RECIT
Revista Eletrônica Científica Inovação e Tecnologia
ISSN: 2175-1846

https://periodicos.utfpr.edu.br/recit

Exploração do paralelismo em arquiteturas multicore, multicomputadores e gpus

RESUMO

A computação paralela visa atender à demanda por alto poder computacional ao utilizar mais de um núcleo de processamento simultaneamente para resolver um problema. Diversas arquiteturas foram propostas para permitir uma melhor exploração do paralelismo, as quais geralmente são seguidas por modelos de programação e ferramentas apropriadas para extrair todo o potencial da arquitetura. Este trabalho apresenta as principais classificações de sistemas paralelos bem como três ferramentas popularmente utilizadas para criação de aplicações paralelas em diferentes arquiteturas. Além disso, o problema da transferência de calor foi utilizado como estudo de caso para exploração do paralelismo com estas ferramentas, obtendo resultados satisfatórios nas implementações paralelas em relação à implementação sequencial.
PALAVRAS-CHAVE: aplicações paralelas, OpenMP, MPI, CUDA

INTRODUÇÃO

A demanda por maior poder computacional tem motivado o desenvolvimento dos computadores desde suas origens. Mesmo com a tecnologia atual, muito superior àquela existente quando surgiram os primeiros sistemas computacionais, ainda há diversos problemas cuja execução requer tempo excessivamente longo em vista da alta demanda de computação. Além disso, a tendência é que esta demanda continue aumentando, uma vez que a solução de um problema é frequentemente acompanhada pelo surgimento de outros ainda mais complexos.

Historicamente, os avanços tecnológicos nos processadores – como o aumento da frequência e número de transistores dos mesmos – fez com que a capacidade de processamento crescesse de forma exponencial. Entretanto, a existência de limitações físicas e tecnológicas no desenvolvimento de um único processador – tais como o consumo de energia, a dissipação de calor e o limite físico do tamanho de transistores – restringe a evolução do desempenho computacional por meio apenas do uso de processadores mais rápidos.

Desse modo, a computação paralela surgiu como solução para atender à necessidade por capacidade computacional cada vez mais elevada. De modo geral, as estratégias paralelas buscam diminuir o tempo de execução de um programa utilizando mais de um núcleo de processamento simultaneamente. Ao passo que isso permite aumentar o poder de processamento disponível, paralelizar a execução de uma aplicação cria uma complexidade adicional à programação. Assim, foram criados modelos de programação bem como arquiteturas otimizadas para proporcionar uma execução mais eficiente de programas paralelos. Tais arquiteturas apresentam-se em diferentes níveis, desde o paralelismo ao nível de instruções no processador, passando por processadores com mais de um núcleo de processamento até a formação de *grids* e *clusters*, que combinam diversas máquinas em um único sistema. Geralmente, diferentes ferramentas são utilizadas para explorar o paralelismo em cada nível, uma vez que, comumente, cada nível apresenta modelos de programação distintos (SCHEPKE, 2009).

Sob este cenário, o objetivo deste trabalho é apresentar três das principais arquiteturas paralelas: sistemas de memória compartilhada, sistemas de memória distribuída e placas gráfics (GPUs), bem como os modelos de programação característicos destas arquiteturas. Junto a isso, o problema da transferência de calor foi utilizado como estudo de caso para exploração do paralelismo utilizando

três das principais ferramentas disponíveis para criação de aplicações paralelas: o OpenMP, o MPI e o CUDA. Essas implementações obtiveram resultados satisfatórios, com a versão em CUDA obtendo o melhor ganho de tempo, executando 49 vezes mais rápido que a implementação sem a exploração de paralelismo.

ARQUITETURAS E MODELOS DE PROGRAMAÇÃO PARALELA

De modo geral, a execução de um programa em um computador consiste em realizar uma sequência de instruções sobre um conjunto de dados. A partir disso, é possível classificar os computadores quanto ao paralelismo sobre o fluxo de dados e o fluxo de instruções. Essa ideia originou a Taxonomia de Flynn, proposta em 1966 e que ainda hoje é uma das principais maneiras de se classificar sistemas paralelos (SCHEPKE, 2009). Ela os divide em quatro classes:

- *Single Instruction Single Data* (SISD) – nesta classe, uma instrução opera sobre um único dado; logo, não há paralelismo presente. Os computadores com um único processador se encontram nesta classe.
- *Multiple Instruction Single Data* (MISD) – aqui, várias instruções operam sobre um único dado. Esta é uma classificação sem uso prático.
- *Single Instruction Multiple Data* (SIMD) – neste caso, uma única instrução é executada simultaneamente em diversos dados. As unidades de processamento gráfico (GPUs) utilizam esta estratégia para obter paralelismo.
- *Multiple Instruction Multiple Data* (MIMD) – por fim, esta categoria engloba as máquinas em que múltiplas instruções são executadas paralelamente sobre diferentes dados. Este seria o caso dos multiprocessadores, *grids* e *clusters*.

Os sistemas MIMD ainda podem ser divididos em sistemas de memória compartilhada e de memória distribuída. Em sistemas de memória compartilhada, os simultâneos fluxos de execução do programa (tarefas) tem acesso a um mesmo espaço de memória; assim, caso uma tarefa atualize um dado na memória global, a alteração já estará "visível" para as demais tarefas. Este é o caso de processadores *multi-core*, que contêm múltiplos núcleos de processamento em um único chip conectados a uma mesma memória. Nestes processadores, pode-se executar várias tarefas concorrentemente por meio do uso de *threads*. De modo geral, sistemas de memória compartilhada têm escalabilidade limitada em função

de restrições físicas: por exemplo, cada núcleo de processamento necessita acesso à memória, gerando um gargalo no sistema. Um método utilizado para minimizar este efeito é que cada núcleo possua uma memória *cache* local. Entretanto, isso resolve apenas parcialmente o problema, uma vez que gera problemas de coerência de *cache*: se um núcleo atualiza um dado em sua *cache*, é necessário replicar esta atualização na *cache* dos outros núcleos, caso o dado esteja compartilhado com as mesmas, para que se garante a integridade dos dados da *cache* (SCHEPKE, 2009).

Já em sistemas de memória distribuída, não há uma memória global e as tarefas não têm acesso à memória das demais. Caso elas queiram trocar dados, é necessário utilizar troca de mensagens de forma explícita. Exemplos de sistemas de memória distribuída são os *clusters* e *grids*, nos quais as múltiplas tarefas são executadas por processos nas diferentes máquinas. Estes sistemas são mais flexíveis e escalonáveis que os sistemas de memória compartilhada. Uma vez que os processadores estão conectados por uma rede – e não no nível de um *chip*, por exemplo – é possível adicionar e remover máquinas sem interferir no restante do conjunto. Naturalmente, não há gargalo no acesso à memória porque não há uma memória global. Entretanto, compartilhar dados passa a ser uma tarefa mais custosa, uma vez que requer troca de mensagens entre os processos; além disso, a complexidade dos programas sofre um aumento, uma vez que esta troca de mensagens passa a ser responsabilidade do programador (SCHEPKE, 2009).

Outro nível de paralelismo é apresentado pelos sistemas SIMD, representados principalmente pelas GPUs. Ao passo que os sistemas MIMD executam em paralelo processadores complexos, o que resulta em um tempo de latência – tempo para executar uma instrução – baixo, as GPUs focam em aumentar o *throughput*, que corresponde à taxa de execução de instruções por tempo. Por isso, as GPUs são processadores massivamente paralelos, contendo centenas ou até milhares de unidades de processamento (ao passo que os multiprocessadores, por exemplo, raramente apresentam mais de algumas dezenas de *cores*), mais simples que os processadores apresentados pelos sistemas MIMD. Isto significa que o tempo de latência não é tão bom como nos processadores dos sistemas MIMD, mas, uma vez que há um número considerável de unidades de processamento, esta deficiência acaba superada pelo ganho em *throughput*.

Em particular, as unidades de processamento de uma GPU são especializadas nas chamadas operações vetoriais – executar uma mesma instrução sobre não um, mas um conjunto de dados ao mesmo tempo, explorando o paralelismo de dados. Para somar dois vetores de oito elementos, por exemplo, um processador tradicional teria que realizar oito adições em sequência; já com operações vetoriais, as oito adições podem ser executadas concorrentemente, aproveitando-se do fato de que a instrução a ser executada pelos núcleos de processamento é a mesma – somar dois valores – alterando-se apenas os dados sobre os quais essa instrução será executada. Assim, as GPUs destinam mais transistores para operações lógicas e aritméticas e menos transistores para operações de controle de fluxo (SILVA, 2014).

Estas divisões das arquiteturas paralelas significam diferentes níveis de paralelismo a serem explorados. Observa-se que é possível combinar estas arquiteturas em um único sistema, formando plataformas com múltiplos níveis de paralelismo. Ferramentas que buscam criar uma camada de abstração para unificar a exploração dos vários níveis de paralelismo foram propostas, mas sem alcançar eficiência máxima em função dos diferentes modelos de programação observados em diferentes arquiteturas paralelas. Assim, as ferramentas mais utilizadas para criação de aplicações paralelas focam em um único nível de paralelismo e modelo de programação (MAILLARD; SCHEPKE, 2008). Neste trabalho, foram consideradas três ferramentas que atuam em níveis diferentes de paralelismo: o OpenMP, o MPI e o CUDA.

O OpenMP (*Open Multi-Processing*) é uma API que permite a criação de aplicações paralelas em sistemas de memória compartilhada. Esta extensão para linguagens sequenciais consiste em um conjunto de diretivas de compilação, funções de bibliotecas e variáveis de ambiente que determinam o comportamento de uma aplicação paralela em tempo de execução.

O OpenMP trabalha sobre o modelo *Fork-Join*, um paradigma para aplicações paralelas no qual uma *thread* principal (chamada "*master thread*") executa trechos de código sequencialmente e, em determinados pontos do programa, ramifica a execução ("*Fork*") em diversas *threads*, cada qual executando um trecho paralelo do programa. Ao final deste trecho, as *threads* são fundidas ("*Join*") novamente ao processo principal, que retoma a execução sequencial (CHAPMAN; JOST; PAS, 2008).

Já o MPI (*Message Passing Interface*) é um padrão para comunicação entre processos por meio da troca de mensagens que permite a criação de aplicações paralelas em sistemas de memória distribuída. O MPI define um conjunto de funções que cobrem os mecanismos envolvidos na comunicação entre processos, desde primitivas para comunicação entre os processos até a criação de tipos de dados derivados para serem utilizados na troca de mensagens. A comunicação mais básica presente no padrão é a comunicação ponto a ponto, envolvendo a troca de mensagens entre dois processos específicos. A partir da comunicação ponto a ponto, são construídos outros tipos de comunicação mais complexos, permitindo a troca de dados por um número maior de processos, chamados comunicações coletivas.

Diferentemente do OpenMP, que opera sobre o modelo *Fork-Join*, a maioria das implementações do MPI utilizam um modelo no qual um conjunto fixo de processos é criado ao início da execução do programa, cada qual com um identificador. Estes processos executam o mesmo programa, mas por vezes executam instruções diferentes devido a operações condicionais de controle de execução (as quais tipicamente são baseadas no identificador do processo). Esta técnica é chamada de "Programa único, múltiplos dados" (do inglês SPMD – *single-program, multiple-data*), sendo uma das abordagens mais comuns para a criação de aplicações paralelas (PACHECO, 1997).

Por fim, o CUDA (*Compute Unified Device Architecture*) é um ambiente de desenvolvimento criado pela NVIDIA para criação de aplicações paralelas em GPUs. Esta ferramenta permite que o programador defina funções especiais, chamadas *kernels*, as quais são executadas em paralelo em *threads* na GPU. O fluxo do programa é controlado pela CPU, que executa trechos de código sequencialmente. O paralelismo se dá quando a CPU ativa a GPU realizando uma chamada de *kernel*. As *threads* são organizadas em conjuntos lógicos de 1, 2 ou 3 dimensões chamados blocos de *threads*. Os blocos, por sua vez, estão organizados em um *grid* lógico de 1, 2 ou 3 dimensões. Cada bloco e cada *thread* possui um identificador único que distingue sua posição nas coordenadas *x*, *y* e *z* dentro do *grid*/bloco, respectivamente.

Em nível de *hardware*, as GPUs da NVIDIA são compostas por um conjunto de blocos físicos chamados *streaming multiprocessors* (SMs), os quais são compostos por um conjunto de núcleos de processamento. Por exemplo, a arquitetura de uma

GPU Kepler com o chip GK110 é composta por 15 SMs, cada qual contendo 192 núcleos de processamento (NVIDIA, 2012). Para que uma *kernel* seja executada na GPU, os blocos lógicos de *threads* definidos pelo programador são divididos em conjuntos com um tamanho fixo de *threads*, cada qual é alocado para um SM. Todos os elementos de processamento da SM executam a mesma instrução simultaneamente, porém sobre dados diferentes, ou seja, explorando o paralelismo SIMD. Entretanto, é possível que as *threads* sigam diferentes caminhos de fluxo e executem instruções diferentes, o que é chamado de divergência de código; nesse caso, a execução das *threads* no SM é serializada, causando perda na eficiência do paralelismo. Fica claro, então, que o modelo de programação CUDA obtém maior desempenho paralelo para programas que apresentam baixa necessidade de controle de fluxo em relação ao número de operações lógicas e aritméticas (SILVA, 2014).

ESTUDO DE CASO

Para avaliar a possibilidade de se paralelizar uma aplicação utilizando as ferramentas mencionadas, o problema da transferência de calor foi utilizado como estudo de caso. Este problema é um exemplo clássico de modelo matemático para uma simulação computacional de um fenômeno da vida real, o qual busca modelar a difusão de calor em sólidos a fim de se determinar a temperatura em qualquer ponto da placa em um dado instante de tempo (BORGES; PADOIN, 2006).

Em uma placa retangular, caso existam pontos submetidos a diferentes temperaturas, ao longo do tempo ocorrerá dissipação do calor por meio de troca de energia entre as partículas do material, dos pontos com mais energia para os pontos com menos, a fim de alcançar um equilíbrio natural. Em uma placa isotrópica e homogênea, na qual as propriedades térmicas do material são as mesmas em qualquer direção, este processo pode ser modelado por meio da equação do calor.

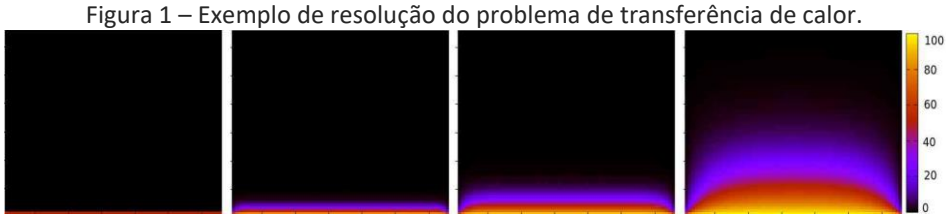
Para isso, é necessário que haja uma discretização do domínio (malha espacial), a qual consiste em considerar uma malha com distância fixa entre os nós de Δx no eixo das abscissas e Δy no eixo das ordenadas. Uma maneira de se realizar a discretização do processo é pela aplicação do Método das Diferenças Finitas, com o qual o processo é modelado pela seguinte equação (BORGES; PADOIN, 2006):

$$U^{n+1}_{i,j} = U^{n}_{i,j} + s_x (U^{n}_{i+1,j} - 2U^{n}_{i,j} + U^{n}_{i-1,j}) + s_z (U^{n}_{i,j+1} - 2U^{n}_{i,j} + U^{n}_{i,j-1})$$

Onde $U^{n}_{i,j}$ representa a temperatura no ponto *i, j* da malha na iteração *n*, a qual é definida em função de um intervalo de tempo arbitrário ($\Delta t = n_k - n_{k-1}$), e s_x, s_z representam constantes definidas em função das medidas escolhidas para as grandezas físicas e da difusividade térmica do material.

Os resultados de uma simulação do problema da transferência de calor a partir do processo descrito são apresentados na Figura 1, onde quatro diferentes momentos da simulação podem ser observados. A imagem retrata um experimento no qual uma fonte de calor (100ºC) é posta no lado inferior do domínio; conforme a simulação avança, o calor espalha-se ao longo da placa quadrada.

Figura 1 – Exemplo de resolução do problema de transferência de calor.



Fonte: Autoria própria.

O método apresentado para resolução do problema de transferência de calor requer grande quantidade de computação, uma vez que é necessário atualizar todos os pontos da malha em cada iteração. Assim, pode-se utilizar processamento paralelo para solucioná-lo de modo mais rápido, observando que, em uma iteração, a computação de cada ponto da malha independe das demais.

Assim, a solução sequencial utiliza duas matrizes: *grid* é a matriz que representa a malha com as temperaturas na iteração atual, enquanto *novo_grid* é a matriz que recebe os novos valores calculados para cada ponto. Ao final de cada iteração, os valores de *novo_grid* são transferidos para *grid*, e o contador do número de iterações é incrementado. O código sequencial é dado a seguir, com simplificações, a fim de comparação com as versões paralelas.

Código 1 – Implementação sequencial para o problema de transferência de calor.

```
1  while(cont_iteracoes < num_iteracoes)
2  {
3      for(i = 1; i <= linhas; i++)
4          for(j = 1; j <= colunas; j++)
```

```
5               novo_grid[i][j]  =  calcula_valor(grid,
    i, j);
6
7       troca_matriz(novo_grid, grid);
8       cont_iteracoes++;
9   }
```

Fonte: Autoria própria.

Já a solução usando OpenMP se baseia na criação de regiões paralelas do código nas quais cada *thread* processa um conjunto de pontos *i, j* da malha. Para tal, utilizou-se a diretiva *#pragma omp parallel for* que faz a divisão do trabalho entre as *threads*. Uma vez que a malha é armazenada em uma memória comum a todas as *threads*, os valores atualizados dos pontos da malha são compartilhados entre elas assim que são computados. Portanto, não é necessário realizar sincronização explícita dos dados; apenas é necessário sincronizar a execução das *threads* ao final de cada iteração, para evitar que uma *thread* inicie a próxima iteração enquanto outra ainda está na iteração atual (o que poderia causar com que a *thread* que iniciou a próxima iteração buscasse na memória um valor que não foi atualizado pela última iteração). Tal sincronização pode ser feita utilizando a diretiva *#pragma omp barrier*, que interrompe a execução da *thread* até que todas as *threads* alcancem a barreira.

O seguinte trecho de código representa a região paralela do código a fim de ilustrar o uso das diretivas do OpenMP. A diretiva *#pragma omp single* faz com que o trecho de código indicado pela mesma seja executado por uma única *thread*; além disso, há uma diretiva *#pragma omp barrier* implícita ao final da direta single.

Código 2 – Implementação com OpenMP para o problema de transferência de calor.

```
1  #pragma      omp      parallel      private(i,      j)
     shared(cont_iteracoes, grid, novo_grid)
2  {
3       while(cont_iteracoes < num_iteracoes)
4       {
5           #pragma omp for
6           for(i = 1; i <= linhas; i++)
7               for(j = 1; j <= colunas; j++)
8                   novo_grid[i][j]                    =
     calcula_valor(grid, i, j);
9
10          #pragma omp single
11          {
12              troca_matriz(novo_grid, grid);
```

```
13              cont_iteracoes++;
14          }
15      }
16  }
```

Fonte: Autoria própria.

Já a solução usando MPI consiste no particionamento do domínio físico do problema (a placa plana) em subdomínios, gerando uma paralelização natural da computação. Cada processo fica responsável pela computação de um dos subdomínios. Ao final de cada iteração do método, faz-se necessária a sincronização das fronteiras dos subdomínios (as quais são dependentes uma das outras uma vez que o cálculo de um ponto da malha requer o valor dos pontos adjacentes); isso é feito por troca de mensagens, utilizando as primitivas *MPI_Send* e *MPI_Recv*. A solução final do problema é obtida pela junção das soluções nos subdomínios.

O trecho a seguir apresenta a implementação com MPI. Observa-se que a primeira parte do código é idêntica ao código sequencial; entretanto, o *grid* a ser atualizado refere-se a um subdomínio do *grid* alocado para o processo executando o código. A segunda parte do código ilustra, de modo simplificado, o processo de troca de mensagens: para cada processo com o qual deve haver sincronização das fronteiras, os métodos *MPI_Send* e *MPI_Recv* são chamados com os devidos parâmetros da comunicação (como os dados a serem enviados/recebidos e o identificador do processo destino/fonte).

Código 3 – Implementação com MPI para o problema de transferência de calor.

```
1  while(cont_iteracoes < num_iteracoes)
2  {
3      for(i = 1; i <= linhas; i++)
4          for(j = 1; j <= colunas; j++)
5              novo_grid[i][j]  =  calcula_valor(grid,
       i, j);
6
7       troca_matriz(novo_grid, grid);
8       cont_iteracoes++;
9
10      for(i = 0; i < num_processos_adjacentes; i++)
11      {
12          MPI_Send(fronteira_para(i),
       tamanho_fronteira, MPI_DOUBLE,
       processo_adjacente(i), tag, MPI_COMM_WORLD);
```

```
13          MPI_Recv(fronteira_de(i),
      tamanho_fronteira, MPI_DOUBLE,
      processo_adjacente(i), tag, MPI_COMM_WORLD,
      &status);
14       }
15 }
```

Fonte: Autoria própria.

Por fim, a solução usando CUDA é baseada na criação de um *kernel* que computa um único ponto da malha. Este *kernel* é então lançado pela CPU para ser executado paralelamente pelas milhares de *threads* da GPU; para cada ponto da malha, há uma execução do *kernel* em uma dada *thread* que processará seu novo valor. Embora o número de *threads* na GPU seja alto, há mais pontos na malha que *threads*; assim, o CUDA fica responsável por escalonar as *threads* para que o *kernel* seja executado o número de vezes definido pelo programador (neste caso, uma execução para cada ponto da malha).

O código a seguir apresenta o *kernel* utilizado na solução em CUDA (linhas 1-11), bem como o código executado na CPU para ativar o *kernel* (linhas 12-18). A diretiva *__global__* indica ao compilador que esta função trata-se de um *kernel*, a ser executado na GPU. As três primeiras instruções do *kernel* calculam a linha e coluna da *thread* dentro da distribuição lógica do *grid* de *threads* a fim de determinar qual ponto da matriz a *thread* deve atualizar. Em seguida o novo valor deste ponto é calculado e atualizado na matriz *d_novo_grid*. O prefixo *d_* é frequentemente utilizado para indicar uma variável que foi alocada na memória da GPU. Nota-se também o número de blocos e número de *threads* por bloco, indicados pela CPU ao lançar o *kernel* (linha 14); haverá *num_blocos* * *tam_bloco* execuções do *kernel*, cada qual calculando um ponto da malha.

Código 4 – Implementação com CUDA para o problema de transferência de calor.

```
1  __global__ void calcula(double *d_grid, double
     *d_novo_grid)
2  {
3      int linha = blockIdx.x / 4;
4      int coluna = (blockIdx.x % 4) * (blockDim.x) +
     threadIdx.x;
5      int posicao = calcula_posicao(linha, coluna);
6      double novo_valor;
7      double antigo_valor = d_grid[posicao];
8
9      novo_valor = calcula_valor(d_grid, i, j);
```

```
10      d_novo_grid[posicao] = new_val;
11  }

        [...]

12  while(cont_iteracoes < num_iteracoes)
13  {
14      calcula<<<num_blocos,      tam_bloco>>>(d_grid,
        d_novo_grid);
15
16      troca_matriz(novo_grid, grid);
17      cont_iteracoes++;
18  }
```

Fonte: Autoria própria.

RESULTADOS

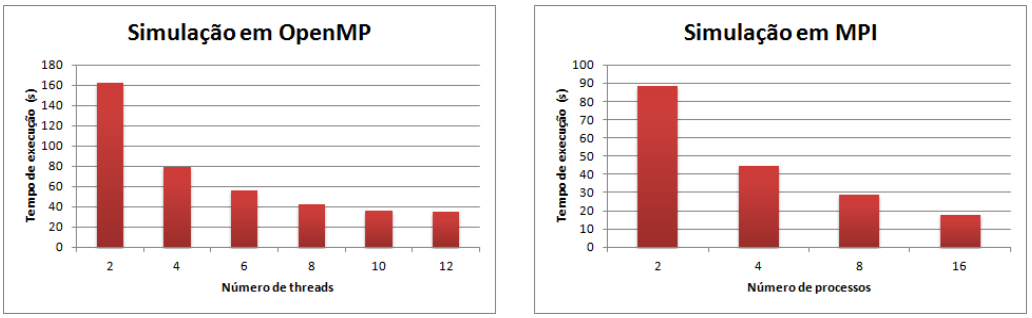
Esta seção apresenta os resultados das implementações paralelas do problema da transferência de calor usando OpenMP, MPI e CUDA. As simulações foram realizadas considerando um domínio de 4.096x4.096 células sobre 1024 iterações. As implementações foram executadas, respectivamente para cada ferramenta, em um servidor com 2 CPUs Xeon E5-2620 (totalizando 12 núcleos) e 128 GB de memória principal, em um *cluster* de 8 nodos de processadores Intel i5 com 4 *cores* cada um e em uma placa gráfica Tesla K20c contendo 2560 núcleos CUDA.

Analisando a arquitetura utilizada para as diferentes ferramentas, observa-se que a solução em MPI é a mais escalável uma vez que é executada em um *cluster*. Para obter mais tarefas executando em paralelo bastaria adicionar mais nodos ao *cluster*, ao passo que as execuções em OpenMP e CUDA são limitadas pelo número de núcleos do processador e da placa gráfica, respectivamente.

Entretanto, é notável que, na prática, o ganho de desempenho de uma aplicação paralela raramente é linear ao número de execuções em paralelo, visto que há trabalho extra para coordenar os fluxos de execução do programa. A otimização do tempo de execução da aplicação requer uma avaliação do balanceamento entre a carga de trabalho e o número de tarefas em paralelo. Neste sentido, a Figura 2 ilustra os resultados da implementação usando OpenMP e MPI com diferentes números de tarefas executando em paralelo. Tal variação não ocorre na versão utilizando CUDA, pois apenas uma placa gráfica foi utilizada,

empregando todas as *threads* disponíveis na mesma. A situação mencionada ocorre, por exemplo, nos tempos de execução da implementação usando OpenMP com 10 e 12 *threads*, no qual observou-se ganho mínimo de desempenho.

Figura 2 – resultados da implementação usando openmp e mpi com diferentes números de tarefas em paralelo.



Fonte: autoria própria.

Tabela 1 – menor tempo de execução por implementação.

Implementação	Tarefas em paralelo	Tempo de execução
Sequencial	1	308,95 s
OpenMP	12	34,59 s
MPI	16	17,43 s
CUDA	2560	6,23 s

Fonte: autoria própria.

Por fim, a Tabela 1 compara o tempo de execução obtido na versão sequencial (sem paralelismo) e nas versões paralelas utilizando OpenMP, MPI e CUDA (considerando os melhores tempo observados). Observa-se que nas três versões paralelas, houve resultados satisfatórios, reduzindo o tempo da execução sequencial em torno de 9, 17 e 49 vezes, respectivamente.

O melhor tempo de execução foi obtida pela versão com CUDA. Isso demonstra o poder da placa gráfica utilizada, com 2560 núcleos de processamento, mas também decorre do fato do problema em questão ser bem adaptado ao modelo de programação em CUDA: como pode-se observar no Código 4, praticamente não há operações de controle de fluxo no *kernel*, ao passo que há várias operações aritméticas. Assim, foi possível explorar todo o paralelismo oferecido pela GPU, sem que houvesse degradação de desempenho pela serialização de código.

CONCLUSÃO

O uso de arquiteturas paralelas é hoje uma abordagem indispensável em áreas que requerem grande capacidade de processamento. Para a construção de uma aplicação paralela, deve-se considerar as diferentes arquiteturas e modelos de programação paralela disponíveis, avaliando os níveis de paralelismo que podem ser explorados em função da natureza da aplicação.

Este trabalho apresentou três das principais arquiteturas paralelas e ferramentas utilizadas para programar nestas arquiteturas, e avaliou implementações paralelas com estas ferramentas para o problema da transferência de calor. Observou-se que a versão com CUDA apresentou o melhor tempo de execução, indicando a boa adaptação do problema ao modelo de programação apresentado pela GPU. Isso ressalta o fato de que, ao considerar a paralelização de uma aplicação, deve-se levar em conta as características do problema ao escolher uma arquitetura paralela a fim de obter um maior ganho de desempenho. Uma estratégia paralela aplicada em uma arquitetura não apropriada pode levar a uma eficiência do paralelismo abaixo do esperado.

Como trabalho futuro, pretende-se explorar abordagens paralelas híbridas para a paralelização de aplicações, combinando as ferramentas aqui apresentadas para extrair paralelismo em diferentes níveis simultaneamente.

RECIT
Revista Eletrônica Científica Inovação e Tecnologia

Exploration of parallelism in multicore, multicomputer and gpu architectures

ABSTRACT

Parallel computing aims to meet the demand for high computational power by simultaneously using more than one processing unit to solve a problem. Several architectures have been proposed to allow for better exploration of parallelism, which are usually followed by programming models and appropriate tools to fully meet the architecture's potential. This work presents the main classifications of parallel systems as well as three tools commonly used for creating parallel applications in different architectures. Moreover, the heat transfer problem was used as a case study for exploiting parallelism with these tools, with the parallel implementations reaching satisfactory results compared to the sequential implementation.

KEYWORDS: parallel computing, OpenMP, MPI, CUDA.

REFERÊNCIAS

BORGES, P.A.P.; PADOIN, E.L. Exemplos de métodos computacionais aplicados a problemas na modelagem matemática. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DO ESTADO DO RIO GRANDE DO SUL, 6., 2006, Ijuí. **Anais...** Porto Alegre: Biblioteca do Instituto de Informática da UFRGS, 2006.

CHAPMAN, B.; JOST, G.; PAS, R.v.d. **Using OpenMP: Portable shared memory parallel programming**. Cambridge, Massachussets: MIT Press, 2008.

MAILLARD, N.; SCHEPKE, C. Programação em Ambientes Computacionais com Múltiplos Níveis de Paralelismo. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DO ESTADO DO RIO GRANDE DO SUL, 8., 2008, Santa Cruz do Sul. **Anais...** Porto Alegre: Biblioteca do Instituto de Informática da UFRGS, 2008. p. 141-142.

NVIDIA. **NVIDIA Kepler GK110 Architecture Whitepaper**. 2012. Disponível em:< https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. Acesso em: 20 ago. 2016.

PACHECO, P. **Parallel Programming with MPI**. Boston: Morgan Kaufmann Publishers Inc, 1997.

SCHEPKE, C. **Ambientes de Programação Paralela**. Porto Alegre, RS: UFRGS, 2009.

SILVA, C. Unidades de Processamento Gráfico – GPUs. In: SILVA, Cleomar. **Programação Genética Maciçamente Paralela em GPUs**. Rio de Janeiro: Pontifícia Universidade Católica do Rio de Janeiro, 2014, p. 47-53.

Recebido: 20 ago. 2016.
Aprovado: 23 nov. 2017.
DOI:
Como citar: Exploração de paralelismo em arquiteturas multicore, multicomputadores e GPU's. R. Eletr. Cient. Inov. Tecnol, Medianeira, v. 8, n. 15, 2017. E – 4535.
Disponível em: <https://periodicos.utfpr.edu.br/recit>. Acesso em: XXX.
Correspondência: