**Programação Concorrente e Distribuida**
**Aluno: Igor Macedo Silva**

**Questões feitas:**
**Cap.1. 1-7**
**Cap.2. 1-3, 5, 7, 10, 15-24**
**Total: 23**

Capítulo 1: 1-7.
Capítulo 2: 1-3, 5, 7, 10, 15-24
Instruções para resolução:
• Procure responder corretamente todas as questões da lista.
• Suas respostas serão validadas de forma oral por amostragem - geralmente de 2 à 3 defesas orais.
• Se não conseguir responder alguma questão, procure esclarecer as dúvidas em tempo em sala de aula com o professor, pelo SIGAA, com um colega, ou por e-mail. Se necessário, é possível marcar um horário para tirar dúvidas na sala do professor.
• Não serão aceitas respostas "mágicas", ou seja, quando a resposta está na lista entregue mas você não sabe explicar como chegou a ela. Sua nota nesse caso será 0 (zero). Mesmo que não saiba explicar apenas parte da sua resposta.
• Procure entregar a resolução da lista de forma organizada. Isso pode favorecer a sua nota.
• Os códigos dos programas requisitados (ou as partes relevantes) deverão aparecer no corpo da resolução da questão.
• A resolução da lista deverá ser entregue em formato PDF em apenas 1 (um) arquivo.
• O envio da resolução pode ser feito inúmeras vezes. Utilize-se disso para manter sempre uma versão atualizada das suas respostas e evite problemas com o envio próximo ao prazo de submissão devido a instabilidades no SIGAA.
• A lista com o número das questões respondidas deve aparecer na primeira folha da lista. Não será aceita alteração nessa lista.
• Procure preparar sua defesa oral para cada questão. Explicações diretas e sem arrodeios favorecerão a sua nota.
• A defesa deverá ser agendada com antecedência. Para isso, indique por email (samuel@dca.ufrn.br) no mínimo 3 horários dentro dos intervalos disponíveis em pelo menos 3 turnos diferentes. Caso não tenha disponibilidade em 3 turnos diferentes, deverá apresentar uma justificativa.
• Os horários disponíveis serão disponibilizados em uma notícia na turma virtual e serão atualizados a medida que os agendamentos forem sendo fixados.
• A defesa oral leva apenas de 10 a 15 minutos em horários fixados com antecedência. Não será tolerado que o aluno chegue atrasado para a sua prova.

**1.1 Devise formulas for the functions that calculate my_first_i and my_last_i in the global sum example. Remember that each core should be assigned roughly the same number of elements of computations in the loop. Hint: First consider the case when n is evenly divisible by p.**

Onde c é o número do core atual

my_first_i = $\left(\frac{n}{p}+(integer)(n\%p>c)\right)\cdot c+(n\%p)\cdot(!((integer)(n\%p>c)))$

my_last_i = $\left(\frac{n}{p}+(integer)(n\%p>c)\right)\cdot(c+1)+(n\%p)\cdot(!((integer)(n\%p>c)))$

algoritmo de teste em python

```
p = 3
n = 22
for c in range(p):
    plus = int(n%p > c)
    print str(plus)+' ['+str(((n/p+plus)*c +(n%p)*int(not(plus)))),
    print ', '+str((n/p + plus)*(c + 1) +(n%p)*int(not(plus)))+']'
```

**1.2 We've implicitly assumed that each call to Compute next value requires roughly the same amount of work as the other calls. How would you change your answer to the preceding question if call i = k requires k + 1 times as much work as the call with i = 0? So if the first call (i = 0) requires 2 milliseconds, the second call (i = 1) requires 4, the third (i = 2) requires 6, and so on.**

Se cada camada de i = k aumenta linearmente em relação à chamada de i = 0, podemos dividir a carga entre as camadas, considerando que a chamada de i = 0 é equivalente a uma carga de trabalho. Assim, pela fórmula da série da soma de 1 até n, S = n*(n+1)/2, podemos descobrir a carga total que deve ser dividida entre o número do cores. Logo se temos P cores, W = S/P será a carga que cada core deve consumir. Logicamente esse número nem sempre será inteiro, e também não temos garantia que um conjunto qualquer de chamadas de i = k irá acumular em W unidades de trabalho. A saída seria descobrir que chamadas de i = k imediatamente passam de W unidades de trabalho, quando somadas. Para isso, podemos ir somando as unidades de trabalho Wi de cada iteração e ir executando essa iteração, até chegar em um momento que a soma de Wi + Wi_1 + Wi_2 seja imediatamente superior a W, e marcadas como executadas em um array compartilhado, de acesso controlado por um lock.

**1.3 Try to write pseudo-code for the tree-structured global sum illustrated in Figure 1.1. Assume the number of cores is a power of two (1, 2, 4, 8, . . . ). Hints: Use a variable divisor to determine whether a core should send its sum or receive and add. The divisor should start with the value 2 and be doubled after each iteration. Also use a variable core difference to determine which core should be partnered with the current core. It should start with the value 1 and also be doubled after each iteration. For example, in the first iteration 0 % divisor = 0 and 1 % divisor = 1 , so 0 receives and adds, while 1 sends. Also in the first iteration 0 + core difference = 1 and 1 − core difference = 0 , so 0 and 1 are paired in the first iteration.**

*Seja ^ a operação de potência

```
For(int i = 0; i < log2(num_cores); i++)
        if (meu_id%2^(i+1) == 0)
                soma_outro = receba soma de core (meu_id+2^i)
                minha_soma += soma_outro
        else
                mande soma para core (meu_id – 2^i)
                mate processo
```

**1.4 As an alternative to the approach outlined in the preceding problem, we can use C's bitwise operators to implement the tree-structured global sum. In order to see how this works, it helps to write down the binary (base 2) representation of each of the core ranks, and note the pairings during each stage:**

**From the table we see that during the first stage each core is paired with the core whose rank differs in the rightmost or first bit. During the second stage cores that continue are paired with the core whose rank differs in the second bit, and during the third stage cores are paired with the core whose rank differs in the third bit. Thus, if we have a binary value bitmask that is 001 2 for the first stage, 010 2 for the second, and 100 2 for the third, we can get the rank of the core we're paired with by "inverting" the bit in our rank that is nonzero in bitmask . This can be done using the bitwise exclusive or ∧ operator. Implement this algorithm in pseudo-code using the bitwise exclusive or and the left-shift operator.**

*Seja ^ a operação de ou exclusivo

```
mask = 0b00000001
For(int i = 0; i < log2(num_cores); i++)
        if (!(meu_id^mask)^mask)
                soma_outro = receba soma de core (meu_id^mask)
                minha_soma += soma_outro
        else
                mande soma para core (meu_id ^mask)
                mate processo
```

**1.5 What happens if your pseudo-code in Exercise 1.3 or Exercise 1.4 is run when the number of cores is not a power of two (e.g., 3, 5, 6, 7)? Can you modify the pseudo-code so that it will work correctly regardless of the number of cores?**

Se rodar com um número que não é potencia de 2, algumas threads ficariam presas, esperando resposta de um core que não existe, ou a própria função retornaria um erro.

Uma solução elegante seria conferir se o número do core do qual a resposta deve vir existe, caso negativo a variável soma_outro deve receber zero.

**1.6 Derive formulas for the number of receives and additions that core 0 carries out using a. the original pseudo-code for a global sum, and b. the tree-structured global sum. Make a table showing the numbers of receives and additions carried out by core 0 when the two sums are used with 2, 4, 8, . . . , 1024 cores.**

a) soma original → n = num_cores-1

b) árvore de soma → n = ceiling(log2(num_cores)) // para casos diferentes de potencias de 2

| | original sum | tree sum |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 1 | 1 |
| 4 | 3 | 2 |
| 8 | 7 | 3 |
| 16 | 15 | 4 |
| 32 | 31 | 5 |
| 64 | 63 | 6 |
| 128 | 127 | 7 |
| 256 | 255 | 8 |
| 512 | 511 | 9 |
| 1024 | 1023 | 10 |

**1.7 The first part of the global sum example—when each core adds its assigned computed values—is usually considered to be an example of data-parallelism, while the second part of the first global sum—when the cores send their partial sums to the master core, which adds them—could be considered to be an example of task-parallelism. What about the second part of the second global sum—when the cores use a tree structure to add their partial sums? Is this an example of data- or task-parallelism? Why?**

A forma de pensar no mento de se estruturar a divisão dos cores explicita que o paralelismo utilizado pela arvore de soma é paralelismo de tarefas. Para esclarecer, não é preciso pensar em como dividir os dados (de forma a balancear a carga), e sim, qual core deve executar a tarefa, qual core deve receber e somar os dados, portanto é possível argumentar que é um paralelismo de tarefa.

**2.1. When we were discussing floating point addition, we made the simplifying assumption that each of the functional units took the same amount of time. Suppose that fetch and store each take 2 nanoseconds and the remaining operations each take 1 nanosecond.**
**a. How long does a floating point addition take with these assumptions?**
**b. How long will an unpipelined addition of 1000 pairs of floats take with these assumptions?**
**c. How long will a pipelined addition of 1000 pairs of floats take with these assumptions?**
**d. The time required for fetch and store may vary considerably if the operands/results are stored in different levels of the memory hierarchy. Suppose that a fetch from a level 1 cache takes two nanoseconds, while a fetch from a level 2 cache takes five nanoseconds, and a fetch from main memory takes fifty nanoseconds. What happens to the pipeline when there is a level 1 cache miss on a fetch of one of the operands? What happens when there is a level 2 miss?**

a) 2ns + 5*2ns + 2ns = 9ns

b) Uma única instrução é executada por vez, então, 1000*9ns = 9000ns

c) A primeira leva 9ns e as seguintes vão terminar a cada 2ns, então 9ns + 999*2ns = 2007ns

d) Quando existe cache miss level 1, o tempo total é acrescentado de 5ns, e no caso de level 2, o tempo total é acrescentado de 5 + 50 = 55ns. Nesse meio tempo, o pipeline deixa de funcionar e a instrução fica aguardando os dados.

**2.2. Explain how a queue, implemented in hardware in the CPU, could be used to improve the performance of a write-through cache.**

É responsabilidade da CPU escrever na memória e na cache, toda vez que acontece uma instrução de store. Logo, sem uma “queue”, seria impossível fazer duas instruções de store muito próximas

uma da outra, pois o tempo de escrita na memória principal não permitiria. Assim, uma “queue” seria responsável por acumular essas instruções de store que foram temporalmente próximas, e permitir que as outras instruções continuem sendo executadas, enquanto a escrita na memória é paralelamente executada pela fila.

**2.3. Recall the example involving cache reads of a two-dimensional array (page 22). How does a larger matrix and a larger cache affect the performance of the two pairs of nested loops? What happens if MAX = 8 and the cache can store four lines? How many misses occur in the reads of A in the first pair of nested loops? How many misses occur in the second pair?**

O efeito de uma matriz ou uma cache maior depende de quanto maior uma é em relação à outra. Uma maior matriz é de se esperar que o número de cache miss aumente, especialmente se o tamanho da linha de cache for menor que o tamanho da linha da matriz, no caso do primeiro loop, e o segundo loop continuaria com cache misses em todas as leituras. Porém, se o número de linhas de cache aumentar, é possível que o segundo loop seja beneficiado se todas as linhas couberem na cache, ainda mais que o primeiro loop, pois a cada linha adicionada na cache, o loop poderia ler o primeiro elemento e depois voltar para ler o segundo que ainda estaria na cache e assim por diante. No caso de MAX=8 e a cache de 4 linhas, o primeiro loop terá 8 cache misses. Ele deve errar o primeiro, fazer o fetch da linha e acertar o restante da linha. E assim por diante com as linhas seguintes. O segundo loop deve ter 64 cache misses, pois todos os acessos serão errados também.

**2.5. Does the addition of cache and virtual memory to a von Neumann system change its designation as an SISD system? What about the addition of pipelining? Multiple issue? Hardware multithreading?**

A adição de cache e memória virtual não alteram o estado de SISD da arquitetura von Neumann, já que o fluxo de instruções sendo executado ao mesmo tempo continua sendo apenas uma. Porém a adição de pipeling, multiple issue pode mudar esse estado, já que tanto no pipeline quanto no multiple issue, existe mais de uma instrução sendo executada ao mesmo tempo. O hardware multithreading, contudo, não permite a execução simultânea de instruções (exceto quando for Simultaneous multithreading), por isso seria ainda SISD, pois apenas uma instrução esta sendo executada.

**2.7. Discuss the differences in how a GPU and a vector processor might execute the following code:**

***sum = 0.0;***

***for ( i = 0; i < n ; i ++) {***

```
        y [ i ] += a * x [ i ];
        sum += z [ i ]* z [ i ];
}
```

O processador vetorial provavelmente iria executar as somas de y e sum separadamente de maneira sequencial, como em dois loops distintos, já que essas somas são completamente independentes. A operação de multiplicação poderia ser executada de maneira completamente vetorial, e a soma poderia ser executada como na árvore de soma mostrada no capítulo anterior, com o auxílio de registradores auxiliares.

A GPU provavelmente poderia executar os dois procedimentos de soma de maneira paralela, devido a independencia de seus nucleos, e as operações de mutiplicação e soma de cada linha poderiam ser executadas de maneira vetorial e utilizando a árvore de soma, dentro de cada núcleo.

**2.10. Suppose a program must execute 10^12 instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in 10 6 seconds (about 11.6 days). So, on average, the single processor system executes 10 6 or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute 10^(12) /p instructions and each processor must send 10^(9)*( p − 1) messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all of its messages, and there won't be any delays due to things such as waiting for messages.**

**a. Suppose it takes 10^−9 seconds to send a message. How long will it take the program to run with 1000 processors, if each processor is as fast as the single processor on which the serial program was run?**

**b. Suppose it takes 10^−3 seconds to send a message. How long will it take the program to run with 1000 processors?**

a)

Instruções por core = 10^12/10^3 = 10^9 ins

Tempo total = tempo execução instrução + tempo troca de mensagem

Cada processador executa 10^6(inst/sec)

tempo execução instrução = 10^9 (inst) * 1/10^6 (sec/inst) = 10^3(sec)

Total de mensagens = 10^9(10^3-1) = 10^12 - 10^9

tempo troca de mensagem = (10^12-10^9)(mens)*(10^-9)(sec/mens) = 999 sec

logo Tempo total = 10^3(sec) + 999 sec

b) tempo execução instrução = 10^9 (inst) * 1/10^6 (sec/inst) = 10^3(sec)
tempo troca de mensagem = (10^12-10^9)(mens)*(10^-3)(sec/mens) = 9.99*10^8 sec = 11563 dias
tempo total = 11563 dias + 10^3 segundos

**2.15. a. Suppose a shared-memory system uses snooping cache coherence and write-back caches. Also suppose that core 0 has the variable x in its cache, and it executes the assignment x = 5 . Finally suppose that core 1 doesn't have x in its cache, and after core 0's update to x , core 1 tries to execute y = x . What value will be assigned to y ? Why?**
**b. Suppose that the shared-memory system in the previous part uses a directory-based protocol. What value will be assigned to y ? Why?**
**c. Can you suggest how any problems you found in the first two parts might be solved?**
a. Como o valor de x não está no Core 1, a mensagem de atualização da variável será ignorada. Porém, logo em seguida é feita a leitura de x pelo Core 1 e, portanto ele vai procurar na memória o valor de x. Sendo o sistema de cache write-back, não é possível saber se x foi atualizado na memória ou não, mas é provável que não pelo curto espaço de tempo. Logo, o valor de y no core 1 deve ser o valor de x disponível na memória.
b. De forma semelhante, como x não está no cache de core 1, a mensagem de invalidez será ignorada, mas quando acessar a memória, o core 1 não deve encontrar o valor correto de x, já que o cache write-back de core 0 não deve ter sido executado.
c. Para a primeiro opção, trocar o sistema para write-through apesar de custoso, deve solucionar o problema.

**2.16. a. Suppose the run-time of a serial program is given by T serial = n^2 , where the units of the run-time are in microseconds. Suppose that a parallelization of this program has run-time T parallel = n 2 /p + log 2 (p). Write a program that finds the speedups and efficiencies of this program for various values of n and p. Run your program with n = 10, 20, 40, . . . , 320, and p = 1, 2, 4, . . . , 128. What happens to the speedups and efficiencies as p is increased and n is held fixed? What happens when p is fixed and n is increased?**
**b. Suppose that T parallel = T serial /p + T overhead . Also suppose that we fix p and increase the problem size.**
**- Show that if T overhead grows more slowly than T serial , the parallel efficiency will increase as we increase the problem size.**

**- Show that if, on the other hand, T overhead grows faster than T serial , the parallel efficiency will decrease as we increase the problem size.**

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
  for(int n=10; n <= 320; n=n*2){
    for(int p=1; p <= 128; p=p*2){
      cout << "n = " << n << " p = " << p << endl;
      float tserial = n*n;
      float tparallel = (n*n)/(float)p + log2(p);
      cout << "tserial: " << tserial << " tparallel: " << tparallel << endl;
      float speedup = tserial/tparallel;
      float efficiency = speedup/p;
      cout << "speedup: " << speedup << " efficiency: " << efficiency << endl;
    }
  }
  return 0;
}
```

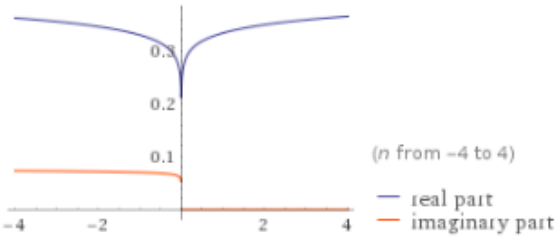
| speedup | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| 10 | 100,00% | 196,08% | 370,37% | 645,16% | 975,61% | 1230,77% | 1322,31% | 1285,14% |
| 20 | 100,00% | 199,00% | 392,16% | 754,72% | 1379,31% | 2285,71% | 3265,31% | 3950,62% |
| 40 | 100,00% | 199,75% | 398,01% | 788,18% | 1538,46% | 2909,09% | 5161,29% | 8205,13% |
| 80 | 100,00% | 199,94% | 399,50% | 797,01% | 1584,16% | 3121,95% | 6037,74% | 11228,07% |
| 160 | 100,00% | 199,98% | 399,88% | 799,25% | 1596,01% | 3180,12% | 6305,42% | 12367,15% |
| 320 | 100,00% | 200,00% | 399,97% | 799,81% | 1599,00% | 3195,01% | 6376,09% | 12688,97% |

| efficiency | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|---|---|
| 10 | 100,00% | 98,04% | 92,59% | 80,65% | 60,98% | 38,46% | 20,66% | 10,04% |
| 20 | 100,00% | 99,50% | 98,04% | 94,34% | 86,21% | 71,43% | 51,02% | 30,86% |
| 40 | 100,00% | 99,88% | 99,50% | 98,52% | 96,15% | 90,91% | 80,65% | 64,10% |
| 80 | 100,00% | 99,97% | 99,88% | 99,63% | 99,01% | 97,56% | 94,34% | 87,72% |
| 160 | 100,00% | 99,99% | 99,97% | 99,91% | 99,75% | 99,38% | 98,52% | 96,62% |
| 320 | 100,00% | 100,00% | 99,99% | 99,98% | 99,94% | 99,84% | 99,63% | 99,13% |

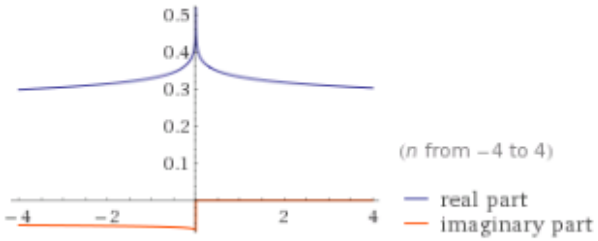
É possível observar que, para o speedup, se n é mantido fixo, o speedup sempre cresce, mas a uma taxa menor. Se p é mantido fixo, é também possível perceber que o speedup cresce, para todo p

maior que 1. No caso da efficiency, se o n é mantido constante, é obsercador que a eficiência diminui rapidamente em n's pequenos e mais devagar em n's grandes. Se p for mantido constante, vemos que a tendência é que a eficiência aumente.

b) Em um caso limite, onde Toverhead = n^1.9, por exemplo, percebemos que a curva da eficiência para um p constante eff = n^2/(p*(n^2/p + n^1.9)), tende a uma assíntota, mas continua crescendo no infinito. Para o outro caso limite onde Toverhead = n^2.1, temos que a curva também tende a uma assíntota, mas de forma decrescente, logo sempre diminuindo até o infinito.




Toverhead = n^1.9




Toverhead = n^2.1

**2.17. A parallel program that obtains a speedup greater than p—the number of processes or threads—is sometimes said to have superlinear speedup. However, many authors don't count programs that overcome "resource limitations" as having superlinear speedup. For example, a program that must use secondary storage for its data when it's run on a single processor system might be able to fit all its data into main memory when run on a large distributed-memory system. Give another example of how a program might overcome a resource limitation and obtain speedups greater than p.**

-Um programa que trabalha com matrizes, pouco maiores que a cache disponível, em um sistema maior pode conseguir colocar todos os elementos de uma matriz se for para um sistema maior, com mais cache e assim se beneficiar melhor da localidade espacial e temporal.
-Um programa que precisa de 8 threads e roda em um sistema com apenas núcleos, pode se beneficiar em um sistema de 4 núcleos, mas que possua threads de hardware ou Simultaneous multithreading.

**2.18. Look at three programs you wrote in your Introduction to Computer Science class. What (if any) parts of these programs are inherently serial? Does the inherently serial part of the work done by the program decrease as the problem size increases? Or does it remain roughly the same?**

**Sequencia de Fibbonacci**

O algoritmo comum da sequencia de fibnacci é

```
static long Fibonacci(long n)
{
  if (n < 2)
  {
    return n;
  }
  long one = Fibonacci(n-1);
  long two = Fibonacci(n-2);
  return one+two;
}
```

Ele pode ser paralelizado abrindo threads para Fibonacci(n-1) e Fibonacci(n-2) mas isso pode abrir mais treads que o desejado. Nesse caso, a parte completamente serial seria o retorno de n e abertura das threads, que deve ser constante.

Outra forma seria calcular de forma serial os k primeiros termos, e depois calcular pela seguinte formula `F(n + k) = F(n + 1) * F(K) + F(n) * F(k – 1),` os k seguinte termos de maneira paralela. Dessa forma a parte serial estaria no calculo inicial, e a medida que o n for aumentando, a parte serial diminui.

**Mutiplicação de Matrizes**

A multiplicação de matrizes pode ser beneficiada do paralelismo de dados, considerando que a matriz pode ser resolvida como se fosse a mutiplicação de matrizes de vários blocos diferentes, e estes blocos podem ser resolvidos em paralelo. A porção serial desse programa está na divisão dos dados entre as threads, inicialização da matriz resultado e inicialização das threads. A medida que o tamanho da matriz aumenta, é possível alocar mais threads, o que implica que o tamanho da parte paralelizável aumenta, ainda mais do que o overhead de criar as threads.

**Ordenação de Vetores**

O MergeSort se utiliza da segmentação do vetor e ordenação de partes do vetor em momentos distintos para depois fazer a concatenação. É possível paralelizar essa ordenação e concatenando o vetor de forma semelhante ao que acontecia na árvore de soma. Assim, a parte serial seria a divisão dos dados e criação das threads. E ela deve diminuir de acordo com o aumento do número de dados para ordenar.

**2.19. Suppose T serial = n and T parallel = n/p + log 2 (p), where times are in microseconds. If we increase p by a factor of k, find a formula for how much we'll need to increase n in order to maintain constant efficiency. How much should we increase n by if we double the number of processes from 8 to 16? Is the parallel program scalable?**

Isolar X para E = n/(n + plog2(p)) = xn/(xn + kp.log2(kp))
X = k log2(kp)/log2(p), se k = 2, x = 8/3

**2.20. Is a program that obtains linear speedup strongly scalable? Explain your answer.**
A eficiência de um programa que obtem speedup linear será sempre 1. Portanto, da definição de fortemente escalável, nós temos que se dobramos o número de cores e a eficiência permanecer constante, como é o caso no speedup linear, o programa é fortemente escalável.

**2.21. Bob has a program that he wants to time with two sets of data, input_data1 and input_data2. To get some idea of what to expect before adding timing functions to the code he's interested in, he runs the program with two sets of data and the Unix shell command time:**
**$ time ./ bobs prog < input data1**
**real 0 m0 .001 s**
**user 0 m0 .001 s**
**sys 0 m0 .000 s**
**$ time ./ bobs prog < input data2**
**real 1 m1 .234 s**
**user 1 m0 .001 s**
**sys 0 m0 .111 s**
**The timer function Bob is using has millisecond resolution. Should Bob use it to time his program with the first set of data? What about the second set of data? Why or why not?**
Com o primeiro dataset, o programa indica o mínimo possível na resolução do timer para tempo real e user, que seria 0.001s. E o tempo de sistema é indicado como 0s. Já para o segundo data set, a gente percebe que o tempo real e de user tem uma diferença significativa em milisegundos, e o tempo de sistema não é zero. Logo, é possível entender pouco do programa rodando o primeiro dataset quando o programa timer é utilizado, pois a resolução de milisegundos não informa muito sobre os tempos. Por outro lado, já conseguimos notar algumas diferenças para o dataset 2 e por isso pode-se utilizar o time com input_data2, mas não com input_data1.

**2.22. As we saw in the preceding problem, the Unix shell command time reports the user time, the system time, and the "real" time or total elapsed time. Suppose that Bob has defined the following functions that can be called in a C program:**
**double utime (void);**
**double stime (void);**

**double rtime (void);**

**The first returns the number of seconds of user time that have elapsed since the program started execution, the second returns the number of system seconds, and the third returns the total number of seconds. Roughly, user time is time spent in the user code and library functions that don't need to use the operating system—for example, sin and cos . System time is time spent in functions that do need to use the operating system—for example, printf and scanf.**

**a. What is the mathematical relation among the three function values? That is, suppose the program contains the following code:**

**u = double utime (void);**

**s = double stime (void);**

**r = double rtime (void);**

**Write a formula that expresses the relation between u , s , and r . (You can assume that the time it takes to call the functions is negligible.)**

r = s + u + stall_time, onde stall_time é o tempo em que o processo não esta sendo executado por algum motivo como acesso à memória ou troca de processos.

**b. On Bob's system, any time that an MPI process spends waiting for messages isn't counted by either utime or stime , but the time is counted by rtime . Explain how Bob can use these facts to determine whether an MPI process is spending too much time waiting for messages.**

O tempo gasto por espera de mensagens MPI está incluso em stall_time, logo stall_time = r – s -u. Assim, dependendo da relação entre stall_time e as outras variáveis é possível entender se o tempo gasto está sendo excessivo. Se considerarmos que o tempo de não-execução do programa é bem menor que o tempo de espera de mensagens, é possível ter uma boa ideia do gasto de espera de mensagens MPI

**c. Bob has given Sally his timing functions. However, Sally has discovered that on her system, the time an MPI process spends waiting for messages is counted as user time. Furthermore, sending messages doesn't use any system time. Can Sally use Bob's functions to determine whether an MPI process is spending too much time waiting for messages? Explain your answer.**

Como o tempo de espera está incluso no tempo de usuário, será muito dificil verificar se o tempo de espera por mensagens está excessivo, por é de se esperar que o tempo de execução do código (sem as mensagens) seja muito maior que o tempo de espera, pois apenas assim seria compensado o tempo de espera. Portanto é difícil separar esses dois tempos sem modificar o código testado.

**2.23. In our application of Foster's methodology to the construction of a histogram, we essentially identified aggregate tasks with elements of data . An apparent alternative would be to identify aggregate tasks with elements of bin counts, so an aggregate task would consist of all increments of bin counts[b] and consequently all calls to Find bin that return b . Explain why this aggregation might be a problem.**

Seria um problema pois o número de tasks que serão resolvidas em paralelo será muito menor. Já que o tempo de Fin_bin() é muito maior que o tempo de fazer um incremento em bin_counts, a maior parte do tempo será gasta em um código não paralelizável. E ainda, se bin_counts for uma váriável compartilhada, pode acontecer condições de corrida que exigem a existencia de um lock na operação, o que inviabiliza a aglomeração de todas as tarefas de incremento  de bin_count

**2.24. If you haven't already done so in Chapter 1, try to write pseudo-code for our tree-structured global sum, which sums the elements of loc bin cts . First consider how this might be done in a shared-memory setting. Then consider how this might be done in a distributed-memory setting. In the shared-memory setting, which variables are shared and which are private?**

**Shared Memory**

```
num_threads = numero de processadores disponiveis
shared array loc_bin_cnts[num_threads]
depois de preencher o array com os valores da soma, continue
start threads();
meu_id = get_thread_id()
for( i =0; i < ceiling(log2(num_threads)); i++)
        if(meu_id%2^(i+1) == 0)
                if (! meu_id+2^i > num_cores-1)
                        loc_bin_cnts[meu_id] += loc_bin_ctns[meu_id+2^i]
```

**Distributed-Memory**

```
num_processo = numero de processadores disponíveis
for( i =0; i < ceiling(log2(num_processes)); i++)
        my_id = Get_rank ();
        if(meu_id%2^(i+1) == 0)
                if (meu_id+2^i > num_cores-1)
                        other_loc_bin_cnts = 0
```

```
        else
                other_loc_bin_ctns = Receive( other_loc_bin_ctns , int , 1, meu_id+2^i);
        loc_bin_cnts += other_loc_bin_ctns
eles
        Send ( loc_bin_cts, int , 1, meu_id – 2^i);
        mate o processo
```