Universidade de São Paulo
Instituto de Matemática e Estatística

Aluno: Edênis Freindorfer Azevedo
Professor: Alfredo Goldman vel Lejbman
Monitor: Marcos Amarís González

StarPU

MAC5742: Computação
Paralela e Distribuída

Resumo

Com o crescente interesse e desenvolvimento do ecossistema da Computação de Alta Performance (do termo, em inglês, *High-Performance Computing*), máquinas com múltiplos núcleos, coprocessadores especializados e placas de vídeo com enorme capacidade de processamento são cada vez mais populares entre entusiastas da tecnologia e indústrias de produção e design de *hardware*. Apesar de consideráveis avanços e esforços para distribuir a computação eficientemente em partes nos paradigmas apropriados, sendo o tipo de abordagem mais comum executar uma aplicação em processadores comuns enquanto partes pré-determinadas de cálculos são repassadas para GPUs, projetar um modelo de execução que unifica todas as entidades computacionais e os diferentes tipos de memórias envolvidos permanece um complicado desafio. Arquiteturas híbridas provêm uma substâncial melhora da eficiência energética, logo estão propensas a se popularizem na era de múltiplos núcleos. No entanto, a complexida apresentada por estas arquiteturas têm impacto direto na programação, então torna-se crucial prover uma camada de abstração para desfrutar do verdadeiro potencial de máquinas projetadas por este modelo.

O objetivo deste trabalho é apresentar um sistema de tempo de execução que provê, em alto nível, um modelo fortemente associado a uma expressiva biblioteca de gerenciamento de memórias e dados, denominada *StarPU*. Seu principal objetivo consiste em oferecer múltiplos designs de kernel com um mecanismo conveniente de gerar tarefas para execução paralela em uma arquitetura heterogênea, além de permitir o desenvolvimento de algoritmos de escalonamentos personalizados.

Sumário

1 Introdução

Como resultado do desenvolvimento das arquiteturas de *hardware*, existe uma grande responsabilidade atribuída ao *software* de cada aplicação. Ao propor mecanismos mais simples para uma unidade computacional, os programadores que devem explicitamente cuidar de aspectos que antes eram responsabilidade do *hardware*, como coerência de cache. Desta forma, a distância entre modelos de programação atuais e a evolução do *hardware* cresce de tal maneira que o programador não consegue lidar com as complexidades envolvidas sem auxílio de bibliotecas adicionais.

Deste contexto, os requisitos abaixo são desejáveis para uma bilioteca para arquiteturas híbridas.

- **Visão unificada das unidades computacionais.** A maior parte do desenvolvimento oferece mecanismos cada vez mais simples de como levar a computação para as *GPU*s. Apesar destes esforços, a única maneira de se conseguir desempenho máximo de uma arquitetura híbrida é espalhar a computação por toda a máquina. Um modelo portátil deve prover uma abstração para todas as unidades computacionais, incluindo *CPU*s.

- **Estrutura de uma aplicação paralela.** Programadores ciêntificos dificilmente conseguem reescrever todos seus códigos para cada inovação de *hardware*. Logo, programadores precisam de uma interface portátil para algoritmos paralelos de forma que possam ser executados em qualquer tipo de máquina paralela com ou sem *GPU*s.

- **Escalonamento dinâmico de tarefas.** Fazer a divisão de tarefas entre as diferentes unidades computacionais normalmente implica no entendimento de programação paralela e aspectos baixo nível do *hardware* associado, o que não é compátivel com o princípio de portabilidade. O escalonamento dinâmico de tarefas dentro do sistema de execução abstraí o programador desta tarefa, e assim é possível obter desempenho portátil. Como não existe uma estratégia de escalonamento que otimiza todo os algoritmo paralelos, o sistema de execução deve prover um mecanismo conveniente para utilizar escalonamentos personalizados.

- **Gerenciamento de dados e memória feito pelo sistema de execução.** O design escalável de arquiteturas de múltiplos núcleos normalmente considera atribuir ao programador assegurar a consistência da memória. Como resultado, modelos de programação para *GPU*s

implicam em fazer transferências explicítas entre a memória global e as memórias locais. Estas operações são feitas com o uso de mecanismos de baixo nível, fortemente associado à arquitetura. Aplicações portáteis devem deferir estas transferências para software de níveis mais baixo, como o sistema de execução que deve assegurar as coerência e disponibilidade dos dados em toda a máquina. Devido ao custo do gerenciamento dos dados no desempenho da aplicação, o escalonador de tarefas deve estar ligeiramente associado a este gerenciador.

- **Propor uma interface tão completa quanto possível.** A interface da biblioteca deve oferecer aos programadores a possibilidade de personalizar o escalonamento de tarefas, assim o sistema de execução não precisa adivinhar o que os programadores sabem perfeitamente. Por outro lado, o sistema deve prover uma camada de *software* de alto nível para *feedback* de desempenho.

- **Diminuir a distância entre os diferentes tipos de bibliotecas para programação paralela.** Designers de bibliotecas de programação são especialistas de domínio, mas não necessariamente especialistas de programação paralela. Desta maneira, abstrações de sistema de execução devem auxiliar no processo de criação de um compilador paralelo para *GPU*s. Assim, confiar em uma camada de abstração de um sistema de execução permite aos designers de bibliotecas de programação e compilador paralelas se concentrarem nos algoritmos e gerar código otimizado para execução das funções *kernel* ao invés de lidar com problemas de baixo nível.

Satisfazendo todos estes requisitos, o sistema de execução *StarPU* foi implementado como uma biblioteca *open source* em *C* composta por mais de 60 mil linhas de código.

1.1 Observações

Esta parte de introdução foi fortemente baseada na seção 1 do artigo de Augonnet, Cédric e Namyst, Raymond [1].

2 Um mecanismo para explorar arquiteturas híbridas

Idealizar um projeto de sistema de execução para máquinas *multicore* heterogêneas, com diferentes *GPU*s e unidades computacionais, introduz de-

safios consideráveis. Arquiteturas *multicore* homogêneas de memória compartilhada são programadas com linguages de alto nível, como *OpenMP*, ou biliotecas de baixo nível, como *pthread*. Nos dois casos, o sistema de execução associado é composto simplesmente por um escalonador de tarefas. Em contraste, unidades heterogêneas necessitam de um sistema muito mais complexo, pois não é possível prover acesso a memória global, nem mesmo compartilhada. Diferentemente de processadores usuais, que acessam toda a memória global transparentemente, *GPU*s normalmente possuem uma memória local, onde realizam todas as suas operações. Sem um sistema de execução apropriado, programadores têm que explicitamente assegurar a consistência e coerência da memória entre as diversas *GPU*s, o que compromete a programabilidade da aplicação.

Cada tecnologia de programação para *GPU*s tem seu próprio modelo de execução (*CUDA* para placas *NVIDIA*, por exemplo), e sua própria interface para manipulação e transferência de dados. Assim, adaptar uma aplicação para uma nova plataforma consiste em refazer uma parte consideravelmente grande do código. Este problema é agravado quando a aplicação precisa aproveitar múltiplas *GPU*s, possivelmente de modelos diferentes.

Para resolver estas e outras questões, surge o *StarPU*, uma camada de abstração implementada como um sistema de execução entre o programador e as unidades de processamento computacionais. Aplicações já existentes que utilizem bibliotecas de *High-Performance Computing* ou similar podem ser usadas sob a *StarPU* para que possam usufruir de diferentes *GPU*s e *CPU*s com esforço mínimo.

A seguir serão apresentados os principais componentes da *StarPU*.

2.1 Gerenciamento de dados

Como *GPU*s e *CPU*s não podem acessar a memória um do outro transparentemente, realizar cálculos nas arquiteturas anteriores implica em mover os dados explicitamente de um para o outro. Assim, a *StarPU* realiza estas operações automaticamente utilizando um protocolo de cache *MSI*[2] para minimizar o número de transferências, assim como particionando funções e usando heurísticas para superar o limite de pouco memória disponível nas *GPU*s.

2.2 Modelo de execução amigável para *GPU*s

A diversidade de tecnologias disponíveis torna a programação em *GPU*s fortemente associada a sua arquiteturas. Logo a *StarPU* provê uma abstração de tarefa que possa ser executada em *CPU*s ou em *GPU*s assíncrona-

mente. Programadores podem implementar uma tarefa em múltiplas linguagens (*CUDA*) ou utilizando bibliotecas (*BLAS*) disponíveis na arquitetura e a abstração é o conjunto de todas estas implementações. Do ponto de vista da programação, *StarPU* é a respondável por executar estas tarefas. A aplicação não precisa considerar o problema de distribuição de carga (do termo, em inglês, *load balancing*).

A seguir serão apresentadas algumas características relativo à *GPU*s.

- **Declarando tarefas e dependências de dados.** A estrutura de uma tarefa *StarPU* inclui uma descrição de alto nível sobre cada dado manipulado por cada tarefa e seu tipo de acesso (leitura, escrita, leitura e escrita). Também é possível declarar dependências entre tarefas, deste modo *StarPU* não apenas garante a consistência e integridade dos dados graças a uma biblioteca de manipulação de alto nível, mas também permite aos programadores expressarem grafos complexos de dependência sem grandes esforços.

- **Oferecendo suporte a vários *hardwares*** O modelo de tarefas adotado é suficientemente poderoso para lidar com diversas *CPU*s. Enquanto o núcleo da *StarPU* lida com as transferências de dados independente de *hardware*, suportar uma nova arquitetura é tão simples quanto possível. Primeiro, é necessário que um *driver* provenha de funções de transferência de memória e dados entre o *host* e a arquitetura. Então é necessário um método que, de fato, execute a operação ou tarefa tipicamente pela interface oferecida pelo *driver*. Este modelo foi implementado com sucesso pelo processador *CELL* e *GPU*s *CUDA* [3].

2.3 Escalonador genérico para arquiteturas híbridas

Na listagem anterior não foi explicado como distribuir tarefas eficientemente, especialmente considerando fatores como *load balancing*. Com as arquiteturas cada vez mais complexas, é improvável que um código portátil que distribui tarefas eficientemente de modo estático seja possível ou até mesmo produtivo.

2.4 Escalonador heterogêneo

Transferência de dados têm um grande impacto no desempenho da aplicação, assim um escalonador que favorece a localidade pode melhorar os benefícios de técnicas de cache pela reutilização de dados. Considerando que múltiplos problemas podem ser resolvidos paralelamente, e que as máquinas

podem não ser totalmente dedicadas, escalonamento dinâmico se faz necessário. No contexto de plataforma heterogêneas, o desempenho pode variar de acordo com a arquitetura e a carga de trabalho. Portanto, é crucial considerar as especificidades de cada computador ao determinar quais tarefas devem ser feitas por ele.

Similarmente ao problema das transferências de dados, a heterogeneidade torna o design e a implementação de um escalonador e suas políticas um grande desafio. Assim, a *StarPU* oferece também em sua interface a possibilidade de construção de um escalonador personalizado. Por meio de mecanismos oferecidos de baixo nível, *StarPU* permite aos programadores os utilizarem em alto nível independente da arquitetura. Como todas as políticas de escalonamento implementam a mesma interface, elas podem ser programadas independente das aplicações, e o usuário pode selecionar a política mais apropriada em tempo de execução.

Neste modelo, cada recurso computacional (do termo, em inglês, *worker*) recebe uma fila abstrata de tarefas. Apenas duas operações podem ser aplicadas nesta fila: envio de tarefa ($PUSH$) e execução de uma tarefa (POP). A fila de verdade pode ser compartilhada entre diversos recursos computacionais, contanto que sua implementação seja segura em um contexto de acessos paralelos, sendo transparente para os *drivers*. Todas as decisões de escalonamento são feitas no contexto destas funções, mas não há prevenção de chamadas sob outras circunstâncias ou periodicamente.

Essencialmente, definir uma política de escalonamento consiste em criar um conjunto de filas e associar as mesmas aos recursos computacionais da máquina. Vários designs podem ser usados para a implementação de uma fila, e filas podem ser organizadas em diversos tipos de topologias. As diferenças entre estratégias resultam no jeito como uma fila é escolhida quando deseja-se executar uma nova tarefa.

2.5 Política do escalonador heterogêneo

Como pode ser esperado, a escolha da política de escalonamento está fortemente associada a aplicação e aos dados do problema. Assim, alguns estudos, como o de Augunnet et al. [2], mostram que a política escolhida tem impacto no desempenho da aplicação. De maneira similar, em um outro artigo, Augunnet et al. [4] demonstra como a escolha da política afeta o desempenho, por meio da definição e construção de modelos de desempenho baseado em funções de *hash*.

2.6 Extensão da linguagem *C*

A *StarPU* escalona tarefas passadas entre as unidades de processamento disponíveis. Uma mesma tarefa pode ter diferentes implementações, cada uma especializada para um tipo de arquitetura.

Conceitualmente, tarefas são funções com parâmetros escalares e *buffers*. *Buffer* denota um conjunto de grandes pedaços de dados que podem necessitar ser transferidos entre as memórias das *CPU*s e *GPU*s e têm um modo de acesso (leitura, escrita, leitura e escrita) por cada tarefa. Estes modos de acesso, assim como a sequência em que as tarefas são executadas permitem que a *StarPU* determine o grafo de precedência de tarefas.

A interface de programação *C* pode ser usada da seguinte maneira. Primeiro, é preciso definir uma estrutura *starpu_ codelet*. Ela descreve a tarefa, suas implementações e seus parâmetros.

```
1  void scale_vector_cpu (void *buffers[], void *arg);
2  void scale_vector_opencl (void *buffers[], void *arg);
3
4  static struct starpu_codelet scale_vector_codelet =
5  {
6     .cpu_funcs = { scale_vector_cpu, NULL },
7     .opencl_funcs = { scale_vector_opencl, NULL },
8     .nbuffers = 1,
9     .modes = { STARPU_RW },
10    .name = "scale_vector"
11 }
```

O código acima descreve uma tarefa com duas implementações: uma com *CPU* e outra com *OpenCL*. Esta tarefa tem um parâmetro, que é um vetor de leitura e escrita. A implementação com *CPU* é da seguinte maneira:

```
1  void scale_vector_cpu (void *buffers[], void *arg)
2  {
3     float *factor = arg;
4     starpu_vector_interface_t *vector = buffers[0];
5     unsigned n = STARPU_VECTOR_GET_NX (vector);
6     float *val = (float *) STARPU_VECTOR_GET_PTR (vector);
7
8     for (unsigned i = 0; i < n; i++)
9        val[i] *= *factor;
10 }
```

O código acima converte o parâmetro escalar para o tipo *float* e o ponteiro sem tipo para o verdadeiro vetor de *float* que a tarefa espera. A computação segue nas linhas seguintes, utilizando o vetor diretamente.

A implementação em *OpenCL* seria parecida com a apresentada abaixo.

```
1  void scale_vector_opencl (void *buffers[], void *arg)
2  {
3     /* ... */
4     err = starpu_opencl_load_kernel (&kernel, &queue, &cl_programs, "scale_vector_opencl",
          devid);
5     err = clSetKernelArg (kernel, 0, sizeof(val), &val);
6     err |= clSetKernelArg (kernel, 1, sizeof(size), &size);
7     /* ... */
8     err = clEnqueueNDRangeKernel (queue, kernel, 1, NULL, &global, &local, 0, NULL, &event);
9     /* ... */
10    clFinish (queue);
11    /* ... */
12 }
```

O código acima serve como esqueleto para funções *OpenCL*. Todas as etapas constituem uma tarefa, como definido pela *StarPU*.

Para definição e execução da tarefa, são necessários dois passos: alocar os dados interessantes em estruturas *StarPU* e executar, de fato, a tarefa. O código abaixo, no mesmo contexto das funções apresentadas acima, exemplifica os passos.

```
1  /**** Passo 1 ****/
2  /* Estrutura StarPU. */
3  starpu_data_handle_t vector_handle;
4  /* Colocando o vetor dentro da estrutura StarPU. */
5  starpu_vector_data_register (&vector_handle, 0, vector, NX, sizeof(vector[0]));
6
7  float factor = 3.14;
8
9  /**** Passo 2 ****/
10 starpu_insert_task (&scale_vector_codelet, STARPU_VALUE, &factor, sizeof(factor),
       STARPU_RW, vector_handle, 0);
11
12 /* ... */
13 /* Esperando todas as tarefas serem executadas. */
14 starpu_task_wait_for_all ();
15 /* Desalocando da estrutura StarPU. */
16 starpu_data_unregister (vector_handle);
```

Como pode ser observado, a *API* de *C* não consegue espressar de forma clara os conceitos associados ao modelo de programação da *StarPU*. Além disso, os argumentos esperados pelas funções devem ser consistentes, pois caso não sejam o resultado da operação terá comportamento indefinido. A *API* também força programadores a lidarem com estruturas internas *StarPU*, que somente interessa à própria *StarPU*.

2.7 Plugin para *GCC*

Com as dificuldades citadas anteriormente, foi desenvolvido um plugin para o *GCC* com o objetivo de facilitar o uso da biblioteca por meio de um conceito definido como anotações, extremamente similar ao apresentado por outra biblioteca de programação paralela, *OpenMP*. As principais estruturas são as apresentadas no código a seguir.

```
1  /* Declaracao de uma tarefa. */
2  void scale_vector_cpu (int size, float vector[size],float factor)__attribute__
       ((task_implementation ("cpu",scale_vector)));
3  /* ... */
4  /* Alocando os dados de "vector" em uma estrutura StarPU. */
5  #pragma starpu register vector
6  /* ... */
7  scale_vector_cpu (size, vector, factor);
8  /* Aguardar a execucao de tarefas StarPU. */
9  #pragma starpu wait
10 /* ... */
11 /* Pegue o vetor da estrutura StarPU para a memoria principal. */
12 #pragma starpu acquire vector
```

O próximo trecho apresenta um programa "*Hello World*", utilizando o plugin *StarPU*.

```
1  #include <stdio.h>
2
3  /* Tarefa. */
4  static void my_task (int x) __attribute__ ((task));
5
6  /*  Implementacao da tarefa. */
7  static void my_task (int x)
8  {
9     printf ("Hello, world! x = %d\n", x);
10 }
11
12 int main ()
13 {
14    /* Inicialize StarPU. */
15    #pragma starpu initialize
16    /* Execute a funcao. */
17    my_task (42);
18    /* Espere terminar. */
19    #pragma starpu wait
20    /* Encerre StarPU. */
21    #pragma starpu shutdown
22    return 0;
23 }
```

Este código é compilado com o comando "*$ gcc ‘pkg-config starpu-1.2 –cflags‘ hello-starpu.c -fplugin=‘pkg-config starpu-1.2 –variable=gccplugin‘ ‘pkg-config starpu-1.2 –libs‘*".

2.8 Observações

Esta parte de introdução foi fortemente baseada no artigo de Augonnet, Cédric e Namyst, Raymond [2].

Mais exemplos são apresentados e explicados no manual oficial da *StarPU*, dispoível em: `http://starpu.gforge.inria.fr/doc/starpu.pdf` ou *online* em `http://starpu.gforge.inria.fr/doc/html/`.

Os exemplos foram baseados nos apresentados por Ludovic [5], que também estão disponíveis nos tutoriais apresentados acima.

3 Aplicações

A *StarPU* já foi tema de inúmeros artigos científicos e utilizada para testes, tanto teóricos quanto práticos. Destes, destacam-se:

- **A linguagem *SOCL***. Uma implementação de *OpenCL* que simplifica e aprimora a experiência de programação em arquiteturas híbridas [6]. *SOCL* permite que as aplicações distribuam dinamicamente *kernels* computacionais entre as unidades de processamento para maximizar sua utilização.

- **Biblioteca de álgebra linear *PaStiX***. Um estudo de comparação entre a *StarPU* e outro sistema de execução para arquiteturas heterogêneas, *PaRSEC*, foi conduzido com a biblioteca *PaStiX*. *StarPU* apresentou bons resultados e é utilizado pela *PaStiX* desde sua versão 5.2.1 [7].

- **Fatoração *Cholesky***. Implementações desta fatoração que aproveitem arquiteturas híbridas com um sistema de execução cujo escalonador seja dinâmico podem chegar a 900 $Gflop/s$ de desempenho [8].

- **Sistema de execução *XcalableMP-dev/StarPU***. Uma linguagem para programação em arquiteturas heterogêneas com controle de tamanho de tarefa para ser alocado a estes recursos heterogêneos dinamicamente durante a execução da aplicação. Seu desempenho chega a ser até 40% melhor do que arquiteturas *GPU*s-exclusiva [9].

- **Novo modelo de programação**. Um modelo baseado em programação paralela funcional que trata algumas desvantagens de baixo nível e oferece a possibilidade de fazer transformações sob o grafo de precedência das tarefas em tempo de compilação e execução [10].

- **Aplicações *FEM* em *GPU*s**. Aplicações de método de elementos finitos (do termo, em inglês, *Finite Element Methods*) podem ser aceleradas devido ao número de acessos não sequenciais necessŕios na memória. Esta implementação é mais rápida em *GPU*s do que em ambientes *CPU*s [11].

- **Produto entre matriz e vetor**. Uma otimização para esta operação foi implementada com a utilização da *StarPU* para gerar resultados satisfatórios [12].

- **Multi-acelerador *OpenCL***. Uma interface implementada de *OpenCL* para escalonamento eficiente, em tempo de execução, de tarefas em um sistema heterogêneo [13].

- **Processamento de imagem**. Algoritmos de processamento de imagem são fundamentais para muitos processos de visão computacional. Estes são extremamente custosos devido ao grande volume de imagens que são processadas. Existem implementações [14] destes que visam aproveitar arquiteturas heterogêneas para acelerar estes algoritmos.

- **Projeto *PEPPHER***. Este projeto aborda a utilização de sistemas de computadores híbridos que consistem de múltiplos *GPU*s e *CPU*s. Seu tipo de abordagem é pluralista, sendo seu objetivo abranger diferentes linguagens paralelas e *frameworks* em diferentes níveis de paralelismo [15].

- **SkePU**. É um esqueleto de *framework* para programação em *GPU*s e *CPU*s. Sua implementação foi sob a interface da *StarPU* e seu objetivo é explorar chamadas de funções entre diferentes esqueletos de *framework* independentes [16].

- **Outro novo modelo de programação**. Um modelo composto pela *StarPU* e *OpenCL* para que esconde detalhes de baixo nível por meio de uma interface de alto nível. Todas as operações específicas de cada arquitetura podem ser encapsuladas de forma eficiente em uma arquitetura heterogênea [17].

- **Características específicas de imagens**. Assim como visto anteriormente, é possível implementar algotimos de processamento de imagens

aproveitando-se da arquitetura heterogênea. Existem implementações eficientes [18] para extração de características, como canto e borda da imagem, que utilizam toda a capacidade de processamento da arquitetura heterogênea.

- **Biblioteca *MAGMA***. Esta é uma biblioteca de álgebra linear, similar em funcinalidade com *LAPACK*, mas com suporte a arquiteturas híbridas e *GPU*-exclusiva [19]. Ela usa o sistema de execução da *StarPU*.

- **Fatoração *QR***. Expressar um conjunto de tarefas como esta fatoração em três passos e outros algoritmos de alto nível eficientemente de forma que a performance obtida seja tão próxima quanto possível aos limitantes superiores de resultados teóricos [20].

- **Fatoração *LU***. Implementações de *kernels* desta fatoração podem ter desempenho de 1 $Tflop/s$, quando beneficiada pelo aproveitamento da arquitetura híbrida. Os resultados têm precisão suficientemente boa para que sejam aproveitados pela maior parte das aplicações [21].

4 Extensões em destaque

Devido ao desenvolvimento acelerado de diversas tecnologias para explorar métodos de paralelização em *CPU* e *GPU*, a *StarPU* oferece rotinas para integração com algumas das bibliotecas mais conhecidas.

A seguir, será apresentado um resumo de como a *StarPU* se relaciona com cada uma destas bibliotecas.

4.1 Integrando *MPI* com *StarPU*

Para ilustrar esta integração, considere um exemplo simples, de passagem de um *token* entre as máquinas. Para cada laço, cada nó recebe um valor de seu predecessor, executa uma operação e envia o resultado para seu sucessor.

```
1  for (loop = 0; loop < nloops; loop++)
2  {
3     if (!(loop == 0 && rank == 0))
4        MPI_Recv(&token,1,MPI_UNSIGNED,(rank+size-1)%size,0,MPI_COMM_WORLD,NULL);
5     increment_token_cpu(&token);
6     if (!(loop == lastloop && rank == lastrank))
7        MPI_Send(&token,1,MPI_UNSIGNED,(rank+1)%size,0,MPI_COMM_WORLD);
8  }
```

Neste instante, é desejável transferir os dados necessários da computação para os dispositivos *GPU*s. Isto pode ser feito manualmente, mas seria necessário assegurar a sincronização sobre os dados *MPI*, além das transferências entre as memórias da *CPU* e *GPU*.

Como a *StarPU* oferece mecanismos que fazem esse gerenciamento automaticamente, os programadores podem se concentrar apenas na aplicação. Com relação às funções, a *StarPU* dispõe de métodos semanticamente parecidos com os encontrados em *MPI*, facilitando a paralelização de aplicações.

Este módulo da biblioteca responsável pela integração é implementado sob um conjunto de funções baixo nível da *StarPU*, assim a semântica provê um poderoso mecanismo que pode ser usado para realizar as transferências de memória entre os processos. O objetivo é disponibilizar uma interface que implemente a comunicação por mensagens de modo consistente com a infraestrutura de gerenciamento de dados da *StarPU*. Para programadores, a mudança é mínima. Basta especificar estruturas de dados *StarPU* ao invés de ponteiros ao fazer a comunicação por mensagens.

Considerando o exemplo citado anteriormente, o código abaixo ilustra como é simples transferir dados e memória entre processos *StarPU* pelo módulo de integração do *MPI*. Graças a um mecanismo interno de controle progressivo de transferência, todas as travas envolvidas são liberadas quando a passagem de dados é finalizada. Todas as comunicações *MPI* são feitas por um único núcleo *StarPU* dedicado.

```
1  starpuvectordataregister(&tokenhandle,0,&token,1,sizeof(unsigned));
2  for (loop = 0; loop < nloops; loop++)
3  {
4     if (!(loop == 0 && rank == 0))
5        starpu_mpi_irecv_detached(&token_handle,1,MPI_UNSIGNED,(rank+size-1)%size,0,MPI_COMM_WORLD,NULL,NULL);
6     starpuinserttask(&increment_token_cl,STARPU_RW,tokenhandle,0);
7     if (!(loop == lastloop && rank == lastrank))
8        starpu_mpi_isend_detached(&token_handle,1,MPI_UNSIGNED,(rank+1)%size,0,MPI_COMM_WORLD,NULL,NULL);
9  }
10 starpu_task_wait_for_all();
```

Assim como no código *MPI*, cada nó recebe um valor do seu predecessor, executa uma operação e envia o resultado para seu sucessor. A ausência de dependências explicítas entre estas operações distintas comprova que o módulo está adequadamente integrado a *StarPU*, não apenas restrita a tarefas, mas a todo tipo de acesso a memória.

4.2 *SORS*: StarPU OpenMP Runtime Support

StarPU provê as funções necessárias para implementar um sistema de execução *OpenMP* em conformidade com as funcionalidades de dependência dados relacionados a tarefas introduzido na versão 4.0. O *SORS* foi projetado para que compiladores *OpenMP*, como o *Klang-OMP*, pudessem ser aproveitados por meio de plugins desenvolvidos.

Ao utilizar o *SORS*, a *thread* principal também executa *tasks OpenMP* como todas as outras em função da manutenção na especificação do modelo de execução padrão do *OpenMP*. Isto difere daquele da *StarPU* usual, onde a *thread* principal apenas executa operações *PUSH* de tarefas.

A implementação atual não suporta regiões paralelas aninhadas: estas podem ser criadas recursivamente. No entanto, apenas o primeiro nível da região paralela pode ter mais de um *worker*. Do ponto de vista da *StarPU*, as regiões paralelas do *SORS* são implementadas como um conjunto de *tasks StarPU* implicítas que seguem o modelo de execução *OpenMP*.

4.3 Observação

Esta subseção está fortemente associada ao exemplo apresentado por Augunnet et al. [22], na seção 2.

5 Referências Bibliográficas

[1] 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>.

[2] AUGONNET, C.; NAMYST, R. A unified runtime system for heterogeneous multicore architectures. In: *2nd Workshop on Highly Parallel Processing on a Chip (HPPC 2008)*. Las Palmas de Gran Canaria, Spain: [s.n.], 2008. Disponível em: <https://hal.inria.fr/inria-00326917>.

[3] AUGONNET, C. et al. Exploiting the cell/be architecture with the starpu unified runtime system. In: BERTELS, K. et al. (Ed.). *Embedded Computer Systems: Architectures, Modeling, and Simulation*. [S.l.]: Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5657). p. 329–339. ISBN 978-3-642-03137-3.

[4] AUGONNET, C.; THIBAULT, S.; NAMYST, R. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In: *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*. Delft, Netherlands: [s.n.], 2009. Disponível em: <https://hal.inria.fr/inria-00421333>.

[5] COURTÈS, L. *C Language Extensions for Hybrid CPU/GPU Programming with StarPU*. [S.l.], abr. 2013. 25 p. Disponível em: <https://hal.inria.fr/hal-00807033>.

[6] HENRY, S. et al. Toward OpenCL Automatic Multi-Device Support. In: SILVA, F.; DUTRA, I.; COSTA, V. S. (Ed.). *Euro-Par 2014*. Porto, Portugal: Springer, 2014. Disponível em: <https://hal.inria.fr/hal-01005765>.

[7] LACOSTE, X. et al. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In: *HCW'2014 workshop of IPDPS*. Phoenix, United States: IEEE, 2014. Disponível em: <https://hal.inria.fr/hal-00987094>.

[8] AGULLO, E. et al. Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. In: *Symposium on Application Accelerators in High Performance Computing (SAAHPC)*. Knoxville, United States: [s.n.], 2010. Disponível em: <https://hal.inria.fr/inria-00547616>.

[9] ODAJIMA, T. et al. Adaptive Task Size Control on High Level Programming for GPU/CPU Work Sharing. In: *The 2013 International Symposium on Advances of Distributed and Parallel Computing (ADPC 2013)*. Vietri sul Mare, Italy: [s.n.], 2013. Disponível em: <https://hal.inria.fr/hal-00920915>.

[10] HENRY SYLVAIN, H. *Programming Models and Runtime Systems for Heterogeneous Architectures*. Tese (Theses) — Université Sciences et Technologies - Bordeaux I, nov. 2013. Disponível em: <https://tel.archives-ouvertes.fr/tel-00948309>.

[11] OHSHIMA, S. et al. Implementation of FEM Application on GPU with StarPU. In: SIAM. *SIAM CSE13 - SIAM Conference on Computational Science and Engineering 2013*. Boston, United States, 2013. Disponível em: <https://hal.inria.fr/hal-00926144>.

[12] ROSSIGNON, C. Optimisation du produit matrice-vecteur creux sur architecture GPU pour un simulateur de réservoir. In: Inria Grenoble. *COMPAS'13 / RenPar'21 - 21es Rencontres francophones du Parallélisme*. Grenoble, France, 2013. Disponível em: <https://hal.inria.fr/hal-00773571>.

[13] SYLVAIN, H.; DENIS, A.; BARTHOU, D. Programmation unifiée multi-accélérateur OpenCL. *Techniques et Sciences Informatiques*, undefined or unknown publisher, v. 31, n. 8-9-10, p. 1233–1249, 2012. Disponível em: <https://hal.inria.fr/hal-00772742>.

[14] MAHMOUDI SIDI, A. et al. Traitements d'Images sur Architectures Parallèles et Hétérogènes. *Technique et Science Informatiques*, Editions Hermes, 2012. Disponível em: <https://hal.inria.fr/hal-00714858>.

[15] BENKNER, S. et al. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. *IEEE Micro*, Institute of Electrical and Electronics Engineers (IEEE), v. 31, n. 5, p. 28–41, 2011. Disponível em: <https://hal.inria.fr/hal-00648480>.

[16] DASTGEER, U.; KESSLER, C.; THIBAULT, S. Flexible runtime support for efficient skeleton programming on hybrid systems. In: *International conference on Parallel Computing (ParCo)*. Gent, Belgium: [s.n.], 2011. Disponível em: <https://hal.inria.fr/inria-00606200>.

[17] SYLVAIN, H. Programmation multi-accélérateurs unifiée en OpenCL. In: *RenPAR'20*. Saint Malo, France: [s.n.], 2011. p. XXX. Disponível em: <https://hal.archives-ouvertes.fr/hal-00643257>.

[18] MAHMOUDI SIDI, A. et al. Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes. In: *Rencontres francophones du parallélisme*. Saint-Malo, France: [s.n.], 2011. Disponível em: <https://hal.inria.fr/inria-00606195>.

[19] AGULLO, E. et al. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In: HWU, W. mei W. (Ed.). *GPU Computing Gems*. Morgan Kaufmann, 2010. v. 2. Disponível em: <https://hal.inria.fr/inria-00547847>.

[20] AGULLO, E. et al. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In: *25th IEEE International Parallel & Distributed Processing Symposium*. Anchorage, United States: [s.n.], 2011. Disponível em: <https://hal.inria.fr/inria-00547614>.

[21] AGULLO, E. et al. LU Factorization for Accelerator-based Systems. In: *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11)*. Sharm El-Sheikh, Egypt: [s.n.], 2011. Disponível em: <https://hal.inria.fr/hal-00654193>.

[22] AUGONNET, C. et al. *StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators*. [S.l.], maio 2014. Disponível em: <https://hal.inria.fr/hal-00992208>.