Supercomputação – Avaliação Final

Instruções

Bem-vindo(a)s à prova final da disciplina de Supercomputação. Leia as instruções abaixo:

SOBRE HORÁRIOS:

- Abertura das salas (07h15); início da prova (07h30); final (09h30); limite (10h30);
- O tempo máximo deve ser utilizado para fazer a prova, preparar a submissão e entregá-la;

SOBRE SUBMISSÕES DA PROVA:

- A submissão da prova deve ser feita impreterivelmente até às 10h30 de 24/maio/2024 (horário de Brasília). NÃO serão aceitas submissões após este horário;
- O aluno pode fazer múltiplas submissões. O sistema considerará a última como oficial;
- A submissão da prova deve ser um ZIP de uma pasta principal**, nomeada com seu nome completo**, e organizada com uma subpasta para cada questão. Na pasta da questão deve ter (i) o código-fonte, (ii) o arquivo de submissão ao Slurm (.slurm) (se aplicável), (iii) o arquivo de saída do Slurm (ou um print comprovando a execução) e (iv) um TXT com suas respostas textuais (se aplicável). ATENÇÃO: nas questões teóricas, pode enviar apenas o TXT das respostas na pasta; na questão de GPU envie seu *jupyter notebook* baixado do Colab.

SOBRE A RESOLUÇÃO DA PROVA:

- É permitida a consulta ao material da disciplina (tudo o que estiver no repositório do Github da disciplina e no site http://insper.github.io/supercomp). Isso também inclui suas próprias soluções aos exercícios de sala de aula, anotações e documentações de C++ e bibliotecas pertinentes em sites oficiais;
- Execuções de código no cluster necessitam, obrigatoriamente, serem realizadas via submissão de Jobs não interativos. A execução de códigos da prova diretamente na linha de comando do SMS-HOST fere a boa prática de utilização. O log do cluster será monitorado, e caso seja constatado que o aluno rodou código sem submeter o job, a nota máxima da questão será 50% do valor original. Ex: a questão vale 3, e o aluno acertou tudo, mas executou de forma incorreta diretamente no nó principal, então a nota da questão será 1,5.

SOBRE QUESTÕES DE ÉTICA E PLÁGIO:

- A prova é individual. Qualquer consulta a outras pessoas durante a prova constitui violação do código de ética do Insper;
- Qualquer tentativa de fraude, como trechos idênticos ou muito similares, ou uso de GenAI, implicará em NOTA ZERO na prova a todos os envolvidos, sem prejuízo de outras sanções.

BOA PROVA! \o/

Questões

A interpretação do enunciado faz parte da avaliação. Em caso de dúvida, pode assumir premissas e explicá-las nos comentários.

Questão 1 – Submissão de Jobs (Total: 1 ponto)

Esta questão dá dicas para as próximas quando você precisará submeter jobs no cluster. Além disso, avalia sua compreensão de como solicitar recursos ao cluster via arquivo ".slurm".

Considere que:

- O parâmetro "`--mem=<tamanho>[unidades]`" especifica a memória real necessária por nó. As unidades padrão são megabytes. Diferentes unidades podem ser especificadas usando o sufixo [K|M|G|T].
- Você pode definir o número de threads usando a variável de ambiente OMP_NUM_THREADS. Uma forma simples é antes de chamar o executável fazer o "export" da variável de ambiente da seguinte forma "`export OMP_NUM_THREADS=<number of threads to use>`".

Pede-se:

- Crie um arquivo "script.slurm" que solicita 2 máquinas, com 5 cores por máquina, e uma quantidade de 3 gigas de memória. O job deve ser executado numa partição chamada "prova", exportando a variável de ambiente OMP_NUM_THREADS para usar 10 threads, e solicitar a execução de um código fictício denominado "./executavel"
- Nesta questão entregue apenas o .slurm criado
- NOTA: não é necessário submeter o job no cluster! Esta é uma questão teórica! ;D

```
#!/bin/bash
#SBATCH --job-name=job_prova            # Define o nome do job
#SBATCH --partition=prova               # Define a partição onde o job será executado
#SBATCH --nodes=2                       # Solicita 2 nós
#SBATCH --ntasks-per-node=1             # Define o número de tarefas por nó (1 tarefa por nó)
#SBATCH --cpus-per-task=5               # Define 5 CPUs por tarefa (total de 5 CPUs por nó)
#SBATCH --mem=3G                        # Solicita 3 GB de memória para o job
#SBATCH --output=job_prova_%j.out       # Direciona a saída padrão para um arquivo de saída
#SBATCH --error=job_prova_%j.err        # Direciona a saída de erro para um arquivo

# Configuração do ambiente
export OMP_NUM_THREADS=10               # Define o número de threads do OpenMP

# Executa o programa
./executavel
```

Questão 2 – Paralelização Híbrida com MPI e OpenMP (Total: 3 pontos)

Desenvolva um programa que utiliza MPI para distribuir partes de um grande vetor (tamanho=100000) entre múltiplos processos e usa OpenMP dentro de cada processo para calcular a soma do quadrado de cada elemento do subvetor recebido. Cada processo deve enviar seu resultado parcial de volta ao processo raiz, que calculará a soma total dos quadrados.

Configuração do job: o código deve usar 4 processos, e 5 cores por máquina. A escolha do número máximo de threads está aberta. Considere que o nó principal também realizará parte dos cálculos.

```
#include <mpi.h>
#include <omp.h>
#include <vector>
#include <iostream>
#include <numeric>

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

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

    const int N = 100000; // Tamanho do vetor
    const int local_size = N / size; // Tamanho do subvetor para cada processo
    std::vector<double> full_vector, sub_vector(local_size);

    // Inicializa o vetor no processo raiz
    if (rank == 0) {
        full_vector.resize(N);
        for (int i = 0; i < N; ++i) {
            full_vector[i] = i;
        }
    }

    // Distribui partes do vetor para todos os processos
    MPI_Scatter(full_vector.data(), local_size, MPI_DOUBLE,
                sub_vector.data(), local_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Cálculo do quadrado dos elementos do subvetor usando OpenMP
    double local_sum = 0;
    #pragma omp parallel for reduction(+:local_sum) num_threads(5)
    for (int i = 0; i < local_size; ++i) {
        local_sum += sub_vector[i] * sub_vector[i];
    }

    // Agregar os resultados locais no processo raiz
    double total_sum = 0;
    MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    // Processo raiz exibe o resultado final
    if (rank == 0) {
        std::cout << "Total sum of squares: " << total_sum << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

```
#!/bin/bash
#SBATCH --job-name=hybrid_mpi_openmp      # Define o nome do job
#SBATCH --output=result_%j.txt            # Salva a saída em um arquivo de texto
#SBATCH --partition=prova                 # Especifica a partição
#SBATCH --nodes=2                         # Solicita 2 nós
#SBATCH --ntasks-per-node=2               # Solicita 2 tarefas por nó
#SBATCH --cpus-per-task=5                 # Solicita 5 CPUs por tarefa
#SBATCH --mem=3G                          # Define a memória total de 3GB

export OMP_NUM_THREADS=5                  # Define 5 threads para o OpenMP

# Carrega os módulos necessários, se aplicável
# module load mpi/openmpi-x.y.z

# Executa o aplicativo
mpirun ./nome_do_executavel
```

Questão 3 – Paralelização de código sequencial (Total: 2.5 pontos)

Contexto:

Você recebeu um código que executa a multiplicação de matrizes de forma sequencial. Este código é usado em um aplicativo de processamento de dados para realizar transformações lineares em grandes conjuntos de dados. Devido ao tamanho das matrizes envolvidas, a execução atual é ineficiente e demorada.

Objetivo:

Seu trabalho é paralelizar este código usando OpenMP, aplicando estratégias sofisticadas de pragmas para otimizar a execução. Você deve identificar os melhores pontos para introduzir paralelismo e escolher os pragmas mais adequados para maximizar a eficiência do código.

Configuração do job: seu código deve ser testado sob 3 tamanhos de matrizes diferentes (escolha quais você quiser) ***E*** 3 configurações distintas de ambientes, sempre submetendo-os na partição "prova" e alocando uma máquina, mas usando 2 cores, 4 cores ou 8 cores.

NOTA: espera-se, portanto, 9 execuções!

Código sequencial: obtenha neste link → https://colab.research.google.com/drive/1jxLtNYW0PaoCWBUEMRw2j4FVtiLiR3rC?usp=sharing

Análise do resultado: apresente também num arquivo TXT a comparação dos tempos de execução sequencial, e nas 9 execuções. O tempo de execução agiu conforme você esperava? Comente e justifique potenciais distorções de expectativa x realidade.

```
#include <iostream>
#include <vector>
#include <ctime>
#include <cstdlib>
#include <omp.h>

const int N = 500;  // Dimension of the matrix

void matrixMultiply(const std::vector<std::vector<double>>& A, const std::vector<std::vector<double>>& B,
std::vector<std::vector<double>>& C) {
    // Paralelizando o loop externo com OpenMP
    #pragma omp parallel for collapse(2) schedule(dynamic) shared(A, B, C)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            double sum = 0.0;
            for (int k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}

int main() {
    // Initialize matrices
    std::vector<std::vector<double>> A(N, std::vector<double>(N));
    std::vector<std::vector<double>> B(N, std::vector<double>(N));
    std::vector<std::vector<double>> C(N, std::vector<double>(N));

    // Randomly assign values to A and B
    srand(time(NULL));
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 100;
            B[i][j] = rand() % 100;
        }
    }

    // Timing the multiplication
    double start_time = omp_get_wtime();
    matrixMultiply(A, B, C);
    double time_taken = omp_get_wtime() - start_time;
    std::cout << "Time taken for matrix multiplication: " << time_taken << " seconds." << std::endl;

    // Optionally print matrix C
    // for (int i = 0; i < N; i++) {
    //     for (int j = 0; j < N; j++) {
    //         std::cout << C[i][j] << " ";
    //     }
```

```
    //     std::cout << std::endl;
    // }

    return 0;
```

Continuação a cargo do aluno, desde que evidenciasse as execuções, fizesse a configuração correta do Slurm e comparasse os resultados de forma pertinente.

Questão 4 – Normalização de Vetor com Thrust (Total: 2 pontos)

1. Acesse o link abaixo e faça uma cópia no seu Google Drive: https://colab.research.google.com/drive/14_EZNglXn2VXe3kpDW3XgEsRkB6R1jjp?usp=sharing
2. Complete o código seguindo a especificação. ATENÇÃO: você provavelmente precisará complementar os imports para o código rodar!
3. Baixe sua cópia do notebook preenchido e executado, e disponibilize na pasta desta questão

Solução:

https://colab.research.google.com/drive/1UUr9P2RBxeNhepg4MNGZfKL43PxlMpOF?usp=sharing

Questão 5 – Fundamentos de Paralelismo (Total: 1.5 pontos)

- **Parte A (0,75 ponto)**: Descreva o modelo de memória compartilhada e memória distribuída, ressaltando seus prós e contras. Qual modelo cada uma das seguintes bibliotecas usa: OpenMP, MPI e Thrust?
- **Parte B (0,75 ponto)**: Explique o que é escalonamento dinâmico em OpenMP e como ele pode ser vantajoso em aplicações de processamento de dados com variabilidade de carga entre as iterações.

Parte A: Modelos de Memória Compartilhada e Distribuída

Modelo de Memória Compartilhada
No modelo de memória compartilhada, todos os processadores (ou threads) têm acesso a uma memória comum. As threads executam em paralelo e podem ler ou escrever nas mesmas áreas de memória.

Prós:
- **Facilidade de Programação**: Como todas as threads compartilham o mesmo espaço de memória, não há necessidade de explicitamente dividir dados entre elas.
- **Eficiência de Comunicação**: Não há overhead de comunicação entre processadores como ocorre no envio de mensagens.

Contras:
- **Escalabilidade**: À medida que o número de threads aumenta, o acesso concorrente à memória compartilhada pode se tornar um gargalo, limitando a escalabilidade.
- **Problemas de Sincronização**: Programas que utilizam memória compartilhada são susceptíveis a condições de corrida e outros problemas de sincronização, que podem ser difíceis de detectar e corrigir.

Bibliotecas:
- **OpenMP**: Utiliza o modelo de memória compartilhada.

- **Thrust**: Operando sobre CUDA para GPUs, Thrust abstrai tanto a memória compartilhada dentro de blocos de threads quanto a memória global da GPU, mas pode ser visto mais como um modelo de memória compartilhada em sua interface de alto nível.

Modelo de Memória Distribuída
No modelo de memória distribuída, cada processador tem sua própria memória local. A comunicação entre processadores é realizada através do envio e recebimento de mensagens.

Prós:
- **Escalabilidade**: Como cada processador opera independentemente, sistemas com memória distribuída podem escalar efetivamente com o aumento do número de processadores.
- **Isolamento de Falhas**: Falhas em um processador não afetam diretamente os outros, pois a memória não é compartilhada.

Contras:
- **Complexidade de Programação**: A necessidade de gerenciar explicitamente a comunicação entre processadores pode complicar o design do software.
- **Overhead de Comunicação**: O tempo necessário para enviar e receber mensagens pode ser significativo, especialmente se os dados necessários não estiverem localmente disponíveis.

Bibliotecas:
- **MPI (Message Passing Interface)**: Utiliza o modelo de memória distribuída, sendo projetado especificamente para sistemas com este tipo de arquitetura.

Parte B: Escalonamento Dinâmico em OpenMP

Definição de Escalonamento Dinâmico
O escalonamento dinâmico (`dynamic scheduling`) em OpenMP é um método para distribuir iterações de um loop entre threads de maneira dinâmica. Uma thread que termina suas iterações pode receber novas iterações à medida que elas se tornam disponíveis, o que ajuda a manter todas as threads ocupadas.

Vantagens em Aplicações com Variabilidade de Carga
- **Balanceamento de Carga**: O escalonamento dinâmico é particularmente útil em cenários onde as iterações do loop têm variabilidade significativa em seus tempos de execução. Ao permitir que as threads busquem trabalho adicional assim que terminam suas tarefas atuais, o escalonamento dinâmico ajuda a evitar que algumas threads fiquem ociosas enquanto outras ainda estão processando iterações pesadas.
- **Eficiência Melhorada**: Esse método maximiza a utilização da CPU ao reduzir o tempo de inatividade das threads, o que pode resultar em uma redução geral no tempo de execução do programa.
- **Flexibilidade**: O escalonamento dinâmico adapta-se bem a situações de carga de trabalho imprevisível ou quando as iterações têm dependências de dados complexas que afetam o tempo de execução.

Assim, o escalonamento dinâmico em OpenMP pode proporcionar uma melhoria significativa no desempenho de aplicações paralelas que enfrentam desafios de balanceamento de carga, tornando-o uma escolha eficaz para a otimização de algoritmos em ambientes de memória compartilhada.