ENGENHARIA AEROESPACIAL E MECÂNICA
MECATRÔNICA
CE-265 PROCESSAMENTO PARALELO

Docente: Jairo Panetta
Discente: Lucas Kriesel Sperotto

Exercício 3 – RELATÓRIO

Paralelização por MPI do "Cálculo de π" no Sistema "NEWTON"

Introdução:

Este relatório esta dividido em três partes distintas: procedimento, resultados e anexos.

No tópico procedimento demostrarei o "passo a passo" para a realização do trabalho, bem como o raciocínio que me levou a deduzir a maneira com o qual paralelizei o código, por fim, serão apresentados os resultados de tempo e desempenho com sua interpretação.

O anexo não se encontra no corpo do texto, mas sim serão anexados arquivos separados no "tidia". Nos arquivos anexados se encontrarão o código fonte paralelizado e os arquivos de retorno dos resultados.

Procedimento:

Nestes testes foram utilizados os softwares "PUTTY" para a conexão "ssh" e o software "WinSCP" para a conexão "sft", ambos em ambiente Windows XP.

Como proposto no trabalho, os testes deveriam ser efetuados no sistema não utilizado para realização do exercício anterior. Durante o processo de programação, preferi testar a paralelização no "crow" (pois eu já estava familiarizado) e depois reexecutar no "cesup".

Esta paralelização foi mais trabalhosa que a anterior, a primeira coisa que fiz foi alterar os argumentos da função "inCircle()", adicionando-lhe os parâmetros para que sua execução fosse sobre uma determinada parcela do domínio (necessário para dividir o domínio entre os "n" processos). Tentei paralelizar apenas a chamada do método, dividindo suas entradas antes da chamada (baseando meu pensamento no OpenMP), mas depois percebi que o código todo executava "n" vezes, então me obriguei a estudar mais e mudar a estratégia.

Confesso que efetuei várias tentativas, até que consegui abstrair como a execução ocorria, onde defini que um processo iria atuar como mestre, e os outros seus escravos. O mestre se encarregaria de calcular as divisões do "range", números de pontos e claro a posição inicial em "x" que cada escravo iria utilizar. Na separação das tarefas o mestre deve enviar por mensagem a cada escravo, os parâmetros calculados e necessários na chamada da função "inCricle()", obviamente o mestre deve receber por mensagem os resultados calculados pelos escravos, juntar essas informações e calcular o resultado final.

Para um melhor entendimento separei as variáveis em grupos conforme sua utilização e criação no código, foram elas: as variáveis criadas para a paralelização, as variáveis utilizadas apenas pelo mestre e as variáveis utilizadas tanto pelo mestre como pelos escravos. Isso me possibilitou observar algumas dependências e possíveis conflitos.

Depois disso criei condições para que somente o mestre executasse a parte da inicialização e finalização do software (colocando um "if" para identificar o mestre). Dessa forma caberia aos escravos apenas a parte de execução (chamada da função "inCircle()"), claro que para um melhor desempenho, bem como proporcionar a execução com um só processo, o mestre também deve executar a chamada da função ("inCircle()"), além disso ele não ficaria "ocioso" enquanto espera pela execução dos escravos. Cabe ressaltar que em alguns casos costuma-se não encarregar o mestre da execução, mas isso depende muito da aplicação e do tamanho tanto do problema como do cluster disponível.

Descrevendo os passos da execução, o mestre calcula o "range", o "x" inicial e o numero de pontos para cada processo, levando em consideração que ele também irá executar a chamada da função "inCircle()". Para possibilitar a execução o mestre envia três mensagens para paca processo, cada uma contendo uma das variáveis calculadas por ele. Seguindo após a execução da função "inCircle()", cada escravo envia uma mensagem ao mestre com o retorno da função, o mestre recebe esta mensagem e soma seus valores com o valor calculado por ele.

No final o mestre se encarrega de calcular o π, seu erro, o tempo da execução e escrever o resultado no arquivo.

Como comentado anteriormente, durante o processo de programação, executei os testes no "crow", sistema que eu estava mais familiarizado. Houve várias tentativas por erros de sintaxe etc.. Uma ferramenta que me utilizei para conseguir tanto entender o processo como para verificar o que estava acontecendo foi escrever no arquivo as trocas de mensagens entre os processos, quando funcionou acabei não retirando os comandos "printf" (aparecerá no arquivo entregue com os resultados mensagens tipo "MESTRE MANDA MENSAGEM 1 PARA ESCRAVO 2").

Após ter compilado e mostrado resultado satisfatório, compilei e executei no "cesup", uma diferença é que no "cesup" não é aceito criação de variáveis dentro da chamada de um laço "for", comentário este que não tem muita relação com a proposta do trabalho, mas enfim, a principal diferença que notei com as sucessivas execuções do mesmo número de processo e em ambos os sistemas, foi justamente à variação do tempo de execução do "cesup", que creio eu causar um erro muito grande no cálculo do Speed-Up. Por isso demostrei os dois resultados obtidos. Comentarei mais sobre isso em momento oportuno.

Tabelas de Resultados

Na tabela 1 está descrito o erro do cálculo, o tempo de execução o Speed-Up e a eficiência da paralelização em função do número de processos executado para o sistema "CROW". E na tabela 2 temos os resultados para a mesma aplicação no sistema do "CESUP". Lembrando que todos os testes foram feitos gerando-se 2^{30} pontos aleatórios.

Tabela 1 – Resultados no "CROW"

Nº de Processos	% Erro de π	Tempo de Execução (s)	Speed-Up	Eficiência
1	0,0001%	44,862545	**1**	100%
2	0,0006%	22,430498	**2,00**	100%
3	0,0017%	15,646624	**2,87**	95,57%
4	0,0019%	12,046783	**3,72**	93,10%
5	0,0018%	9,759162	**4,60**	91,94%
6	0,0017%	8,168634	**5,49**	91,53%
7	0,0011%	7,000905	**6,41**	91,54%
8	0,0014%	6,117158	**7,33**	91,67%

Tabela 2 – Resultados no "CESUP"

Nº de Processos	% Erro de π	Tempo de Execução (s)	Speed-Up	Eficiência
1	0,0001%	77,383224	**1**	100%
2	0,0006%	67,990471	**1,14**	56,91%
3	0,0017%	28,929062	**2,67**	89,16%
4	0,0019%	20,249968	**3,82**	95,53%
5	0,0018%	21,545797	**3,59**	71,83%
6	0,0017%	14,996552	**5,16**	86,00%
7	0,0011%	15,300364	**5,06**	72,25%
8	0,0014%	13,531890	**5,72**	71,48%

Comparando-se os gráficos, podemos observar os diferentes padrões de comportamendo do Speed-Up e consequentemente da eficiência. Para melhor ilustrar e visualizar esbocei alguns gráficos que estão apresentados a seguir.

Efetuei basicamente cinco baterias de testes, mas a primeira foi descartada por conter um erro (já comentado anteriormente) que não possibilitou a execução no "cesup". Assim alterei o código e refiz os testes em ambos os sistemas. Considerei então os resultados da segunda bateria até a quinta.

O gráfico 1 mostra os tempos de execução para a segunda bateria, comparando os tempos do "cesup" com os tempos da "crow". No gráfico 2 temos os Speed-Up calculados, no gráfico 3 as respectivas eficiências. Já nos gráficos 5 e 6 temos os Speed-Up para as sequencias de testes no mesmo sistema. E por fim no gráfico 4 os erros calculados de π, que não variou de um sistema para o outro, só teve variação para o numero de processos.

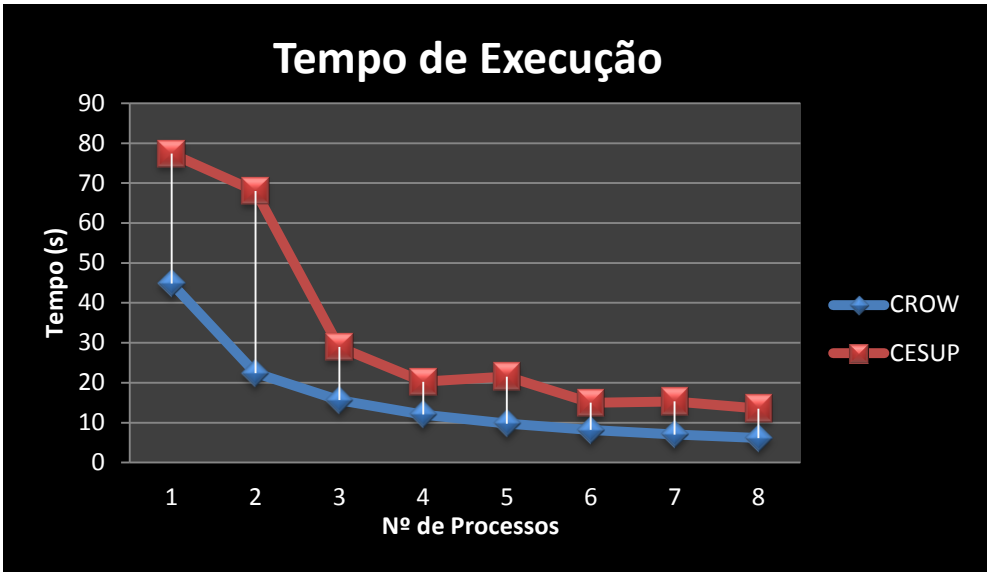

Gráfico 1 – Tempos de execução "CROW" X "CESUP"

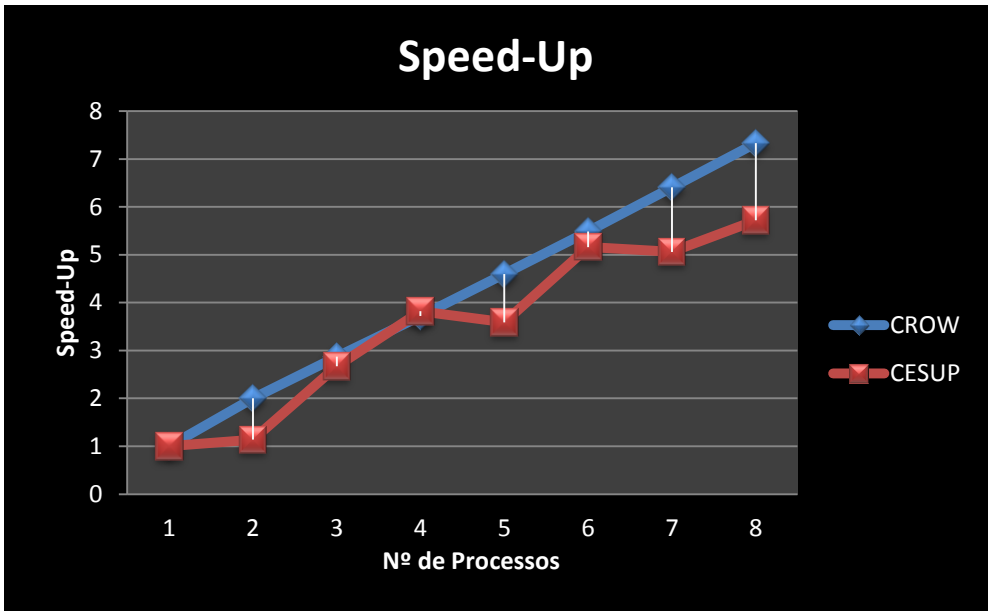

Gráfico 2 – Speed-Up "CROW" X "CESUP"

Olhando para esses dois primeiros gráficos, vemos a continuidade do tempo e Speed-Up do "crow", e claro a descontinuidade dos resultados obtidos a partir do "cesup". Desses comportamentos, podemos comprovar os resultados das eficiências no gráfico 3. Temos um padrão de comportamento das variáveis mostradas até agora para o sistema "crow", mas no "cesup" isso não foi conseguido, os valores são instáveis (não seguem um padrão). Acredito que isto se deve a propriedades de hardware da maquina, e claro ao seu uso. Para explicar melhor, monitorei todas as vezes as filas dos dois sistemas sempre que eu iniciava uma bateria de testes, no "crow" a fila estava sempre vazia, no entanto, o "cesup" possuía grande demanda de uso.

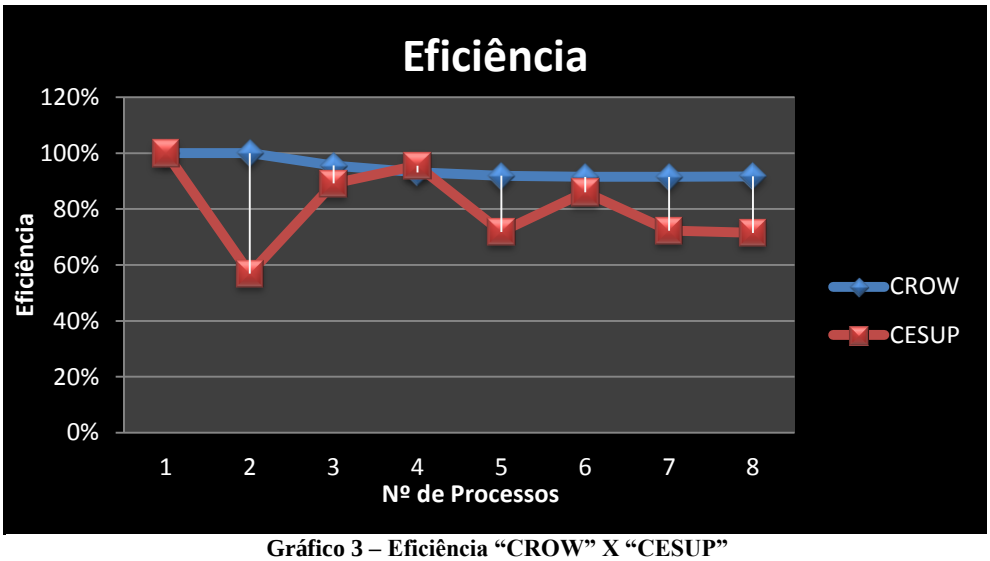

Gráfico 3 – Eficiência "CROW" X "CESUP"

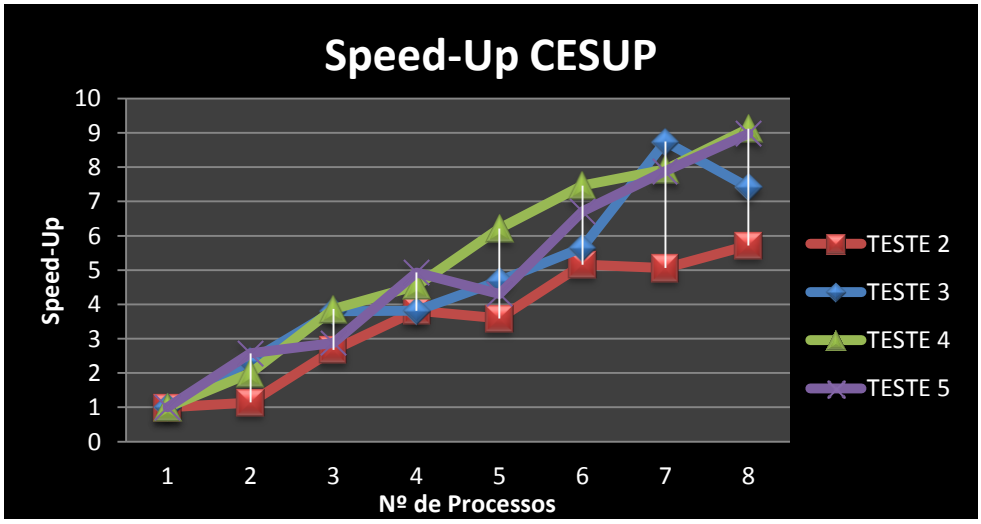

Gráfico 4 – Comparação dos Speed-Up em baterias de teste no "CESUP"

Com o gráfico 4 e 5, espero mostrar a impossibilidade de se obter confiabilidade no Speed-Up calculado, claro que isso poderia ser corrigido efetuando-se uma sequencia de testes para o mesmo número de processos e posteriormente calcular uma média sobre esses valores, mas eu optei por não realizar isso justamente por desconhecer os critérios que tornariam essa média aceitável.

Finalmente chegamos à parte mais divertida do trabalho, o comportamento do erro para os cálculos do número π. Efetuei algumas pesquisas e o numero pseudoaleatório gerado pela função "rand()" é determinístico, ou seja, uma operação matemática o define de acordo com seu anterior, a biblioteca usa uma "semente" como primeiro valor da série, como no caso não é especificada, o padrão adotado pela biblioteca é zero. Dessa forma cada processo gera a mesma sequencia de numero pseudoaleatório.

Uma maneira usual de ter números aleatórios mais "confiáveis" é utilizar o relógio da maquina como semente, ou então ferramentas de números aleatórios reais baseados em decaimento radioativo (HOT BITS - http://www.fourmilab.ch/hotbits/) ou do ruído atmosférico (http://www.random.org).

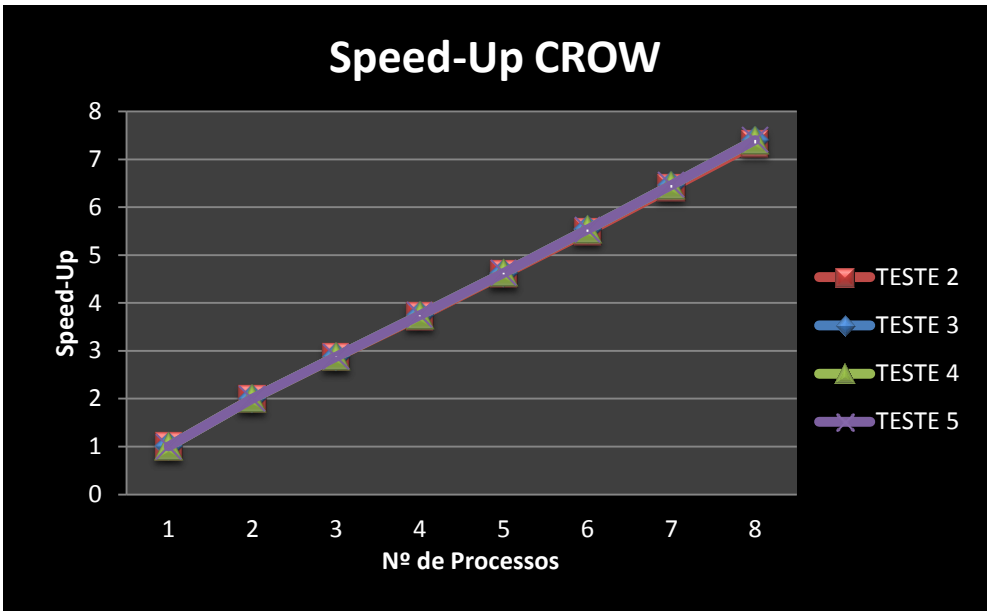

Gráfico 5 – Comparação dos Speed-Up em baterias de teste no "CROW"

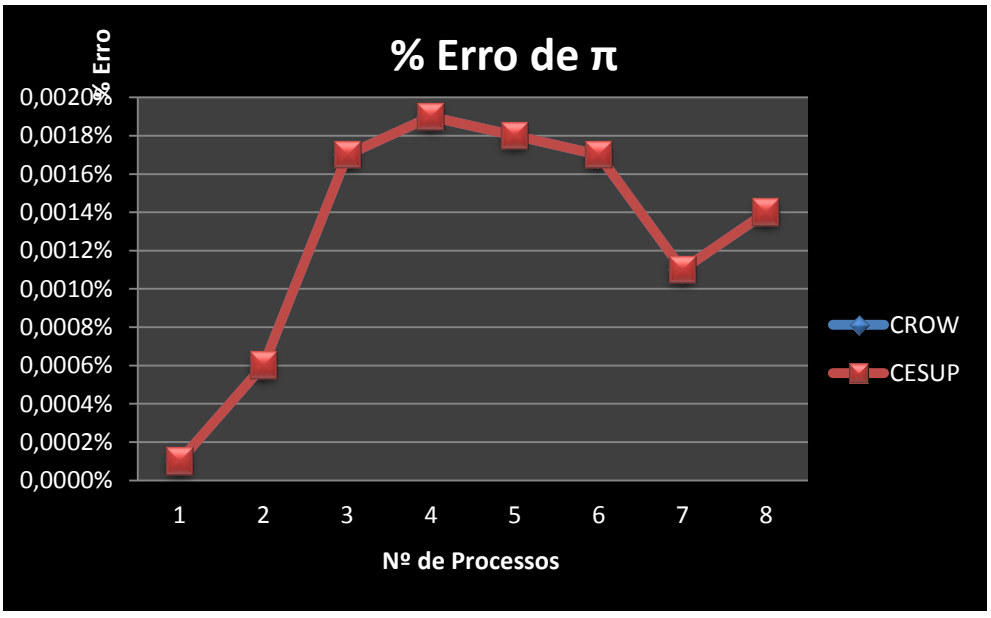

Gráfico 6 - % Erro de π no "CROW" e no "CESUP"

Como cada processo gera a mesma sequencia de números pseudoaleatórios, temos um valor limite superior par aos números randômicos gerados pela biblioteca. De forma que se inferior ao número total de pontos, causa um comportamento repetitivo de distribuição. Então temos um domínio com pontos distribuídos de forma

Para cada número de processos, teremos a mesma distribuição de pontos sobre pedaço de domínio de cada processo, dessa forma, a distribuição de pontos sobre o domínio muda seguindo um padrão de distribuição, sendo assim, o número de pontos para cada processo dentro e fora do círculo é diferente, alterando o resultado.