implementação de um modelo de fila de tarefas mestre/trabalhador de computação paralela.

III. Descrição do Problema

O problema se baseia na leitura de uma lista de "tarefas", de um arquivo, em que cada tarefa consiste em um código de carácter, que indica uma ação, e um número que representa o tempo que esta ação é exigida.

O código "p" faz referencia à "processar", sendo o processamento de uma tarefa por algum trabalhador, e então o número n na frente significa o tempo que deve-se esperar, utilizando da função *sleep* na implementação para a simulação.

Para o código "e", de "esperar", faz referencia a simulação de uma pausa nas tarefas de entrada pelo mestre, sendo então o número n a frente com o tempo de espera, novamente implementado para a simulação a função *sleep*.

Deve-se também realizar o armazenamento de algumas variáveis agregadas, sendo a soma do tempo de todos os processamentos; a contagem de números de tempo impares; e o tempo mínimo e máximo dos processamento; Ao final então sendo mostrado ao usuário.

IV. Desenvolvimento

Utilizamos da linguagem C para nossa implementação, para o modo sequencial houve a disponibilização juntamente com as especificações, então fora desenvolvido o com uso do paralelismo de memória distribuída, a seguir apresentamos algumas escolhas e estruturas utilizadas no código.

- `Fila de tarefas`: criado uma fila dinâmica *queue q* com o tipo *type queue* para armazenas as tarefas a serem processadas pelos trabalhadores;
- `Buffer auxiliar`: fora criado uma matriz para o *Buffer*, com a mesma quantidade de linhas do arquivo de instruções e duas colunas, para o armazenamento da lista instruções do arquivo, em que para cada linha é armazenado a ação e o tempo de processamento. Fora desenvolvido para que possa percorrer por *Index* as linhas desejadas;
- `Macro`: para a melhora do processamento foram selecionados as funções de criação da fila e verificação de fila vazia, e definidas ao inicio do código, com o uso do *#define*, o pré processamento não ocupa memória da execução e as referencias é alterada na chamada de cada, melhorando o desempenho e também a visualização do código.
- `Uso de TAGS`: na troca de mensagem entre o trabalhador e o mestre, fora definido três tipos de mensagem:
  - **0:** envio do mestre das tarefas para o trabalhador ocioso;
  - **1:** o trabalhador ocioso envia ao mestre o seu *world_id*, ou seja, o seu id para então o mestre saber para quem deve enviar a próxima *task*;
  - **2:** ao final dos processamentos há o envio dos resultados das variáveis agregadas de cada trabalhador,

e com isso enviado em um vetor de quatro inteiros do trabalhador ao mestre.

Para além destas estruturas comentadas, a lógica de implementação seguida para esta implementação segue, inicialmente, pelo Mestre, entra-se no primeiro *loop*, no qual enquanto não finalizou as instruções, é lido o próximo *Index*, é pego a *task* da vez e caso seja "e" mantem o sleep e segue para o próximo, caso seja "p" inserido na fila de tarefas.

Tendo que cada trabalhador deve-se avisar ao mestre que está ocioso, passando na mensagem com a *tag* 1 o seu *ID*, para que o mestre então saiba para quem enviar.

O Mestre então, após inserir na fila, utiliza do *MPI_Irevc*, sendo esta a função não bloquente para receber uma comunicação, e então verifica com o *MPI_Test* se já houve algum envio, ou seja, se há algum trabalhador ocioso que está enviando o seu *ID*, se ainda não houver trabalhador ocioso, é seguido em outro *loop*, para o mestre continuar lendo o arquivo de instruções, no nosso caso, sendo o *buffer*.

Neste *loop* é lido a próxima linha (*Index*), tendo três possibilidades, se já acabou o arquivo é setado a *flag* de final de arquivo e sai dos *loops*, se não, a próxima *task* pode ser "p", o qual é inserido na fila de tarefa, ou se for "e", ocorre o *sleep*. Após então é executado novamente o *MPI_Test*, para verificar se já houve envio de algum trabalhador avisando que está ocioso, se houve, é setado a flag *Ready* e finaliza o *loop* interno, caso não, continua lendo o arquivo.

Ao sair do *loop* interno então há duas possibilidades, a primeiro é se houve o recebimento com o *MPI_Test* e então pode-se remover uma *task* da fila de tarefa e envia-lá a este trabalhador, a segunda possibilidade é de final de arquivo, nesta então é inserido o *MPI_Wait* e mantém aguardando algum trabalhador avisar que está ocioso e enviar a próxima tarefa.

Após, é finalizado o *loop* e se mantém apenas a finalização da fila de tarefas, sendo enquanto a fila não estiver vazia, é aguardado a comunicação de trabalhadores e ao receber remove da fila e envia a tarefa para processamento.

Ao finalizar, chega-se a parte final, sendo um *loop* de um até a quantidade de trabalhadores e para cada execução é aguardado novamente o aviso que está ocioso de qualquer trabalhador, e como já finalizou os serviços é enviado a este trabalhador uma *flag* padrão de zero para avisar a finalização dos processamentos. Está *flag* então faz que o trabalhador finaliza o *loop*, e prepara o envio dos seus resultados. No Mestre então é aguardado o recebimento de um vetor de tamanho quatro, com os resultados das variáveis agregadas (soma, quantidade de impares, valor mínimo e valor máximo), após então todos os trabalhadores enviar os resultados é processado pelo mestre e organizado os resultados finais, no qual após é mostrado na tela final para o usuário.

Na figura 1 é representado as comunicações comentadas entre o trabalhador e o mestre para melhor visualização.

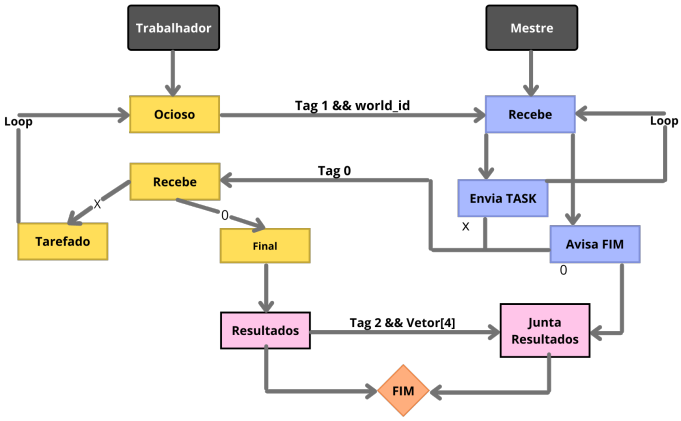
Para a compilação do trabalho:

1) Vá a raiz do projeto e execute:
2) *make* <tipo>;

Temos para o tipo:

- sequencial: compilação e execução do código sequencial;

Fig. 1: Representação MPI



- mpi: compilação e execução do código com *mpi*;
- all: compilação e execução de ambos códigos;
- clear: limpar os arquivos executáveis criado.

A seguir, padronizamos para passagem dos argumentos ao momento de execução, sendo importante ressaltar de para execução com *MPI* a passagem é a quantidade de processo e com isso o mestre entra na conta, ou seja, se utilizar *mpirun -n 4 ./executavel*, serão três trabalhadores e um mestre. Sabendo disto, temos:

- `-n`: Quantidade de processos, sendo a soma de um do mestre mais a quantidade de trabalhadores para o processamento;
- `-f`: caminho do arquivo com a lista de serviços a serem processados ou diretamente no código.

V. Metodologia

Fora criados cinco arquivos de entrada, com as tarefas para os testes, diversificando as ações e valores. Para estudo fora definido para o primeiro arquivo vinte linhas de tarefas, o segundo com quarenta linhas, o terceiro com oitenta, o quarto com cento e sessenta e o quinto com trezentos e vinte, ou seja, duplicando os valores.

Com estes arquivos foram analisado de duas maneira, tendo além da resposta correta quando comparadas com o do código sequencial e com *Threads*. A primeira maneira/quesito fora analisado o tempo de cada processo, utilizando a função *MPI_Wtime()*, no qual foi colocado no início de sua execução e antes do *MPI_finalize*, tendo então os seus tempos, para este quesito fora executado com quatro, seis e dez processos para todos os arquivos. Já para a segunda, são analisados o desempenho com o programa de análise de performance de atividades *TAU Commander* [5] e [6] sendo um poderoso programa para analisar os ambientes de execuções do *software*, tendo suporte para o *MPI*, no qual utilizamos prioritariamente focando nas métricas das comunicações, sendo então as quantidades de chamadas de *MPI_Irecv*, *MPI_Recv*, *MPI_Send*, *MPI_Test* e *MPI_Barrier* e também sinalizando o tempo para tais funções de *MPI*. Para este então fora selecionado o menor (um), o médio (três) e o maior (cinco) arquivo entradas, porém aplicado para maiores quantidade de trabalhadores,

sendo então quatro, sete e onze processos, com isso um para o mestre e o resto a quantidade de trabalhadores.

TABLE I: Configurações do notebook

S.O	Debian 10
Processador	AMD Ryzen 5-3500U CPU @ 2.1GHz x 4 - 64 bits
Memória RAM	8 GB's
Memória SSD	300 GB's
Placa de Vídeo	AMD Radeon Rx Vega 8

VI. Resultados

Iniciamos os resultados com a primeira análise, sendo então os tempos analisados com *MPI_Wtime()*.

Para o tempo executamos com os quatro primeiros arquivos de entradas, para quatro, seis e dez processos, sendo então um para o mestre e os outros para trabalhadores:

TABLE II: Resultados entrada 1

Execucao	4	6	10
Mestre	68.00s.	54.00s.	51.00s.
T1	68.00s.	54.00s.	50.00s.
T2	65.00s.	47.00s.	42.00s.
T3	64.00s.	51.00s.	51.00s.
T4	-	50.00s.	45.00s.
T5	-	49.00s.	42.00s.
T6	-	-	42.00s.
T7	-	-	42.00s.
T8	-	-	44.00s.
T9	-	-	47.00s.
Final	1m8,834s	0m54,840s	0m51,841s

TABLE III: Resultados entrada 2

Execucao	4	6	10
Mestre	80.00s.	51.00s.	42.00s.
T1	80.00s.	46.00s.	42.00s.
T2	77.00s.	49.00s.	26.00s.
T3	74.00s.	49.00s.	29.00s.
T4	-	47.00s.	30.00s.
T5	-	51.00s.	30.00s.
T6	-	-	32.00s.
T7	-	-	27.00s.
T8	-	-	27.00s.
T9	-	-	25.00s.
Final	1m20,833s	0m51,864s	0m42,888s

TABLE IV: Resultados entrada 3

Execucao	4	6	10
Mestre	165.00s.	112.00s.	80.00s.
T1	140.00s.	89.00s.	80.00s.
T2	139.00s.	88.00s.	52.00s.
T3	165.00s.	112.00s.	54.00s.
T4	-	85.00s.	56.00s.
T5	-	86.00s.	51.00s.
T6	-	-	67.00s.
T7	-	-	61.00s.
T8	-	-	54.00s.
T9	-	-	51.00s.
Final	2m45,845s	1m52,849s	1m20,856s

TABLE V: Resultados entrada 4

Execucao	4	6	10
Mestre	266.01s.	175.01s.	117.00s.
T1	266.01s.	175.01s.	98.00s.
T2	266.01s.	166.01s.	109.00s.
T3	266.01s.	167.01s.	99.01s.
T4	-	165.01s.	101.01s.
T5	-	166.01s.	99.01s.
T6	-	-	99.01s.
T7	-	-	101.01s.
T8	-	-	117.00s.
T9	-	-	99.01s.
Final	4m26,839s	2m55,847s	1m57,871s

TABLE VI: Resultados entrada 5

Execucao	4	6	10
Mestre	610.02s.	385.01s.	251.01s.
T1	608.02s.	385.01s.	228.01s.
T2	610.02s.	375.01s.	229.01s.
T3	603.02s.	371.01s.	243.01s.
T4	-	374.01s.	227.01s.
T5	-	374.01s.	251.01s.
T6	-	-	232.01s.
T7	-	-	228.01s.
T8	-	-	231.01s.
T9	-	-	232.01s.
Final	10m10,856s	6m25,867s	4m11,861s

É definida então a seguir os resultados primordiais do programa *Tau Commander*. Para três dos arquivos de entradas, o primeiro, terceiro e o cinto e para todos com cinco, sete e onze processos.

Fig. 2: Entrada 1 - 5 Processos

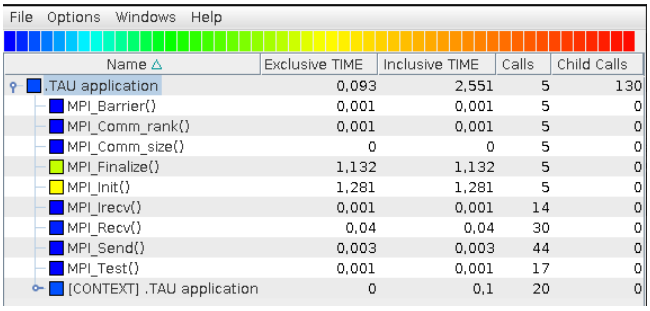
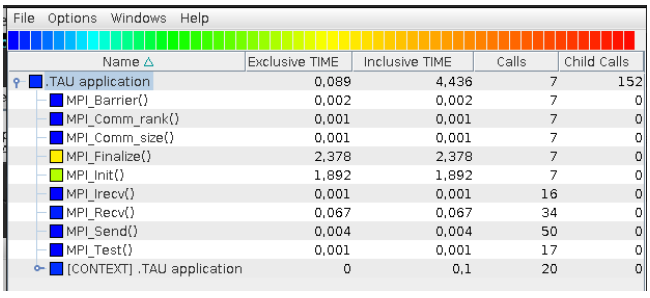


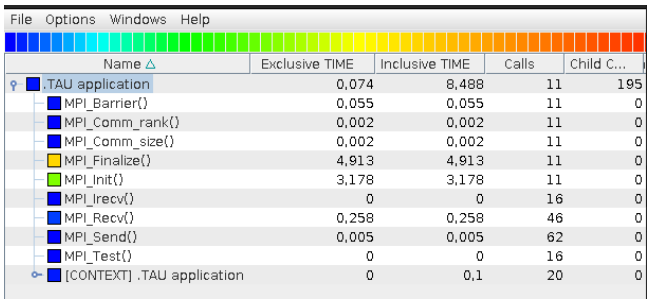
Fig. 3: Entrada 1 - 7 Processos



VII. Análise e Discussão

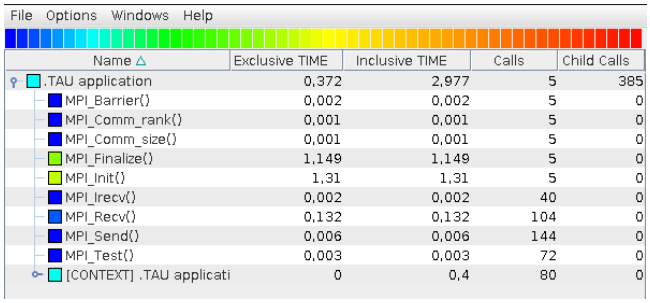
Para o primeiro quesito, ocorre como esperado no tempo de execução, quanto mais processos há na execução, menor o

Fig. 4: Entrada 1 - 11 Processos



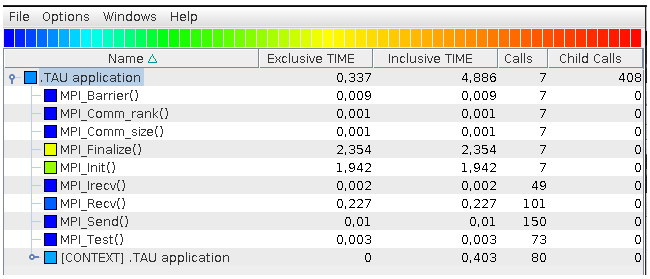
Name	Exclusive TIME	Inclusive TIME	Calls	Child C...
.TAU application	0,074	8,488	11	195
MPI_Barrier()	0,055	0,055	11	0
MPI_Comm_rank()	0,002	0,002	11	0
MPI_Comm_size()	0,002	0,002	11	0
MPI_Finalize()	4,913	4,913	11	0
MPI_Init()	3,178	3,178	11	0
MPI_Irecv()	0	0	16	0
MPI_Recv()	0,258	0,258	46	0
MPI_Send()	0,005	0,005	62	0
MPI_Test()	0	0	16	0
[CONTEXT] .TAU application	0	0,1	20	0

Fig. 5: Entrada 3 - 5 Processos



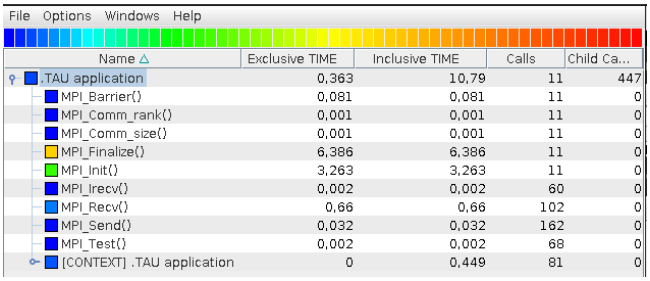
Name	Exclusive TIME	Inclusive TIME	Calls	Child Calls
.TAU application	0,372	2,977	5	385
MPI_Barrier()	0,002	0,002	5	0
MPI_Comm_rank()	0,001	0,001	5	0
MPI_Comm_size()	0,001	0,001	5	0
MPI_Finalize()	1,149	1,149	5	0
MPI_Init()	1,31	1,31	5	0
MPI_Irecv()	0,002	0,002	40	0
MPI_Recv()	0,132	0,132	104	0
MPI_Send()	0,006	0,006	144	0
MPI_Test()	0,003	0,003	72	0
[CONTEXT] .TAU applicati	0	0,4	80	0

Fig. 6: Entrada 3 - 7 Processos



Name	Exclusive TIME	Inclusive TIME	Calls	Child Calls
.TAU application	0,337	4,886	7	408
MPI_Barrier()	0,009	0,009	7	0
MPI_Comm_rank()	0,001	0,001	7	0
MPI_Comm_size()	0,001	0,001	7	0
MPI_Finalize()	2,354	2,354	7	0
MPI_Init()	1,942	1,942	7	0
MPI_Irecv()	0,002	0,002	49	0
MPI_Recv()	0,227	0,227	101	0
MPI_Send()	0,01	0,01	150	0
MPI_Test()	0,003	0,003	73	0
[CONTEXT] .TAU application	0	0,403	80	0

Fig. 7: Entrada 3 - 11 Processos



Name	Exclusive TIME	Inclusive TIME	Calls	Child Ca...
.TAU application	0,363	10,79	11	447
MPI_Barrier()	0,081	0,081	11	0
MPI_Comm_rank()	0,001	0,001	11	0
MPI_Comm_size()	0,001	0,001	11	0
MPI_Finalize()	6,386	6,386	11	0
MPI_Init()	3,263	3,263	11	0
MPI_Irecv()	0,002	0,002	60	0
MPI_Recv()	0,66	0,66	102	0
MPI_Send()	0,032	0,032	162	0
MPI_Test()	0,002	0,002	68	0
[CONTEXT] .TAU application	0	0,449	81	0

tempo para terminar o programa, isto ocorre pelo paralelismo destas tarefas, no qual aproveita o máximo de trabalhadores possíveis para o processamento e também o paralelismos para esperar as tarefas do mestre, tendo o tempo consequentemente diminuído na visão total. Isto fica mais claro no decorrer que se aumenta o arquivo de entrada, já para o tempo de cada processo, como a entrada de arquivo é variada os valores o equilíbrio não é possível ver apenas com estes resultados, porém é perceptível que todos estão trabalhando o máximo possível.

Já para a segunda análise com o uso da ferramenta *Tau*, podemos perceber a grande quantidade de chamadas de comunicação, sendo o *MPI_Send* a principal em ambas partes da comunição e com isso com o crescimento de trabalhadores e também de processos no arquivo de entrada a comunicação tende a crescer tendenciosamente, juntamente com as outras funções de comunicação, porém mais despertas, já que mantém o mesmo objetivo porém aplicando a comunicação não bloqueante, como é no caso do *MPI_Irecv* e *MPI_Recv*, no qual somando o *Calls* de ambos em cada análise pode-se perceber que é o mesmo de*MPI_Send*, pois a cada momento que é recebido uma comunicação é enviado também algum dado para seguir o fluxo em 1. Vale ressaltar também o uso do *MPI_Teste* no qual pode-se perceber que no decorrer que cresce o arquivo de entrada causa mais chamadas da função, isto ocorre pois o mesmo é utilizado como meio para saber se houve o envio da outra parte da comunicação e com isso saber se pode prosseguir, e então quando há diversos processos com tempos maiores causa excesso de trabalhador ocioso e com isso sem envio ao Mestre, tendo que o *MPI_Test* fique executando diversas vezes até os processamentos finalizar.

Conclusão

Finalizamos tendo que o projeto colaborou na experiência e elaboração de códigos com paralelismo, desenvolvendo melhor os conceito aprendidos em sala de aula. O código em si elaborado atingiu as expectativas e há o bom desempenho quando comparado com o sequencial, porém quando comparado com o tempo do uso de *Threads* percebe-se que há uma perca de performance, e quanto maior fica o arquivo de entrada maior causa diferença de tempo entre o uso do *MPI* e de *Threads*. Como ponto a melhorar, poderia melhorar a lógica para diminuir a troca de dados entre os processos, otimizando assim a sua performance. Para trabalhos futuros recomenda-se focar em uma análisa especifica para desenvolvimentos entre *MPI* e *Threads* analisando o limite máximo do arquivo de entrada para que a comparação similar, complementando também a diferença causada no melhor desempenho das *Threads* quando se cresce as fila de tarefas nas execuções. [1]

References

[1] L. Calvin; SNYDER, “Snyder: Parallel programming,” vol. 24, 2008.

[2] RookieHPC, “MPI documentation.” https://www.rookiehpc.com/mpi/docs/index.php, 2019 - 2021. Online; acessado em 6 Novembro 2021.

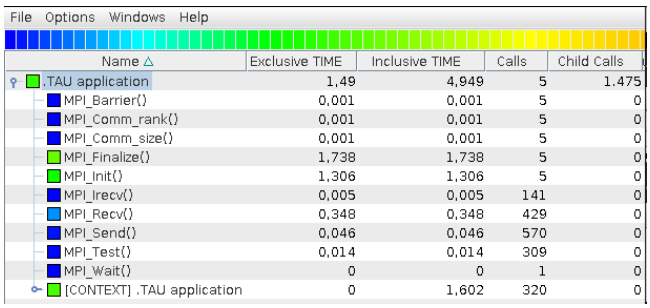
[3] W. Kendall, “A Comprehensive MPI Tutorial Resource.” https://mpitutorial.com/, 2021. Online; acessado em 6 Novembro 2021.

[4] B. Barney and L. L. N. Laboratory, “Message Passing Interface (MPI).” https://hpc-tutorials.llnl.gov/mpi/, 2021. Online; acessado em 6 Novembro 2021.

[5] D. M. ParaTools, “Tau commander: Introductory guide,” vol. 1.

[6] S. S. ParaTools, “Parallel performance optimization using tau training and workshop,” 2019.

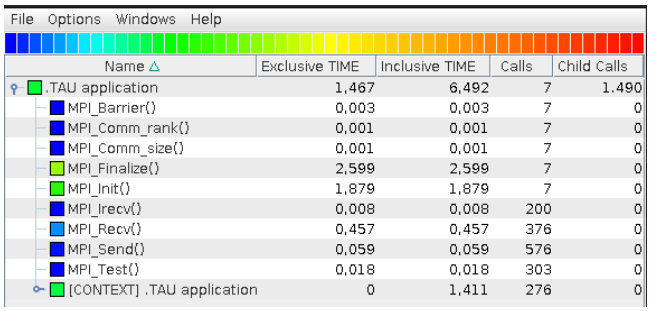
Fig. 8: Entrada 5 - 5 Processos



File Options Windows Help

Name △	Exclusive TIME	Inclusive TIME	Calls	Child Calls
.TAU application	1,49	4,949	5	1.475
MPI_Barrier()	0,001	0,001	5	0
MPI_Comm_rank()	0,001	0,001	5	0
MPI_Comm_size()	0,001	0,001	5	0
MPI_Finalize()	1,738	1,738	5	0
MPI_Init()	1,306	1,306	5	0
MPI_Irecv()	0,005	0,005	141	0
MPI_Recv()	0,348	0,348	429	0
MPI_Send()	0,046	0,046	570	0
MPI_Test()	0,014	0,014	309	0
MPI_Wait()	0	0	1	0
[CONTEXT] .TAU application	0	1,602	320	0

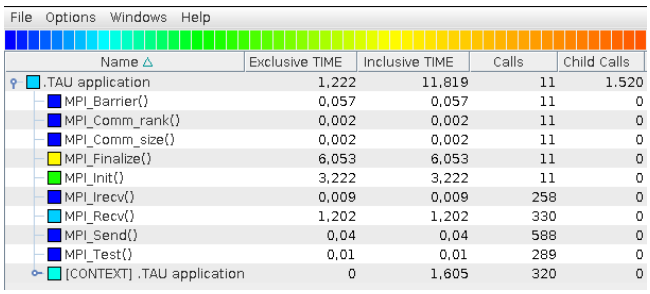
Fig. 9: Entrada 5 - 7 Processos



File Options Windows Help

Name △	Exclusive TIME	Inclusive TIME	Calls	Child Calls
.TAU application	1,467	6,492	7	1.490
MPI_Barrier()	0,003	0,003	7	0
MPI_Comm_rank()	0,001	0,001	7	0
MPI_Comm_size()	0,001	0,001	7	0
MPI_Finalize()	2,599	2,599	7	0
MPI_Init()	1,879	1,879	7	0
MPI_Irecv()	0,008	0,008	200	0
MPI_Recv()	0,457	0,457	376	0
MPI_Send()	0,059	0,059	576	0
MPI_Test()	0,018	0,018	303	0
[CONTEXT] .TAU application	0	1,411	276	0

Fig. 10: Entrada 5 - 11 Processos



File Options Windows Help

Name △	Exclusive TIME	Inclusive TIME	Calls	Child Calls
.TAU application	1,222	11,819	11	1.520
MPI_Barrier()	0,057	0,057	11	0
MPI_Comm_rank()	0,002	0,002	11	0
MPI_Comm_size()	0,002	0,002	11	0
MPI_Finalize()	6,053	6,053	11	0
MPI_Init()	3,222	3,222	11	0
MPI_Irecv()	0,009	0,009	258	0
MPI_Recv()	1,202	1,202	330	0
MPI_Send()	0,04	0,04	588	0
MPI_Test()	0,01	0,01	289	0
[CONTEXT] .TAU application	0	1,605	320	0