Universidade Federal São João del Rei
Curso de Ciência da Computação

Trabalho Prático I: Relatório

Computação Paralela

Discente: Julio Cesar da Silva Rodrigues

Docente: Rafael Sachetto Oliveira

Outubro
2023

Conteúdo

1 Introdução

Neste trabalho, é apresentada uma solução de paralelização em um problema de se encontrar a quantidade de divisores, dada uma entrada variável com quaisquer valores. O objetivo principal é identificar porções da implementação que por talvez apresentarem maior uso computacional, possam ser paralelizadas utilizando a biblioteca **OpenMPI**[1] com o paradigma de computação paralela mestre-escravo para obter ganhos significativos em tempo de execução.

Além disso, é realizada uma análise detalhada quanto a como cada tipo de entrada afeta no tempo de execução do programa, a granularidade (i.e., taxa entre computação e comunicação dos processos), uso de recursos computacionais (CPU, memória principal, ...) e o potencial *speed up* que o aumento da paralelização (i.e., número de processos) proporciona.

2 Desenvolvimento

Toda a concepção do cálculo do número de divisores, assim como as funções responsáveis pela paralelização e manipulação de E/S foram implementadas na linguagem C (excetuando a análise gráfica de resultados e geração dos instâncias de teste, estes que foram gerados utilizando a linguagem Python[2] com auxílio das bibliotecas Matplotlib[3] e seaborn[4]). A implementação foi desenvolvida com o auxílio da ferramenta de controle de versionamento GitHub[5], e encontra-se disponível publicamente no repositório `https://github.com/juliorodrigues07/parallel_primes`.

A organização do código se apresenta de forma bastante simples. Existem um par de arquivos de código fonte (juntamente com seus *headers .h* associados) que são responsáveis por realizar as manipulações de entrada e saída (*data_management*) e calcular o número de divisores de cada valor da entrada (*divisors*). No código fonte principal (*main.c*) encontra-se a utilização da biblioteca OpenMPI para introdução do paralelismo ao programa. Além disso, também existe um segundo arquivo (*sequential.c*) para execução sequencial para fins de comparação de resultados.

[1] Mais informações disponíveis em: `https://www.open-mpi.org/`

[2] Mais informações disponíveis em: `https://www.python.org/`

[3] Mais informações disponíveis em: `https://matplotlib.org/`

[4] Mais informações disponíveis em: `https://seaborn.pydata.org/`

[5] Mais informações disponíveis em: `https://github.com`

2.1 Cálculo do Número de Divisores

Para efetuar o cálculo do número de divisores, é testada a divisão exata (sem resto) de todos os valores da entrada até o divisor máximo possível do valor N, ou seja, $\frac{N}{2}$. Além disso, sabemos que dado um número qualquer N, se este não possuir qualquer divisor exato D tal que

$$1 < D \leq \sqrt{N}, \tag{1}$$

N é primo. Neste caso, podemos afirmar que o número N possui apenas dois divisores (1 e N), descartando a necessidade de testar até o limite superior de $\frac{N}{2}$ valores. Veremos mais adiante que tal abordagem trará ganhos significativos dependendo do tipo e tamanho da entrada.

Além disso, o incremento do laço é dependente do valor com o qual será calculado o número de divisores a cada chamada à função. Matematicamente, é possível provar que um número ímpar nunca é divisível por um número par, ou seja, lidando com um número ímpar podemos incrementar o contador em 2, reduzindo pela metade o número de iterações no pior caso $\left(\text{laço itera até o limite de } \frac{N}{2}\right)$. No entanto, para valores pares o incremento ainda é unitário.

2.2 Instâncias Geradas

Para testar a implementação de forma minimamente satisfatória, foram geradas três instâncias com base na teoria da complexidade correspondente ao melhor, pior e "médio" caso. Todas as instâncias possuem números sorteados pseudo aleatoriamente com no máximo 30 *bits*, isto para evitar *overflow* devido ao limite superior do tipo inteiro na linguagem C ($2^{31} - 1$). Por fim, o último aspecto em comum que todas possuem é a quantidade fixa de mil valores cada como entrada.

A instância correspondente ao melhor caso é composta somente de números primos, ou seja, estes serão submetidos sempre no máximo à divisão por um divisor D, cujo valor é $\sqrt{N}$. Como grande parte dos valores presentes neste arquivo de entrada possuem muitos *bits*, o número de iterações realizadas será consideravelmente menor $\left(\sqrt{N} << \frac{N}{2}\right)$.

Já a instância do melhor caso possuem somente números compostos, ou seja, o número de iterações no cálculo do número de divisores de um valor N será sempre o máximo: $\frac{N}{2}$. Por fim, como pode-se presumir, a instância do caso médio possui uma mistura de números primos e compostos (aproximadamente 50% de cada tipo).

2.3 Perfil de Desempenho Sequencial

Analisando a execução da solução para o problema sem quaisquer tipos de paralelismo introduzidos, é nítida a ociosidade dos recursos computacionais, principalmente por parte da CPU, utilizando apenas um núcleo para calcular o número de divisores de cada valor do arquivo de entrada.

Com auxílio da ferramenta de perfilamento **gprof**[6], é observado que em média todo o tempo de processamento é oriundo apenas da execução da função de cálculo do número de divisores de cada valor. Da forma como a implementação foi realizada, tal cálculo independente de quaisquer outros valores presentes no arquivo de entrada, ou seja, seria perfeitamente possível na teoria paralelizar por completo o processamento (problema embaraçosamente paralelo), com um núcleo ficando responsável pelo cálculo de cada valor.

No entanto, sabemos que tal abordagem é inviável na prática com entradas cujo tamanho ultrapassa as centenas ou milhares de valores. Logo, foi decidido por paralelizar tal execução conforme o poder computacional à disposição. Todos os testes foram realizados em um computador com processador Intel© Core™ i5-8265U CPU que possui 4 núcleos e frequência base 1,60 GHz.

2.4 Identificação das Oportunidades de Paralelização

Como a maior parte do esforço computacional se concentra na execução da função que calcula o número de divisores de cada número, foi decidido por aplicar uma decomposição do domínio na implementação. Em outras palavras, a entrada é fragmentada em um número determinado de porções, estas que serão direcionadas para cada processo para execução em paralelo.

Como estratégia dominante na paralelização, optei por utilizar somente duas fases de comunicação durante toda a execução. Antes de qualquer processamento, o processo mestre ($rank = 0$) realiza a leitura do arquivo de entrada por completo para a memória e calcula os vetores correspondentes aos parâmetros reais de *sendcount* e *displacements* conforme o tamanho da entrada e o número de processos em execução. Tais vetores são de extrema relevância para a distribuição das porções para processamento paralelo.

Por fim, a aplicação de poucas fases de comunicação entre processos durante a execução do programa implicará em uma granularidade grossa para o processamento. Em outras palavras, a taxa de computação em relação a taxa de comunicação será grande. Após todos estes processos uma barreira (*MPI_Barrier*) é inserida para sincronizar todos os processos, sanando

[6]Mais informações disponíveis em: `https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html`

alguma dependência de computação, para então iniciar-se a cronometrar o tempo genuíno de computação com paralelismo.

2.5 Paralelização

A rotina selecionada para enviar os fragmentos do arquivo de entrada para cada processo escravo foi a *MPI_Scatterv*. O motivo por trás de tal escolha é de que uma de suas alternativas (*MPI_Scatter*) possui a restrição de que o tamanho da entrada deve ser divisível pelo número de processos. No caso da rotina *MPI_Scatterv*, essa restrição é inexistente, bastando fornecer os vetores correspondentes ao tamanho de cada porção e deslocamento, estes que foram calculados anteriormente conforme o tamanho da entrada.

Após a distribuição dos dados, cada processo fica responsável por um fragmento da entrada, estes que serão operados em paralelo, calculando o número de divisores de cada valor sem que exista qualquer dependência entre tais que possa tornar o processo ocioso ou bloqueante.

Finalizada a computação paralelizada, os resultados do processamento de cada porção são agrupados utilizando a rotina complementar da *MPI_Scatterv*: a *MPI_Gatherv*. Novamente os vetores que correspondem ao tamanho de cada porção de cada processo e os deslocamentos são utilizados como parâmetros reais. Logo após é inserida uma nova barreira para sincronizar todos os processos, já que neste caso há grande probabilidade de que o tamanho dos fragmentos não sejam iguais entre si, encurtando ou atrasando o processamento. Por fim, a cronometragem é então finalizada para análise de resultados.

Concluídas todas estas etapas, o processo mestre então realiza a última operação de E/S, armazenando o número de divisores de cada valor do arquivo de entrada em sua respectiva ordem em um arquivo de saída.

3 Compilação e Execução

Para compilar e executar o programa sequencialmente sem ou com perfilamento, basta executar os respectivos comandos via terminal:

make sequential

make with_prof

Para compilar a versão paralelizada do programa, basta executar o seguinte comando via terminal:

make

Para executar a versão paralelizada do programa de forma compartilhada (localmente) ou distribuída (várias máquinas), basta executar os respectivos comandos via terminal:

make shared_run

make distributed_run

Os parâmetros de execução podem e devem ser alterados no arquivo *Makefile*, alterando o arquivo de teste, número de processos e o *hostfile*. Na compilação e execução, deve-se atentar para a explicitação do compilador MPI, utilizando opções distintas com ou sem caminhos relativos e absolutos como: {mpicc, mpirun, home/<user_name>/openmpi/bin/mpi<cc or run>, home/<user_name>/<user_name>/openmpi/bin/mpi<cc or run>}, sendo a última opção para uso em laboratório.

Também é possível gerar suas próprias instâncias de teste utilizando o *script Python* presente no diretório *instances*. Basta passar como parâmetros o tamanho máximo dos números em *bits* e a quantidade a ser gerada executando o seguinte comando como exemplo:

python3 instance_generator.py -l 30 -n 10000

4 Análise de Resultados

Como resultado mais relevante, podemos observar o comportamento completamente distinto do programa quanto ao tipo de entrada com a qual este está lidando. Não apenas sequencialmente, mas também em execução paralela, as instâncias que continham somente números primos foram as entradas com as quais o programa conseguiu executar com maior eficiência. Como mencionado na Subseção 2.2, todas as entradas possuem o mesmo tamanho, e os valores internos também possuem em média a mesma magnitude (até 30 *bits*).

A razão pela qual o programa é mais eficiente com instâncias que contém maior quantidade de números primos, é de que o número de iterações no cálculo do número de divisores é reduzido drasticamente. Primeiramente, todo número primo é ímpar, logo o número de iterações no pior caso já é reduzido pela metade $\left(\frac{N}{4}\right)$. Além disso, o teste de primalidade reduz ainda mais a quantidade de iterações, ou seja, as divisões são testadas até o limite de

$\sqrt{N}$, com incremento de 2 no divisor a cada iteração. Como grande parte dos números presentes na entrada possuem grande quantidade de *bits*, o número de iterações S é reduzido, tal que $\sqrt{S} << \frac{N}{2}$.

Na Figura 1, podemos observar que o tempo de execução para instâncias com mil valores é completamente discrepante entre os tipos de entrada. Entradas contendo somente números primos são executadas em milissegundos, enquanto entradas contendo somente números compostos são executadas em minutos. Como é de se presumir, entradas que possuem uma mistura de tipos de números (primos e compostos) possuem como tempo de execução um "ponto médio" entre os dois extremos (melhor caso e pior caso).

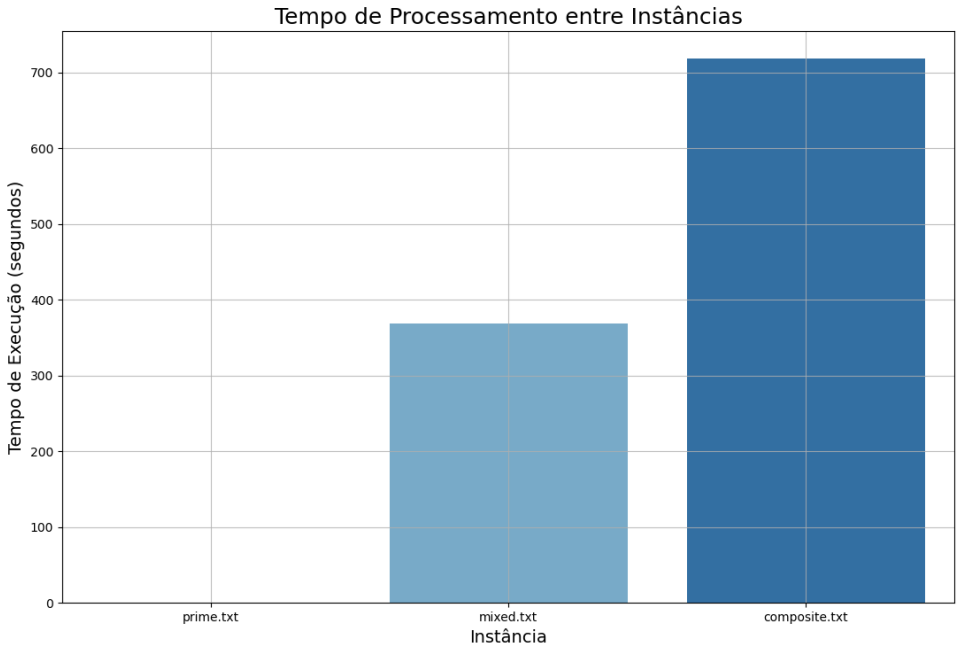

Figura 1: Tempo de execução sequencial

Como o tempo de execução para instâncias com 1000 valores contendo somente números primos é baixo, não há grandes variações na mensuração do mesmo variando o número de processos para execução em paralelo. Como mostra a Figura 2, embora ainda seja possível alcançar um *speed up* de aproximadamente 2x em relação à execução sequencial, aumentar o número de processos poderá eventualmente provocar piora no tempo de execução devido ao aumento da comunicação entre processo mestre e escravos.

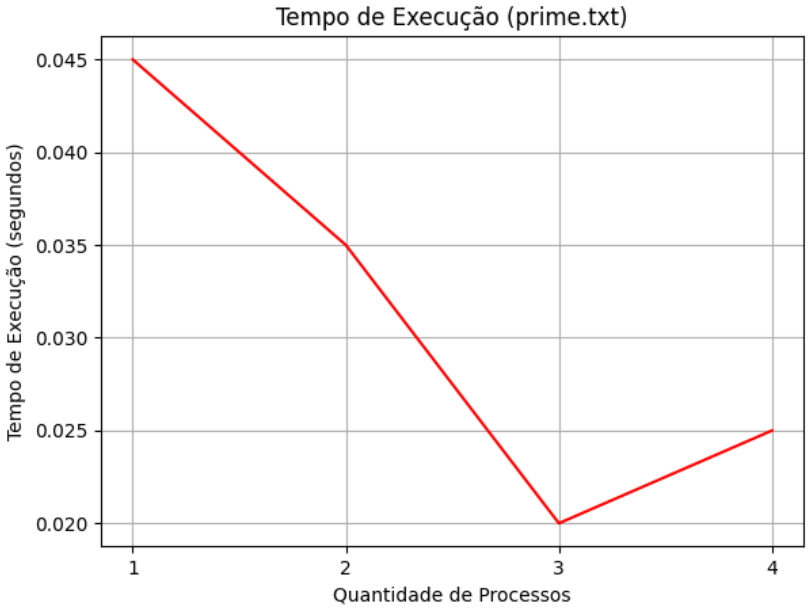

Figura 2: Tempos de execução com números primos

Para execução das instâncias contendo somente números compostos e a mistura de ambos, o aumento no número de processos certamente promoveu ganhos no tempo de execução dos programas. Como mostrado na Figura 3 e 4, o aumento gradual do número de processos provocou um *speed up* de aproximadamente 2,4x e 2,6x em relação à execução sequencial para as instâncias composta e mista, respectivamente. No entanto, é nítido que os ganhos em tempo de execução não são inversamente proporcionais ao aumento do número de processos (e.g., 4 processos, tempo 4x menor). Tal observação pode ser explicada novamente pelo aumento na comunicação entre processos.

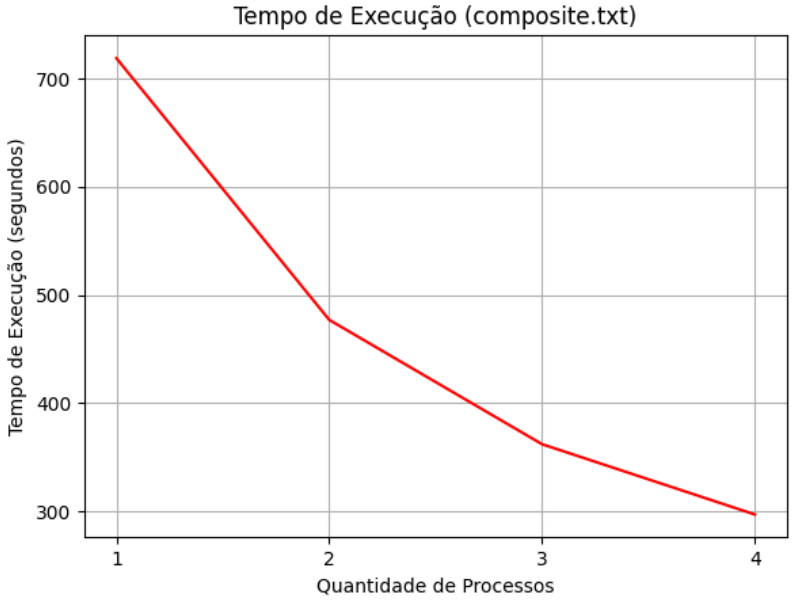

Figura 3: Tempos de execução com números compostos

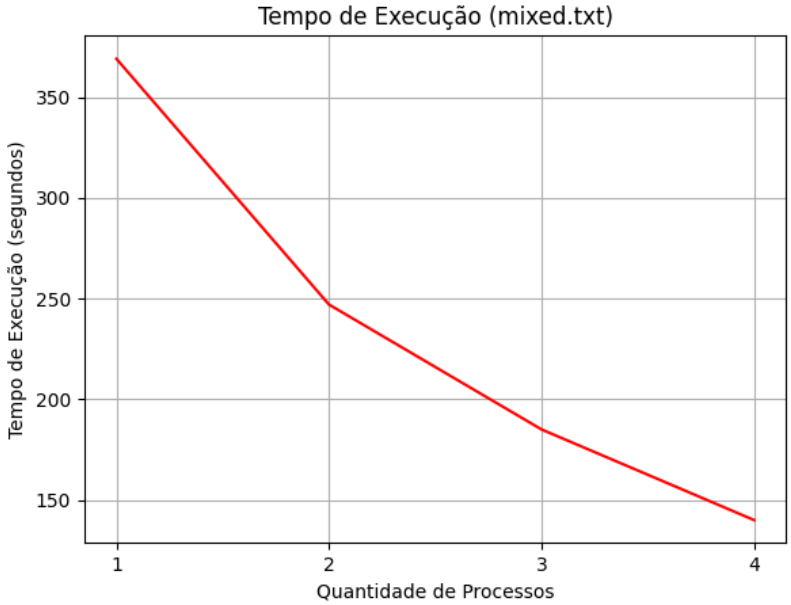

Figura 4: Tempos de execução com números primos e compostos

Por fim, podemos observar na Figura 5 o quadro geral das execuções sequenciais e em paralelo, com diferentes números de processos e instâncias de tipos distintos. A paralelização do programa certamente trouxe ganhos em tempo de execução para execução com entradas mistas e compostas. No entanto, não é possível obter conclusões significativas sobre o ganho com entradas primas. Para mensurar a eficiência do programa com este tipo de entrada, é necessário gerar instâncias significativamente maiores para experimentação.

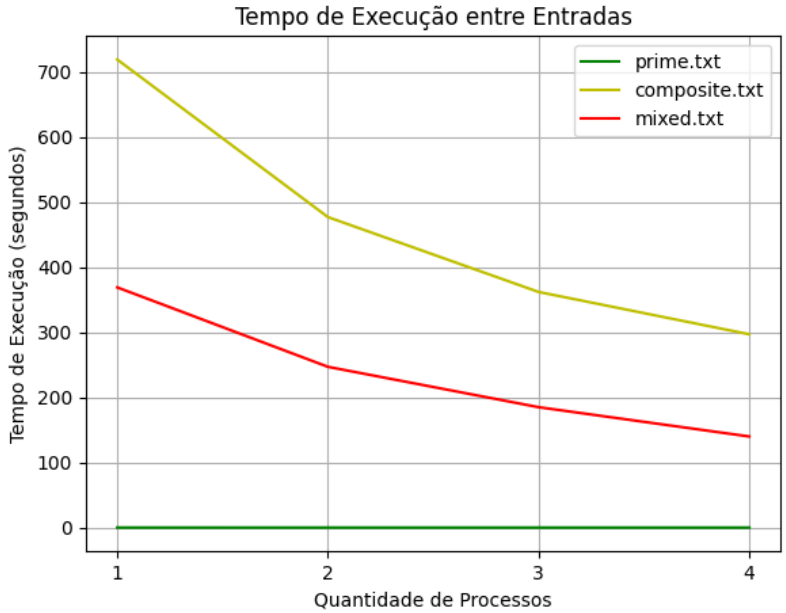

Figura 5: Tempos de execução variando número de processos e entradas

5 Limitações

Como principal limitação, é possível citar o descumprimento parcial do modelo mestre-escravo. Na utilização da rotina *MPI_Scatterv*, os fragmentos do arquivo de entrada são particionados de forma que todos os processos envolvidos no comunicador recebem um fragmento (Figura 6). Em outras palavras, o processo mestre que deveria apenas ser responsável pelas operações de E/S (carregando o arquivo de entrada para a memória e armazenando o resultado em um arquivo de saída), também realiza o processamento do número de divisores de uma porção dos dados de entrada.

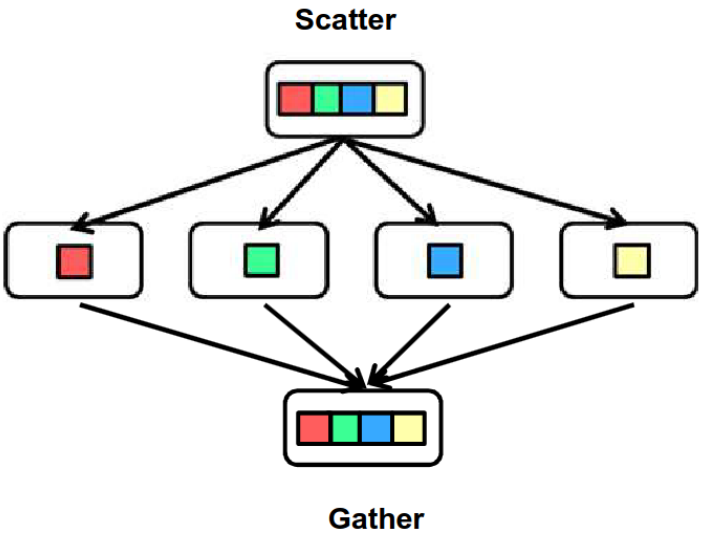

Figura 6: Princípio de funcionamento das rotinas *Scatter*

Fonte: Mike Bailey, Oregon State University[7]

Para contornar tal obstáculo, é possível criar um comunicador restrito aos processos escravos, para que então possa ser realizada a operação de *Scatter* com o primeiro processo escravo como raiz ($rank = 1$), distribuindo os dados somente entre os processos escravos. Posteriormente o agrupamento dos dados seria realizado no comunicador global, para que então o processo mestre possa armazenar o resultado em um arquivo de saída.

Outro obstáculo encontrado no desenvolvimento deste trabalho foi a execução em ambientes distribuídos. Surgiram diversas adversidades quanto às tentativas de conexões via SSH[8] com as máquinas presentes nos laboratório do DComp. Por isto, não foi possível coletar resultados sobre o tempo de execução de processos em máquinas distintas, o possível atraso que a comunicação por rede poderia acarretar na execução do programa, a utilização dos recursos computacionais como memória principal, e o eventual *speed up* que tal forma de execução poderia oferecer.

[7] Disponível em: `https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/mpi.2pp.pdf`

[8] Mais informações disponíveis em: `https://www.openssh.com/manual.html`

6 Conclusão

Neste trabalho, foi explorada uma abordagem de computação paralela utilizando a biblioteca OpenMPI para um problema bastante simples, mas no qual foi possível explorar diversos conceitos como decomposição de domínio, granularidade e paradigmas como mestre-escravo e SMPD (Single Program Multiple Data).

Com este exemplo didático, podemos observar de forma nítida, a complexidade por trás de construir um programa que será executado em todos os processos (núcleos e/ou máquinas) de forma indiscriminada. Por isto, sempre é necessário ter extrema cautela no controle dos *ranks* de cada processo, assim como a alocação e o acesso a memória realizado por cada um, principalmente em um ambiente de computação com memória distribuída.

Referências

- The Message Passing Interface (MPI) - Parallelism on Distributed CPUs: `https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/mpi.2pp.pdf`;
- Profiling Programs With prof, gprof, and tcov: `https://docs.oracle.com/cd/E19059-01/stud.10/819-0493/OtherTools.html`;
- scatter variable length data: `https://stackoverflow.com/questions/43642561/scatter-variable-length-data`;
- Executing Shell Commands from a C program: `https://www.cs.uleth.ca/~holzmann/C/system/shell_commands.html`;
- Material disponível no SIGAA na disciplina de Computação Paralela.