Capítulo

4

Introdução ao Desenvolvimento de Aplicações Paralelas com o Paradigma Orientado a Tarefas e o Runtime StarPU

Lucas Leandro Nesi, Vinícius Garcia Pinto, Marcelo Cogo Miletto, Lucas Mello Schnorr
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Samuel Thibault
LaBRI, STORM, INRIA
Bordeaux, França

Resumo

As novidades e a heterogeneidade de recursos computacionais na área da computação de alto desempenho, como aceleradores GPGPUs, continuam contribuindo para a complexidade da programação paralela. O paradigma orientado a tarefas permite algumas facilidades nesta programação por que transfere para um runtime muitas responsabilidades que seriam anteriormente realizadas pelo programador nos paradigmas tradicionais. Desta forma, é responsabilidade do runtime escalonar as tarefas, gerenciar a memória, e escolher o recurso/implementação à ser utilizada. Para isso, neste paradigma basta o programador definir as tarefas, suas implementações em recursos heterogêneos diferentes, e suas dependências de dados. Neste minicurso, será apresentado uma introdução ao paradigma de programação paralela orientado a tarefas, e como construir programas que executam em recursos heterogêneos utilizando o runtime StarPU. Demonstraremos como programar tarefas com múltiplas implementações, utilizando CPU e GPU. Apresentaremos exemplos de programas como a soma e multiplicação de matrizes e como encontrar o maior elemento de uma lista. Por fim, faremos a introdução a ferramentas e métodos de como analisar o desempenho destes programas, com o emprego do framework StarVZ.

4.1. Introdução

Ao longo dos anos o cenário de HPC passou por diversas transformações drásticas, com o objetivo de fornecer cada vez mais poder computacional. A busca por um maior desempenho levou à larga adoção de sistemas heterogêneos. Estes são frequentemente formados por um conjunto de processadores multicore, aliados a aceleradores específicos, como as GPUs. Ao mesmo tempo em que a heterogeneidade nos traz uma maior capacidade computacional, também tem-se um maior desafio ao construir programas que utilizem dessa capacidade em sua totalidade. Programas paralelos devem considerar aspectos como o balanceamento de carga, custos de comunicação, e a escalabilidade e portabilidade de desempenho, o que se torna ainda mais difícil nestes cenários onde existem vários nós computacionais com diferentes tipos de recursos. A heterogeneidade é vista como um caminho para alcançarmos o patamar da computação Exascale (Dongarra et al., 2017).

Tais mudanças arquiteturais demandam o uso de diferentes estratégias para que se possa explorar, de forma eficiente, os recursos computacionais presentes nos supercomputadores de hoje. Dada a maior complexidade dos sistemas de HPC, o trabalho do programador de desenvolver uma aplicação paralela envolve diversos níveis, tornando-se algo igualmente complexo. Para facilitar este trabalho, determinados paradigmas de programação podem ser usados.

O paradigma orientado a tarefas permite a construção de programas paralelos através dos conceitos de tarefas e dependência de dados (AUGONNET et al., 2011). Uma tarefa é a unidade computacional básica, e contém um trecho de código específico que ela irá executar sobre um conjunto de dados. Desta forma, é possível escrever códigos voltados para diferentes dispositivos que realizam uma mesma operação. Cada tarefa acessa determinadas regiões de dados, sendo que os conflitos de leitura e escrita em uma mesma região de memória causam dependências entre diferentes tarefas. A combinação destes dois conceitos implica em um grafo acíclico dirigido (DAG), onde os vértices são as tarefas e as arestas as dependências. Assim, a aplicação pode ser vista nesta estrutura de grafo.

A grande vantagem de se modelar uma aplicação desta maneira é que ela pode ser tratada por outro componente que faz parte deste paradigma: o sistema de *runtime* (THIBAULT, 2018). Esse sistema encontra-se em uma camada entre a aplicação e as interfaces dos recursos computacionais, tendo uma visão geral da aplicação como um DAG, e a disponibilidade e capacidade dos recursos computacionais. Essa camada facilita uma série de tarefas que antes eram de responsabilidade do programador, como por exemplo o controle dos recursos, o escalonamento das tarefas de forma paralela, e o gerenciamento dos dados entre os diferentes recursos. Isso torna este paradigma uma opção interessante, flexível e versátil para o desenvolvimento de aplicações paralelas.

O restante deste documento está estruturado da seguinte forma. A Seção 4.2 apresenta as noções básicas sobre o Paradigma de Programação Paralela Orientada a Tarefas e os conceitos a serem utilizados. A Seção 4.3 apresenta o *Runtime* StarPU e suas particularidades. A Seção 4.4 mostra passo a passo como construir a primeira aplicação utilizando o StarPU. A Seção 4.5 traz outros exemplos de programas utilizando o StarPU. A Seção 4.6 apresenta uma introdução de como analisar estes programas. A Seção 4.7 conclui este minicurso.

4.2. Paradigma de programação paralela orientado a tarefas

O Paradigma de Programação Paralela Orientado a Tarefas (*task-based programming* ou *data flow scheduling*) é um conceito (BRIAT et al., 1997) que utiliza uma estratégia mais declarativa na programação. Isso permite que um *runtime* (*midleware*) interprete estas declarações e seja responsável por ações como escalonamento e gerenciamento dos dados. Em outros paradigmas tradicionais, estas ações devem ser programadas diretamente pelo desenvolvedor da aplicação paralela. No geral, esta estratégia facilita na programação e reduz a complexidade no desenvolvimento de aplicações paralelas, entretanto, reduz o controle do programador sobre a execução (Dongarra et al., 2017). Este paradigma está se tornando cada vez mais popular desde a década de 2000, quando diferentes projetos começaram a utilizá-lo (THIBAULT, 2018; Dongarra et al., 2017) tendo em vista a heterogeneidade dos sistemas computacionais de alto desempenho.

A estrutura de uma aplicação paralela orientada a tarefas consiste em um conjunto de tarefas e um conjunto de bloco de dados. Cada tarefa, que normalmente é uma região de código sequencial (em CPU), trabalha em alguns destes blocos de dados, aplicando suas operações neles e modificando-os, ou salvando seus resultados em outros blocos de dados. Um bloco de dados é simplesmente uma subdivisão do domínio a ser trabalhado. Se a aplicação trabalha em uma matriz, cada bloco de dados pode ser uma submatriz, e pode ser identificado com uma coordenada, por exemplo. Se o domínio da aplicação é uma lista, cada bloco de dados por ser uma sub lista contínua da posição i até a posição j. Uma tarefa possui uma implementação e pode ser executada várias vezes sobre conjunto de dados diferentes. Como por exemplo, uma tarefa de multiplicar submatrizes de uma matriz, pode ser aplicada em submatrizes de coordenadas diferentes várias vezes. Neste caso, cada tarefa em dados diferentes possui um identificador único, mas o nome da tarefa e a implementação é igual para todos elas. Alguns *runtimes* permitem a submissão de tarefas apenas informado os blocos de dados utilizados. Desta maneira, a interação entre as tarefas ocorre devido a utilização dos mesmos blocos de memória por elas. Assim, se uma tarefa T^A, submetida primeiro, modifica um bloco de dados D^A, e em seguida, uma tarefa T^B, submetida depois de T^A, lê D^A, existe uma dependência implícita entre T^A e T^B. Na maneira que T^A é uma dependência de T^B, e deve ser executada antes de T^B para garantir a coerência da aplicação.

Usualmente, uma aplicação paralela orientada a tarefas pode ser representada por um grafo acíclico dirigido (*Directed Acyclic Graph* – DAG). Neste grafo, os nós são as tarefas, e as arestas as dependências entre elas. Uma aresta dirigida da tarefa T^A para a T^B significa que T^A deve ser executada antes de T^B. Estas dependências podem ser herdadas da reutilização de dados (como explicado anteriormente) ou, em alguns *runtimes,* explicitamente inseridas pelo desenvolvedor.

Discretizar um algoritmo para o paradigma orientado a tarefas nem sempre é um processo trivial, e nem todos os algoritmos são bons candidatos para tal conversão (THIBAULT, 2018). Entretanto, isso não significa que os problemas não admitam algoritmos que utilizem este paradigma de forma eficiente.

Vamos selecionar um problema simples, uma operação de redução em um vetor, para demonstrar a criação de um algoritmo orientado a tarefas e a construção do DAG correspondente. Neste exemplo, vamos procurar o maior valor em um vetor. Sabemos

que podemos dividir a lista em sub listas para procurar o maior elemento em cada uma e depois procurar o maior entre os resultados. Desta maneira, podemos definir uma tarefa como responsável por procurar o maior valor em uma sub lista e retornar o maior valor. Para fins didáticos, vamos considerar uma lista de 27 elementos. Podemos criar uma tarefa que descobre o maior número entre três elementos, vamos chamá-la de `MD3`, *maior dos três*. Em um primeiro momento, podemos aplicar ela nove vezes nesta lista, em nove triplas diferentes, gerando nove resultados. Sabemos que o maior número da lista é um dentre este nove resultados, basta analisar eles novamente, aplicando a mesma tarefa em cada uma das três possíveis triplas resultantes da primeira etapa. A Figura 4.1 demonstra o grafo da execução destas tarefas com o comportamento dos dados da lista de 27 elementos. Todas as tarefas utilizadas são do tipo `MD3` e são aplicadas em triplas contínuas da lista. As tarefas possuem um identificador (`id`) mostrando uma possível ordem de criação delas. Neste exemplo, o maior elemento é o número 99, primeiramente encontrado pela tarefa 6, e então pela tarefa 11 e finalmente pela última tarefa 13. O DAG desta aplicação mostrando as dependências entre as tarefas pode ser visualizado na Figura 4.2.

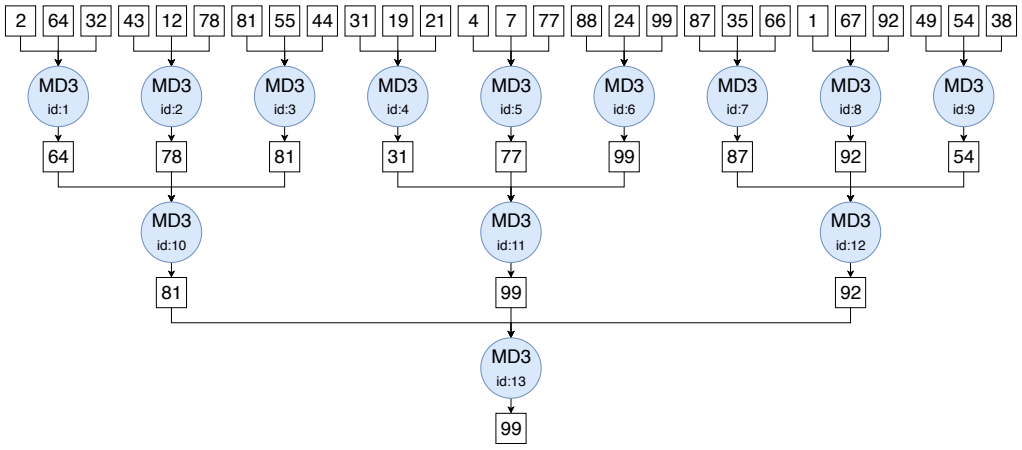

Figura 4.1. Grafo de execução com os dados da aplicação de encontrar o maior elemento em uma lista de 27 elementos utilizando uma tarefa de três entradas chamada de MD3. Fonte: Autores.

O DAG presente na Figura 4.2 exemplifica as oportunidades de paralelismo desta execução. Por exemplo, todas as tarefas do primeiro nível (ids de 1 a 9) são independentes, isso quer dizer, trabalham em dados diferentes e podem ser executadas em paralelo. Ainda, tarefas de níveis diferentes que não possuem dependência, por exemplo, tarefa 1 e tarefa 11 podem também ser executadas em paralelo. Este exemplo mostra os graus de liberdade para possíveis paralelismos. Como a programação é mais declarativa, afinal o programador apenas definiu as tarefas e suas entradas, não foi necessário especificar que tarefa x pode executar ao mesmo tempo com a tarefa y, nem em que momento elas devem executar ou como devem ser mapeadas para os recursos computacionais. Em um exemplo simples como esse, o programador poderia ter descrito manualmente as possibilidades de paralelismo, ou seja, especificar que instruções em quais dados podem ser executadas paralelamente. No entanto, na medida que os problemas se tornam mais complexos, com interações de dados mais robustas, tal processo manual se torna demasiadamente laborioso. Com a especificação e declaração das tarefas pode-se inferir o paralelismo com base

no DAG, e utilizar qualquer combinação dos conjuntos possíveis paralelos de tarefas. Esta inferência pode ser feita durante a execução da aplicação utilizando um *runtime*.

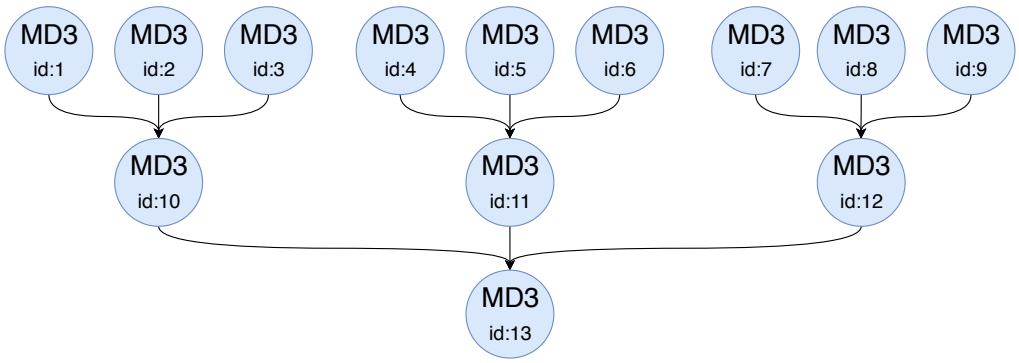

Figura 4.2. DAG da aplicação de encontrar o maior elemento em uma lista de 27 elementos utilizando uma tarefa de três entradas chamada de MD3. Fonte: Autores.

Na computação de alto desempenho, podemos encontrar diversos exemplos de *runtimes*. Duas ferramentas muito populares de programação paralela `OpenMP` (DAGUM; MENON, 1998) e `MPI` (GROPP; LUSK; SKJELLUM, 1999) utilizam runtimes para suas execuções. Isto é, existe uma entidade que é executada juntamente com a aplicação, realizando ações que não foram tomadas diretamente pelo programador. Na programação de aplicações baseadas em tarefas, alguns exemplos de *runtimes* existentes são PaRSEC (BOSILCA et al., 2012), OmpSs (DURAN et al., 2011), OpenMP >4.0(OpenMP, 2013), XKaapi (GAUTIER et al., 2013), e StarPU (AUGONNET et al., 2011). Entre os exemplo citados, o StarPU é um *runtime* que permite a utilização de recursos heterogêneos (CPUs e GPGPUs) e multi-nó com seu módulo StarPU-MPI (AUGONNET et al., 2012).

O *runtime* pode realizar diversas otimizações baseado no DAG. A primeira delas é fazer o mapeamento das tarefas para os recursos, e identificar as oportunidades de paralelismo. Os *runtimes* podem disponibilizar diversos escalonadores para realizar estas operações. Outro exemplo é realizar comunicações em paralelo com as computações, efetuando operações de gerenciamento de memória.

4.3. O runtime StarPU

O StarPU é um *runtime* orientado a tarefas de propósito geral para programação de aplicações paralelas para computadores heterogêneos *multicore*. Ele foi inicialmente desenvolvido por Cédric Augonnet em sua Tese de Doutorado (AUGONNET, 2011) na *Université de Bordeaux*. O StarPU provê uma interface para que aplicações submetam suas tarefas em recursos. A Figura 4.3 apresenta a pilha de software de uma aplicação paralela orientada a tarefas e a localização da aplicação e do *runtime* sobre a arquitetura da máquina.

O StarPU usa o modelo STF (*Sequential Task Flow*) (AGULLO et al., 2016), onde a aplicação pode submeter sequencialmente as tarefas durante a execução e o *runtime* dinamicamente as escalona para os recursos. Neste modelo, o DAG não precisa ser inteiramente conhecido, e as tarefas podem ir sendo submetidas gradativamente, sendo

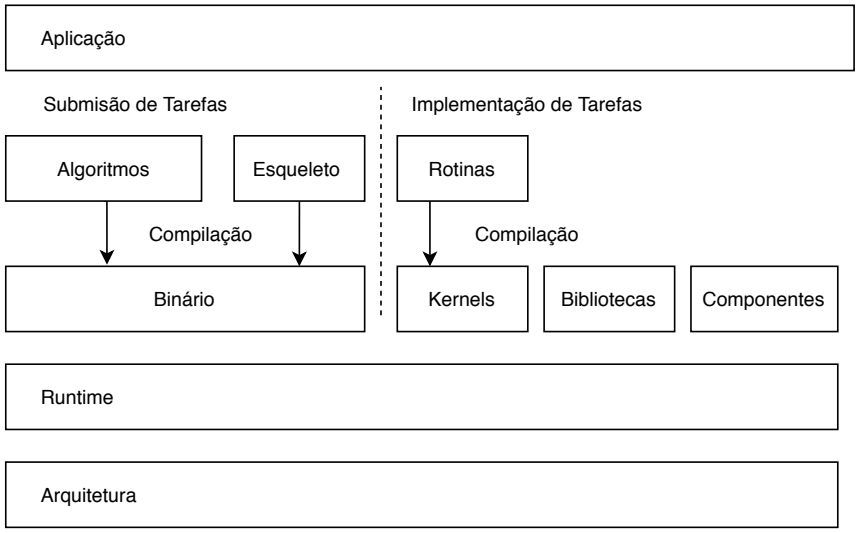

Figura 4.3. *Software stack* Utilizando o runtime StarPU. Fonte: Thibault (2018).

criadas dinamicamente durante a execução. O StarPU associa a cada recurso computacional um trabalhador, sendo que este trabalhador pode executar uma tarefa por vez sem preempção. Por exemplo, cada *core* de um processador e cada GPU é um trabalhador. Ainda, as tarefas podem ter múltiplas implementações (CPU/GPU), e a decisão de qual implementação utilizar é feita pelo *runtime* durante a execução. Isso permite a execução da aplicação em ambientes heterogêneos com uma certa facilidade ao programador.

As dependências entre as tarefas que estruturam o DAG são herdadas da reutilização de dados entre as tarefas. Os blocos de dados no `StarPU` são organizados na estrutura `data_handle`, e devem ser definidos antes da criação das tarefas que os utilizam. Desta maneira, no StarPU, as dependências entre as tarefas são implícitas. Esta característica é mais simples e diferente de outras abordagens como OpenMP-Tasks (v. 3.0) e MPI, onde o programador deve informar as dependências ou comunicações manualmente. No OpenMP 4.0, o programador pode definir as dependências de memória usando a cláusula `depend`.

Para escalonar as tarefas nos trabalhadores, o StarPU pode utilizar diferentes heurísticas de escalonamento. Dependendo da disponibilidade dos trabalhadores e da heurística utilizada, o escalonador pode escolher dinamicamente uma das implementações da tarefa e executá-la. Exemplos de heurísticas de escalonamento são `lws` (local work-stealing), `eager` (deque centralizado), `dm` (deque model) baseado no algoritmo heft (TOPCUOGLU; HARIRI; WU, 2002) onde modelos de desempenho das tarefas são utilizados, `dmda` (deque model data aware) uma versão modificada do `dm` para considerar os dados. O escalonador a ser utilizado pode ser definido pela variável de ambiente `STARPU_SCHED` no momento da execução. Mais informações sobre os escalonadores disponíveis podem ser encontrados na página do StarPU[1].

A análise de desempenho das aplicações em StarPU é possível pela utilização de

[1]<http://starpu.gforge.inria.fr/doc/html/Scheduling.html>

rastros de execução. O StarPU pode ser configurado para gerar estes rastros em formato FxT (DANJEAN; NAMYST; WACRENIER, 2005), que podem ser posteriormente convertidos para outros formatos de arquivo como Pajé (SCHNORR; STEIN; KERGOMMEAUX, 2013) que, por sua vez, pode ser utilizado por diferentes *frameworks* para a análise de desempenho.

4.4. Como construir seu primeiro programa

Esta Seção descreve como construir o primeiro programa utilizando a abordagem orientada a tarefas e o *runtime* StarPU. Primeiramente é descrito como realizar a instalação das bibliotecas necessárias. Após é realizada a elaboração do primeiro programa passo a passo.

4.4.1. Instalação do StarPU e ambiente

A página do StarPU[2] oferece maneiras alternativas de instalação. Para o propósito deste minicurso, utilizaremos o gerenciador de pacotes para supercomputadores `spack`[3]. Para baixar o `spack` é necessário apenas o comando `git` para a clonagem do repositório. Após essa operação, devemos configurar o ambiente carregando um arquivo de configuração. Todos os comandos estão também disponíveis publicamente no *companion*[4]

```
git clone https://github.com/spack/spack.git
source spack/share/spack/setup-env.sh
```

Código 4.1. (bash) Baixar Spack

O `StarPU` pode ser obtido utilizando o gerenciador `spack` pelo repositório do `solverstack`[5], que pode ser adicionado pelos seguintes comandos.

```
git clone https://gitlab.inria.fr/solverstack/spack-repo.git
spack repo add spack-repo/
```

Código 4.2. (bash) Adicionar repositório no Spack

Com essa configuração podemos instalar o `StarPU` utilizando o `spack`. Este gerenciará a instalação de quase todas as dependências necessárias. Pode-se configurar a instalação utilizando algumas variações, como a instalação do `StarPU` com suporte a `CUDA`. Para fins deste minicurso o seguinte comando instala o `StarPU` com os devidos pacotes necessários e gera um diretório com todos os cabeçalhos e bibliotecas.

```
spack install starpu@1.3.1~fast+fxt~mpi+cuda~openmp ^cuda@10.0.130
export STARPU_HOME=$(pwd)/starpu
export PKG_CONFIG_PATH=$STARPU_HOME/pkgconfig:\
       $STARPU_HOME/lib/pkgconfig:$PKG_CONFIG_PATH
export LD_LIBRARY_PATH=$STARPU_HOME/lib:\
       $STARPU_HOME/lib64:$LD_LIBRARY_PATH
spack view soft $STARPU_HOME starpu@1.3.1
```

Código 4.3. (bash) Instalar StarPU e configurar ambiente

[2]<http://starpu.gforge.inria.fr/doc/html/BuildingAndInstallingStarPU.html>
[3]<https://spack.io/>
[4]<https://gitlab.com/lnesi/minicurso-starpu-erad-2020>
[5]<https://gitlab.inria.fr/solverstack/spack-repo>

Podemos utilizar a aplicação `pkg-config` para buscar os cabeçalhos e bibliotecas necessárias para compilar com o GCC um programa escrito com StarPU, assumindo que a variável de ambiente `PKG_CONFIG_PATH` foi devidamente configurada. No caso de uma instalação de StarPU com CUDA, as bibliotecas de CUDA também devem ser especificadas. Podemos compilar o programa `hello world` abaixo com o comando do Código 4.5. Este programa cria uma tarefa que imprime *"Hello World"* e mostra a topologia dos recursos identificados. Nas próximas seções será explicado como construir uma aplicação de somar matrizes.

```c
#include <stdlib.h>
#include <limits.h>
#include <starpu.h>

void func_cpu(void *buffers[], void *args)
{
    printf("Hello World!\n");
}

struct starpu_codelet codelet_world =
{
    .cpu_funcs = { func_cpu },
    .nbuffers = 0,
    .name = "hello_world",
};

int main(){
    starpu_init(NULL);
    starpu_topology_print(stdout);
    starpu_task_insert(&codelet_world, 0);
    starpu_task_wait_for_all();
    starpu_shutdown();
}
```

Código 4.4. (C) Esqueleto de uma aplicação StarPU

```bash
gcc $(pkg-config --cflags starpu-1.3 cuda-10.0) ./hello_world.c\
    $(pkg-config --libs starpu-1.3 cuda-10.0)
```

Código 4.5. (bash) Compilar uma aplicação StarPU

4.4.2. Bases de um programa StarPU

A estrutura de um programa StarPU deve começar com o *include* do cabeçalho da biblioteca do StarPU, `starpu.h`. As funções do StarPU só poderão ser utilizadas após a inicialização da biblioteca, realizada pela função `starpu_init()`. Esta função faz uma verificação no sistema para determinar quantos trabalhadores existem e inicializa as estruturas de dados necessárias. Ao fim do programa, a função `starpu_shutdown()` precisa ser invocada para finalizar o StarPU. Esta chamada é importante porque algumas ações como a gravação dos rastros ocorre somente quando esta função é chamada. Desta maneira, o esqueleto de um programa StarPU pode ser visto no Código 4.6.

```c
#include <starpu.h>
```

```c
int main(){
    starpu_init(NULL);
    //Criar e Submeter Tarefas
    starpu_shutdown();
    return 0;
}
```

Código 4.6. (C) Esqueleto de uma aplicação StarPU

4.4.3. Descritor de funções (*codelet*)

Codelet é uma estrutura do StarPU (`struct starpu_codelet`) que contém as diversas implementações de um kernel. Basicamente uma tarefa tem um *codelet* associado e irá executar uma das implementações previstas. Para definir um *codelet* é necessário definir pelo menos uma implementação, por exemplo, uma implementação em CPU. Para tal, define-se a variável interna `cpu_funcs` com um ponteiro para uma função, por exemplo `func_cpu`. Utilizamos como exemplo a construção de uma tarefa que faz a soma de duas submatrizes A e B, e salva em C. Precisamos definir quantos blocos de dados este *codelet* usa, usando a variável `nbuffers`, por exemplo 3, e os modos de acesso a cada um dos buffer utilizando a variável `modes`. Utilizamos `STARPU_R` para somente leitura, `STARPU_W` para somente escrita, e `STARPU_RW` para leitura e escrita. Definimos esses valores em uma lista. Neste caso, se queremos os dois primeiros buffers como leitura e o terceiro como escrita, utilizamos `{ STARPU_R, STARPU_R, STARPU_W }`. Ainda, opcionalmente, podemos definir um nome para este *codelet*, neste caso `soma_bloco`. O Código 4.7 apresenta a construção completa deste *codelet* (assumindo a existência de uma função `func_cpu` com a implementação da funcionalidade da tarefa.

```c
struct starpu_codelet codelet_soma =
{
    .cpu_funcs = { func_cpu },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W },
    .name = "soma_bloco"
};
```

Código 4.7. (C) Exemplo da declaração de codelet

As implementações das funções dos *codelets* sempre tem a assinatura `void func(void *buffers[], void *args)`. Para acessar os blocos de memória dentro da função é utilizada a macro `STARPU_VECTOR_GET_PTR` (para o nosso caso onde utilizaremos vetores) com o buffer e identificador corretos. Neste caso, como queremos realizar a operação $C = A + B$, criamos três variáveis e utilizamos a macro para acessar os endereços de memória corretos. Após isso realizamos a nossa operação. Após a finalização da tarefa, o StarPU automaticamente irá gerenciar a memória caso necessário (salvar ou mover blocos que foram escritos). Um exemplo desta implementação está presente no Código 4.8.

```c
void func_cpu(void *buffers[], void *args)
{
    float *A = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
    float *B = (float *)STARPU_VECTOR_GET_PTR(buffers[1]);
    float *C = (float *)STARPU_VECTOR_GET_PTR(buffers[2]);
    for(int i=0; i < BLOCK_TOTAL_SIZE; i++){
```

```
        C[i] = A[i] + B[i];
    }
}
```

Código 4.8. (C) Exemplo de implementação de tarefa

4.4.4. Descritor de dados (*data handle*)

Para as tarefas utilizarem os blocos de memória, devemos registrá-los no StarPU na forma de `starpu_data_handle_t`. Basicamente, realizamos a alocação convencional de uma variável (utilizando `malloc`, por exemplo), e depois utilizamos a função `starpu_matrix_data_register` para criar um handle. Podemos então utilizar esta área alocada para inicializar estes dados na RAM.

```
float* matrix_a[NUMBER_BLOCKS];
starpu_data_handle_t matrix_a_handle[NUMBER_BLOCKS];
for(int i=0; i<NUMBER_BLOCKS; i++){
    matrix_a[i]=(float*)malloc(WIDTH * WIDTH * sizeof(float));
    starpu_matrix_data_register(&matrix_a_handle[i], STARPU_MAIN_RAM, (
   uintptr_t)matrix_a[i], WIDTH, WIDTH, WIDTH, sizeof(float));
}
```

Código 4.9. (C) Esqueleto de uma aplicação StarPU

Antes da finalização do starpu, com `starpu_shutdown` é necessário desalocar os `data_handles` utilizando a função `starpu_data_unregister()`. No nosso caso, podemos realizar isso com o seguinte bloco de código.

```
for(int i=0; i < NUMBER_BLOCKS; i++){
    starpu_data_unregister(matrix_a_handle[i]);
    starpu_data_unregister(matrix_b_handle[i]);
    starpu_data_unregister(matrix_c_handle[i]);
}
```

Código 4.10. (C) Esqueleto de uma aplicação StarPU

4.4.5. Tarefa (*Task*)

Finalmente, podemos submeter a tarefa com o *codelet* e os blocos de memória corretos. Neste caso, como queremos realizar a operação de soma de duas matrizes A e B e salvar na matriz C, e como as matrizes estão subdivididas em blocos, devemos definir cada tarefa como realizando a operação $bloco_c[i] = bloco_a[i] + bloco_b[i]$. O Código 4.11 apresenta a submissão da tarefas em todas as submatrizes.

```
for(int i=0; i<NUMBER_BLOCKS; i++){
        starpu_task_insert(&codelet_soma, STARPU_R, matrix_a_handle[i],
   STARPU_R, matrix_b_handle[i], STARPU_W, matrix_c_handle[i], 0);
}
```

Código 4.11. (C) Esqueleto de uma aplicação StarPU

4.4.6. Heterogeneidade e implementações em GPU

Para utilizar a heterogeneidade dos recursos o StarPU permite múltiplas implementações do *codelet*. Para isso, devemos definir a variável `cuda_funcs`, para uma função em

CPU que executará uma chamada de função para um recurso CUDA. Neste exemplo, utilizaremos diretamente a biblioteca `cublas`. Ressalta-se que a função a ser definida em `cuda_funcs` deve ser uma função normal em CPU, que chama um *kernel* ou uma função de uma biblioteca em CUDA. Podemos ainda utilizar a variável do *codelet* `where` para obrigar a execução do *codelet* na GPU. O seguinte *codelet* apresenta as modificações.

```
struct starpu_codelet codelet_soma =
{
    .cpu_funcs = { func_cpu },
    .cuda_funcs = { func_gpu },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W },
    .name = "soma_bloco",
    .where = STARPU_CUDA
};
```

Código 4.12. (C) Esqueleto de uma aplicação StarPU

Um exemplo de função que invoca a biblioteca `cublas` para realizar a soma de matrizes é dada no código abaixo.

```
#include <cuda_runtime.h>
#include <cublas_v2.h>
#define CHECK_CUBLAS(x) if(x!=CUBLAS_STATUS_SUCCESS){printf("Cublass
   error: %d\n", x);};
cublasHandle_t cublas_mainhandle;

void func_gpu(void *buffers[], void *args)
{
    float alpha_beta = 1;
    float *A = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
    float *B = (float *)STARPU_VECTOR_GET_PTR(buffers[1]);
    float *C = (float *)STARPU_VECTOR_GET_PTR(buffers[2]);
    cublasSetStream(cublas_mainhandle, starpu_cuda_get_local_stream());
    CHECK_CUBLAS(cublasSgeam(cublas_mainhandle,
                CUBLAS_OP_N, CUBLAS_OP_N, WIDTH, WIDTH,
                &alpha_beta, A, WIDTH, &alpha_beta,
                B, WIDTH, C, WIDTH));
    cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

Código 4.13. (C) Esqueleto de uma aplicação StarPU

Um detalhe de implementação é que para utilizar a biblioteca `cublas` devemos inicializar ela utilizando a função `cublasCreate();` no main do nosso programa. A compilação agora do programa deve informar a biblioteca `cublas`.

```
gcc $(pkg-config --cflags starpu-1.3 cuda-10.0) \
    ./soma_matrix_cuda.c \
    $(pkg-config --libs starpu-1.3 cudart-10.0 cublas-10.0)
```

Código 4.14. (bash) Compilar uma aplicação StarPU

4.5. Exemplos de aplicações

Apresentamos nesta seção duas aplicações paralelas baseadas em tarefas com o *runtime* StarPU: a clássica multiplicação de matrizes em sua versão com blocos, seguida de um exemplo de como encontrar o maior valor em um vetor. A versão completa de todos estes códigos está disponível publicamente no *companion*[6].

Multiplicação de matrizes

Para realizar a multiplicação de matrizes podemos utilizar a versão do algoritmo baseada em blocos. Sendo k a largura do bloco, o bloco $C[i][j]$ é por:

$$C[i][j] = SUM(A[k][j] \times B[i][k]) \tag{1}$$

Desta forma, cada tarefa pode computar $C = C + A \times B$. Neste exemplo, duas tarefas que atualizem o mesmo C não podem executar ao mesmo tempo. Criando uma série de dependências. A estrutura da aplicação fica igual a anterior com as seguintes diferenças. O código do *codelet* fica conforme listado em 4.15, sendo que a listagem do Código 4.16 apresenta a inserção das tarefas na *thread* sequencial do programa principal (*main*).

```
void block_mm_cpu (void *buffers[], void *args)
{
    float *A = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
    float *B = (float *)STARPU_VECTOR_GET_PTR(buffers[1]);
    float *C = (float *)STARPU_VECTOR_GET_PTR(buffers[2]);

    for(int i=0; i < WIDTH; i++){
       for(int j=0; j < WIDTH; j++){
          for(int k=0; k < WIDTH; k++){
             C[j * WIDTH + i] += A[j * WIDTH + k] * B[k * WIDTH + i];
          }
       }
    }
}

struct starpu_codelet codelet_multi =
{
    .cpu_funcs = { block_mm_cpu },
    .cpu_funcs_name = { "block_mm_cpu" },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W },
    .where = STARPU_CPU,
    .name = "multiplica_por_bloco"
};
```

Código 4.15. (C) Codelet de uma multiplicação de matrizes em blocos.

```
for(int i=0; i<NUMBER_BLOCKS_WIDTH; i++){
    for(int j=0; j<NUMBER_BLOCKS_WIDTH; j++){
        for(int k=0; k<NUMBER_BLOCKS_WIDTH; k++){
```

[6]<https://gitlab.com/lnesi/minicurso-starpu-erad-2020>

```
            starpu_task_insert(&codelet_multi,
                STARPU_R, matrix_a_handle[j * NUMBER_BLOCKS_WIDTH + k],
                STARPU_R, matrix_b_handle[k * NUMBER_BLOCKS_WIDTH + i],
                STARPU_W, matrix_c_handle[j * NUMBER_BLOCKS_WIDTH + i],
                0);
        }
    }
}
```

Código 4.16. (C) Esqueleto de uma aplicação StarPU

Encontrar o maior valor em um vetor

Este problema esta descrito na Seção 4.2. O grande diferencial deles é definir os dados de tal forma a criar corretamente as dependências. Pode-se utilizar algumas abordagens diferentes para criar e particionar os `data_handles`. Iremos assumir que cada tarefa realiza a operação em uma lista de tamanho três. Podemos então realizar a alocação de `data_handles` com tamanho três. Entretanto, as respostas das tarefas são apenas um valor. Caso utilizássemos estes `data_handles` de tamanho três como resultado para múltiplas tarefas, um problema de dependência iria existir. Veja que a tarefa teria permissão de leitura/escrita sobre os dados, então se duas tarefas escrevem sobre os mesmos dados uma dependência existirá para garantir a coerência entre elas. Entretanto, sabemos que cada tarefa só salvaria seu resultado em uma posição específica. Para descrever este comportamento em StarPU, devemos dividir os `data_handles` em sub-`data_handles`. Para isso, criamos um filtro informando como será esta divisão. Como queremos dividir um vetor de tamanho três em blocos de tamanho um, utilizamos o filtro a seguir.

```
struct starpu_data_filter filter_resposta =
{
    .filter_func = starpu_vector_filter_block,
    .nchildren = 3
};
```

Código 4.17. (C) Filtro para divisão de um data_handle.

Devemos então informar a utilização deste filtro para o StarPU com a função `starpu_data_partition()`. A submissão de tarefas segue normalmente com um `data_handle` resultante da função `starpu_data_get_sub_data()` como demonstrado no código a seguir.

```
starpu_data_partition(handles_resposta[reposta_id], &filter_resposta);

starpu_data_handle_t sub_resposta = starpu_data_get_sub_data(
   handles_resposta[reposta_id], 1, i%DIVISAO);

starpu_task_insert(&mycodelet,
                   STARPU_R, handles_entrada[i],
                   STARPU_W, sub_resposta,
                   0);
```

Código 4.18. (C) Dividindo um data_handle em múltiplos e utilizando os sub-data_handles na submissão de tarefas.

4.6. Analisando aplicações StarPU

Esta Seção apresenta métodos para analisar as aplicações StarPU. Primeiramente apresentamos algumas métricas especializadas para avaliar as aplicações. Segundo, iremos apresentar porque estas métricas são insuficientes, demonstrando uma solução com uso do *workflow* StarVZ.

4.6.1. Métricas

As métricas fundamentais para análise de desempenho de aplicações paralelas são o *makespan*, o *speed-up* e a eficiência. Entretanto, em cenários como o apresentado neste minicurso, que incluem recursos de processamento heterogêneos e escalonamento dinâmico, tais métricas se mostram insuficientes (PINTO et al., 2016; PINTO et al., 2018b; PINTO et al., 2018a; NESI et al., 2019).

Na seção a seguir, apresentaremos um breve exemplo de análise de desempenho de uma aplicação implementada com tarefas executando sobre o StarPU. Esta análise será conduzida com auxilio do *framework* StarVZ[7]. Embora a funcionalidade base do StarVZ seja a visualização de rastros, diversos recursos adicionais são disponibilizados para auxiliar o analista a identificar e compreender a origem de gargalos ou anomalias. Entre estes recursos podemos citar a exibição de arestas de dependências entre as tarefas, cálculo de estimativas para o tempo de execução, computação da ociosidade dos trabalhadores e identificação de tarefas com duração anômala. Além disso, por meio de painéis gráficos complementares, podemos adicionar outras informações relevantes como a quantidade de tarefas submetidas, quantidade de tarefas prontas para execução em cada instante e a progressão da computação ao longo do tempo.

4.6.2. Visualizações e StarVZ

A criação de visualizações com o *framework* StarVZ é composta de duas etapas. A etapa inicial consiste em um pré-processamento dos rastros gerados no final da execução da aplicação com StarPU. A segunda etapa, que consiste na análise propriamente dita, é realizada com auxílio do pacote R do StarVZ. Para baixar o *framework* StarVZ e instalar o pacote R, é necessário executar os comandos abaixo.

```
git clone https://github.com/schnorr/starvz.git
apt install -y r-base libxml2-dev libssl-dev \
    libcurl4-openssl-dev libgit2-dev libboost-dev
./starvz/R/install.R
# pajeng
apt install -y git cmake build-essential \
    libboost-dev asciidoc flex bison
git clone git://github.com/schnorr/pajeng.git
mkdir -p pajeng/b ; cd pajeng/b
cmake ..
make
```

Código 4.19. (bash) *Download*, configuração e instalação do StarVZ e suas dependências

Para obter rastros de uma execução com StarPU é necessário definir a variável de

[7]<https://github.com/schnorr/starvz>

ambiente `STARPU_GENERATE_TRACE`. Além disso, é necessário que o StarPU tenha sido compilado com suporte ao FxT (veja Código 4.3). A variável `STARPU_FXT_PREFIX` permite escolher o local onde serão disponibilizados os arquivos contendo os rastros.

```
gcc $(pkg-config --cflags starpu-1.3) ./exemplos/mult_matrix.c \
    $(pkg-config --libs starpu-1.3) -o mult_matrix
STARPU_FXT_PREFIX=$PWD/ STARPU_GENERATE_TRACE=1 ./mult_matrix
```

Código 4.20. (bash) Compilação e Execução de Aplicação `mult_matrix` da seção anterior com rastreamento habilitado

Após a execução serão gerados arquivos com os rastros da aplicação (um ou mais arquivos com nome `prof_file_*`). O pré-processamento destes rastros com StarVZ se dá pela forma abaixo.

```
export PATH=starvz/:$PATH
export PATH=pajeng/b:$PATH
export PATH=$STARPU_HOME/bin:$PATH
./starvz/src/phase1-workflow.sh ./ ""
```

Código 4.21. (bash) Pré-processamento dos rastros com StarVZ

Em seguida, os rastros pré-processados devem ser carregados na linguagem R por meio do pacote do StarVZ, assumindo que os dados encontram-se no diretório corrente. As visualizações produzidas com StarVZ são altamente customizáveis. Dessa forma, devemos utilizar um conjunto básico de parâmetros que permite ativar ou desativar painéis conforme desejado ou conforme as informações disponíveis nos rastros coletados. O comando abaixo carrega uma configuração de base que já está inclusa no pacote StarVZ baixado anteriormente. Em seguida, devemos escolher quais dados queremos adicionar ou remover da visualização. Em seguida, podemos chamar a função `the_master_function()` que cria o gráfico. O exemplo completo em R está no Código 4.22 e um exemplo de resultado está presente na Figura 4.4.

```
library(starvz)
dtrace <- the_fast_reader_function("./")
pajer <- config::get(file = "starvz/full_config.yaml")

# desativa visualizacao dos estados internos do StarPU
pajer$starpu$active = TRUE
# habilita curvas de tarefas submetidas e ativas
pajer$submitted$active = TRUE
pajer$st$abe$active = TRUE

# seleciona tarefas para desenhar as dependencias
pajer$st$tasks$active = TRUE
pajer$st$tasks$levels = 5
pajer$st$tasks$list = dtrace$State %>% group_by(X, Y) %>%
     filter(End==max(End), !is.na(JobId)) %>% .$JobId
the_master_function(dtrace)
```

Código 4.22. (R) Geração de visualizações com StarVZ

A visualização presente na Figura 4.4 é composta de quatro painéis. O painel superior apresenta a visualização espaço-tempo da aplicação, onde o eixo vertical *y* representa os recursos de processamentos que, neste caso representam os *cores* do processador

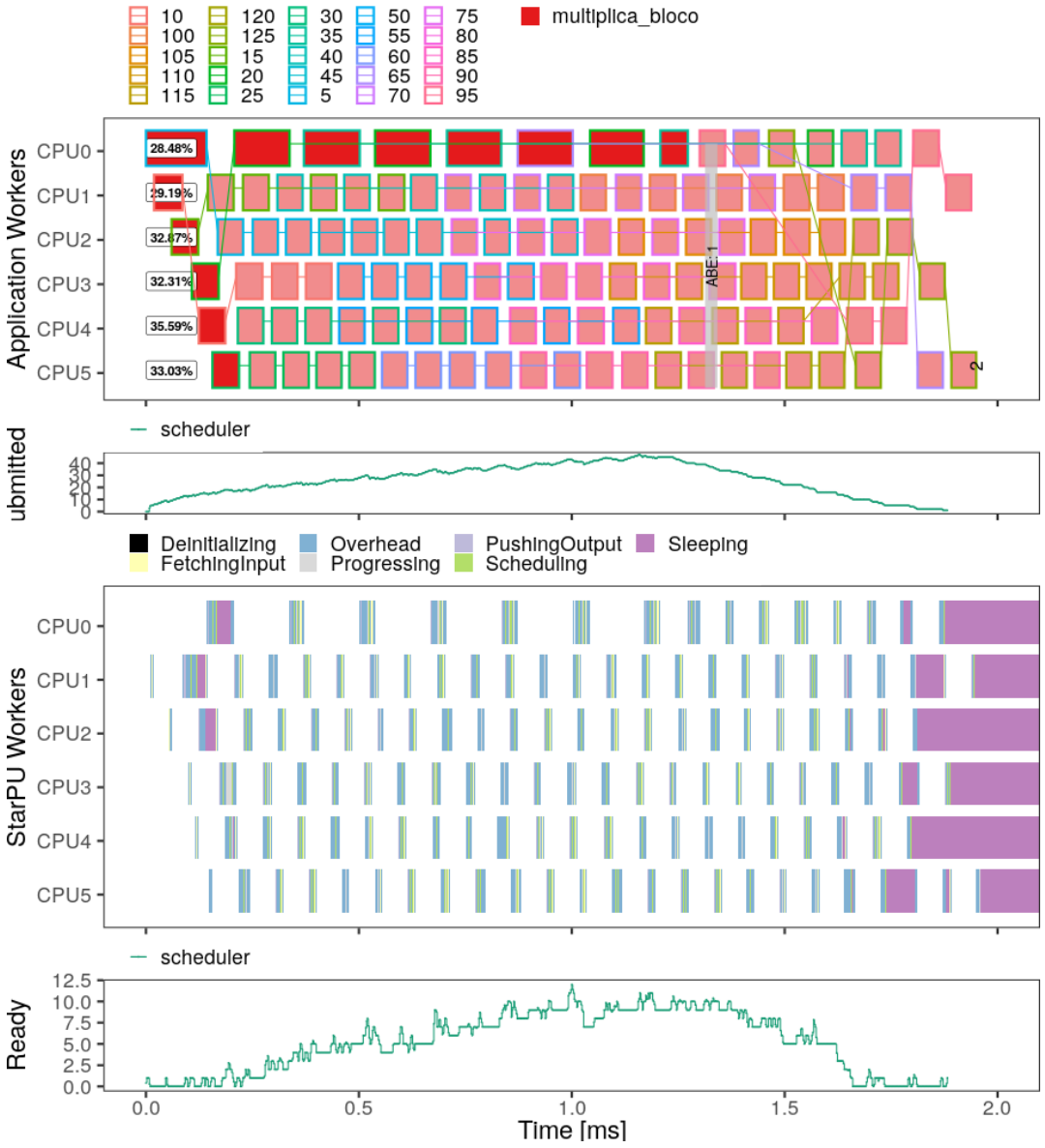

Figura 4.4. Exemplo de visualização produzida com o *framework* StarVZ para a aplicação `mult_matrix`. Fonte: Autores.

e o eixo horizontal x representa o tempo de execução (duração das tarefas). O eixo x deste e dos demais painéis é mantido sincronizado de forma a facilitar a correlação entre os diferentes painéis. O segundo painel apresenta o número de tarefas submetidas ao longo do tempo (eixo y). O terceiro painel apresenta a visualização espaço-tempo do *runtime* StarPU de forma análoga ao primeiro painel. O último painel apresenta o número de tarefas prontas para execução ao longo do tempo (eixo y).

A análise do primeiro painel permite observar que a ociosidade dos trabalhadores é consideravelmente elevada, oscilando em torno de 30%. Inicialmente, poderíamos supor que a ociosidade decorre da falta de paralelismo. Entretanto, a correlação com o

último painel mostra que, ao menos em parte do tempo de execução, o número de tarefas prontas para execução se mantém elevado o suficiente para ocupar todos os trabalhadores. Descartada esta primeira suposição, podemos partir para a análise do diagrama espaço-tempo do *runtime*. Vemos que os períodos de ociosidade no processamento das tarefas da aplicação coincidem com a execução de atividades de gerenciamento do próprio StarPU. Em geral, o *overhead* gerado pelo StarPU costuma ser pouco significante no tempo total de execução da aplicação, porém, neste exemplo, a curta duração das tarefas (em média 0.059 ms) faz com que ele se torne perceptível. A taxa de ociosidade elevada também pode explicar a lacuna entre o limite teórico calculado pelo StarVZ (barra vertical cinza em 1,33 ms) e o *makespan* observado.

O diagrama espaço-tempo da aplicação mostra também que a duração de algumas tarefas, em especial aquelas executadas pelo trabalhador CPU0, foi considerada como anômala (*outlier*) pelo StarVZ. Estas tarefas são representadas pela cor vermelha em destaque em oposição ao vermelho esmaecido usado para colorir as demais tarefas de mesmo tipo. Ao comparar este diagrama com o gráfico de tarefas submetidas (segundo painel), podemos perceber que a ocorrência destes *outliers* na CPU0 coincide com o crescimento da curva de tarefas submetidas. Após a estagnação do crescimento do número de tarefas submetidas (em torno do instante 1,25 ms), a duração média das tarefas executadas pela CPU0 se aproxima daquelas executadas nos demais trabalhadores. Isto nos permite supor que a *thread* principal da aplicação (`main`) está alocada no mesmo *core* físico do trabalhador CPU0. A solução para tal conflito reside no uso da variável de ambiente `STARPU_MAIN_THREAD_BIND`, dedicando um *core* específico para a submissão das tarefas.

4.7. Conclusão

Este minicurso tem por objetivo apresentar como se programar aplicações paralelas utilizando o paradigma de programação orientado a tarefas. Este paradigma de programação tem se tornado cada vez mais útil tendo em vista que com ele a complexidade da programação de recursos heterogêneos se torna menor. O paradigma orientado a tarefas permite algumas facilidades nesta programação por que transfere para um ambiente de execução (*runtime*) muitas responsabilidades que seriam anteriormente realizadas pelo programador nos paradigmas tradicionais. Espera-se que o texto deste minicurso traga os conceitos básicos iniciais para se permitir a programação utilizando o paradigma orientado a tarefas. Para ir além do que é apresentado neste minicurso, sugerirmos consultar a vasta documentação, tutoriais e minicursos disponíveis no site do StarPU: <http://starpu.gforge.inria.fr/tutorials/>.

Agradecimentos

Este trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001, do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), e dos projetos: FAPERGS ReDaS (19/711-6), MultiGPU (16/354-8) e GreenCloud (16/488-9), do projeto CNPq 447311/2014-0, do projeto CAPES/Brafitec 182/15 e CAPES/Cofecub 899/18, e com apoio do projeto Petrobras (2018/00263-5).

Referências

AGULLO, E. et al. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Tr. Math. Softw.*, ACM, New York, NY, USA, v. 43, n. 2, 2016. ISSN 0098-3500.

AUGONNET, C. *Scheduling Tasks over Multicore machines enhanced with acelerators: a Runtime System's Perspective*. Tese (Theses) — Université Bordeaux 1, dez. 2011. Disponível em: <https://tel.archives-ouvertes.fr/tel-00777154>.

AUGONNET, C. et al. Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In: TRÄFF, J. L.; BENKNER, S.; DONGARRA, J. J. (Ed.). *Recent Advances in the Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 298–299. ISBN 978-3-642-33518-1.

AUGONNET, C. et al. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Conc. Comp.: Pract. Exp., SI:EuroPar 2009*, John Wiley and Sons, Ltd., v. 23, 2011.

BOSILCA, G. et al. Dague: A generic distributed dag engine for high performance computing. *Parallel Computing*, Elsevier, v. 38, n. 1-2, p. 37–51, 2012.

BRIAT, J. et al. Athapascan runtime: Efficiency for irregular problems. In: SPRINGER. *European Conference on Parallel Processing*. [S.l.], 1997. p. 591–600.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, IEEE, v. 5, n. 1, p. 46–55, 1998.

DANJEAN, V.; NAMYST, R.; WACRENIER, P.-A. An efficient multi-level trace toolkit for multi-threaded applications. In: SPRINGER. *European Conference on Parallel Processing*. [S.l.], 2005. p. 166–175.

Dongarra, J. et al. With extreme computing, the rules have changed. *Computing in Science Engineering*, v. 19, n. 3, p. 52–62, May 2017. ISSN 1521-9615.

DURAN, A. et al. Ompss: a proposal for programming heterogeneous multi-core architectures. *Paral. Proces. Letters*, World Scientific, v. 21, n. 02, 2011.

GAUTIER, T. et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: *IEEE Intl. Symposium on Parallel and Distributed Processing*. [S.l.: s.n.], 2013. ISSN 1530-2075.

GROPP, W.; LUSK, E.; SKJELLUM, A. *Using MPI: portable parallel programming with the message-passing interface*. [S.l.]: MIT press, 1999. v. 1.

NESI, L. L. et al. Visual performance analysis of memory behavior in a task-based runtime on hybrid platforms. In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019. p. 142–151. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/CCGRID.2019.00025>.

OpenMP. *OpenMP Application Program Interface Version 4*. OpenMP Architecture Review Board, 2013. Disponível em: <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf>.

PINTO, V. G. et al. Detecção de Anomalias de Desempenho em Aplicações de Alto Desempenho baseadas em Tarefas em Clusters Híbridos. In: *WPerformance 2018 - 17º Workshop em Desempenho de Sistemas Computacionais e de Comunicação*. Natal, Brazil: [s.n.], 2018. p. 1–14. Disponível em: <https://hal.inria.fr/hal-01842038>.

PINTO, V. G. et al. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. *Concurrency and Computation: Practice and Experience*, v. 30, n. 18, p. e4472, 2018. E4472 cpe.4472. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4472>.

PINTO, V. G. et al. Analyzing dynamic task-based applications on hybrid platforms: An agile scripting approach. In: *Third Workshop on Visual Performance Analysis, VPA@SC 2016, Salt Lake, UT, USA, November 18, 2016*. [s.n.], 2016. p. 17–24. Held in conjunction with SC16. Disponível em: <https://doi.org/10.1109/VPA.2016.008>.

SCHNORR, L.; STEIN, B. de O.; KERGOMMEAUX, J. C. de. Paje trace file format, version 1.2.5. *Laboratoire d'Informatique de Grenoble, France, Technical Report*, 2013.

THIBAULT, S. *On Runtime Systems for Task-based Programming on Heterogeneous Platforms*. Tese (Habilitation à diriger des recherches) — Université de Bordeaux, dez. 2018. Disponível em: <https://hal.inria.fr/tel-01959127>.

TOPCUOGLU, H.; HARIRI, S.; WU, M.-y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, IEEE, v. 13, n. 3, p. 260–274, 2002.