

Análise do Impacto de Runtimes Task-Based no Desempenho de Algoritmos de Álgebra Linear Densa

Matheus Augusto Tregnago, Philippe O. A. Navaux, Lucas Mello Schnorr

¹ Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{matregnago, navaux, schnorr}@inf.ufrgs.br

Resumo. *Runtimes de escalonamento de tarefas dependentes como StarPU e PaRSEC abstraem a complexidade de plataformas heterogêneas, mas adotam estratégias diferentes de escalonamento e gerenciamento de dados. Este trabalho compara o desempenho dos dois runtimes nas fatorações de Cholesky e LU sem pivotamento da biblioteca Chameleon, mantendo fixa a biblioteca de álgebra linear para isolar o efeito do runtime. Este estudo exigiu estender a interface Dynamic Task Discovery (DTD) do PaRSEC e os codelets do Chameleon com suporte a GPU. Os experimentos mostram que o StarPU é superior ou equivalente na maior parte dos casos. O `parsec:gd` só obtém vantagem na máquina com RTX 4070 (455 GFLOPS em FP64, aproximadamente metade da capacidade da CPU). Na máquina com RTX 4090 (1.290 GFLOPS, quase o dobro da CPU), o StarPU vence em toda a faixa avaliada, com cerca de 30% de vantagem mesmo no tamanho de bloco ótimo de cada runtime.*

1. Introdução

Nos últimos anos, as arquiteturas de computação de alto desempenho têm se tornado cada vez mais heterogêneas, combinando processadores multicore convencionais com aceleradores especializados, como GPUs, FPGAs e outros aceleradores. Essa diversidade de tecnologias, somada às diferenças entre os modelos de execução e as hierarquias de memória, torna difícil programar essas plataformas de forma eficiente. Uma das soluções para esse problema é a utilização de runtimes que utilizam o modelo Sequential Task Flow (STF). Nesse modelo, uma aplicação é decomposta em tarefas com dependências de dados explícitas, o que permite escalonar sua execução de forma dinâmica, conforme os recursos disponíveis [Pei et al. 2022]. Para facilitar o uso desse paradigma, diversos runtimes de escalonamento de tarefas foram criados para mapear as tarefas de uma aplicação para os recursos computacionais disponíveis, de maneira transparente ao desenvolvedor. Esses sistemas decidem quando e onde cada tarefa será executada, gerenciam as transferências de dados necessárias entre as diferentes unidades de processamento e memórias e equilibram a carga de trabalho entre os recursos [Thibault 2018].

Dentre os runtimes de escalonamento de tarefas existentes, este trabalho compara especificamente o StarPU e o PaRSEC por serem dois runtimes utilizados pelo Chameleon como backend para execução das tarefas e por estarem, entre os runtimes suportados, entre os mais maduros e amplamente utilizados para algoritmos de álgebra linear densa em plataformas heterogêneas. No StarPU [Augonnet et al. 2011], o desenvolvedor deve fornecer diferentes implementações para uma mesma tarefa, uma para CPU e outra para GPU, por exemplo, submetendo o grafo seguindo o modelo STF. É o próprio ambiente

de execução que seleciona automaticamente a implementação mais adequada, considerando o ambiente computacional e a disponibilidade de recursos, buscando otimizar o desempenho. Isso isola o desenvolvedor dos detalhes de escalonamento, permitindo criar aplicações mais simples e fáceis de escalar. PaRSEC [Bosilca et al. 2011] tem foco em ambientes de memória distribuída compostos por múltiplos nós multicore, possivelmente equipados com aceleradores. Diferentemente de runtimes baseados no modelo STF, como o StarPU, que instanciam explicitamente um nó de DAG na memória para cada tarefa submetida, o PaRSEC representa as dependências entre tarefas de forma compacta e simbólica, o que lhe permite escalar para milhões de tarefas com baixo overhead de memória. O escalonamento é totalmente distribuído, cada nó executa sua própria instância do escalonador, sem coordenação central, o que favorece a sobreposição entre computação e comunicação em estratégia similar àquela usada pelo StarPU-MPI.

Embora StarPU e PaRSEC compartilhem o objetivo de abstrair o escalonamento de tarefas em plataformas heterogêneas, os escalonadores de cada runtime avaliados neste trabalho diferem em uma dimensão central. O StarPU decide o escalonamento dinamicamente, sendo orientado por um modelo de desempenho aprendido a partir do histórico de execuções. Os escalonadores do PaRSEC realizam escalonamento sem tais modelos de desempenho. Estas diferentes estratégias sugerem que a comparação do desempenho obtido nestes runtimes se faz necessária. Uma análise do estado da arte nos permite estabelecer que diversos trabalhos realizam tal comparação, inclusive com algoritmos de álgebra linear densa, mas cada um deles isola apenas parte do problema, limitando a comparação a um único runtime ou comparando runtimes distintos por meio de bibliotecas de álgebra linear diferentes. Por exemplo, Hoque et al. [Hoque et al. 2017] comparam os dois paradigmas de programação oferecidos pelo PaRSEC: o Parameterized Task Graph (PTG) e o Dynamic Task Discovery (DTD) utilizando o Chameleon em conjunto com PaRSEC, StarPU e QUARK. A avaliação, no entanto, é restrita a execuções somente em CPU, sem considerar aceleradores. Lisito et al. [Lisito et al. 2025] avaliam a fatoração LU com pivotamento parcial, comparando o StarPU (com o Chameleon) e o PaRSEC (com o DPLASMA). Como cada runtime é avaliado em conjunto com uma biblioteca de álgebra linear diferente, eventuais diferenças de desempenho observadas podem ser atribuídas tanto ao runtime quanto à implementação específica de cada biblioteca. Lacoste et al. [Lacoste et al. 2014] substituem o escalonador interno do solver direto esparsa PaStiX pelo StarPU e pelo PaRSEC, mantendo a biblioteca fixa e avaliando ambos os runtimes em plataformas heterogêneas equipadas com aceleradores. A comparação, no entanto, se dá no contexto da álgebra linear esparsa, cujo grafo de tarefas é irregular e determinado pela estrutura de esparsidade da matriz, o que impede transferir suas conclusões para as fatorações densas por blocos consideradas aqui. Agullo et al. [Agullo et al. 2015] comparam a fatoração de Cholesky sobre o StarPU com limites teóricos de desempenho em plataformas heterogêneas. Eles consideram apenas o StarPU, sem comparação direta com outro runtime de escalonamento de tarefas.

A diferença de estratégia de escalonamento entre StarPU e PaRSEC sugere, à primeira vista, que o StarPU deveria ter uma vantagem sistemática sobre o PaRSEC em algoritmos de álgebra linear densa. Um dos objetivos deste trabalho é verificar empiricamente se essa expectativa se confirma, fixando uma única biblioteca de álgebra linear densa e variando apenas o runtime e o escalonador utilizado. Para colocar tal comparação em prática, este trabalho utiliza o Chameleon [Agullo et al. 2012], uma biblioteca de

álgebra linear densa que implementa fatorações como LU, Cholesky e QR por meio de algoritmos por blocos [Buttari et al. 2007], delegando o escalonamento de suas tarefas a runtimes externos, o StarPU e o PaRSEC. Os experimentos foram executados em duas partições do cluster PCAD, no INF/UFRGS: (i) tupi, com 1× Intel Core i9-14900KF e 1× NVIDIA RTX 4090; (ii) poti, com 1× Intel Core i7-14700KF e 1× NVIDIA RTX 4070. Por intermédio de metodologia reprodutível baseada em Nix, os resultados indicam que o StarPU é superior ou equivalente na maior parte dos casos, e que a exceção depende do equilíbrio entre a capacidade de CPU e de GPU de cada máquina. Na poti, com a RTX 4070 (455 GFLOPS em FP64), o `parsec:gd` supera o StarPU em 4% a 8% para $N \geq 20.000$; na tupi, com a RTX 4090 (1.290 GFLOPS), o StarPU vence em toda a faixa avaliada, com cerca de 30% de vantagem em $N = 40.000$ mesmo no tamanho de bloco ótimo de cada runtime.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 descreve os conceitos utilizados neste trabalho, enquanto a Seção 3 apresenta os métodos e materiais utilizados. A Seção 4 detalha nossa metodologia experimental, seguida pelos resultados experimentais na Seção 5. Por fim, a Seção 6 conclui este trabalho.

2. Fundamentação Teórica

Apresentamos os runtimes StarPU e PaRSEC, e a biblioteca de álgebra linear Chameleon.

StarPU [Augonnet et al. 2011] é um framework escrito em C para programação orientada a tarefas em plataformas híbridas compostas por CPUs e GPUs, com suporte também a ambientes distribuídos por meio da extensão StarPU-MPI. As tarefas de uma aplicação StarPU são definidas por meio de codelets, estruturas que agrupam uma ou mais implementações de uma mesma operação, além da forma como cada um de seus parâmetros é acessado. A partir da ordem de submissão das tarefas e dos modos de acesso declarados para cada dado, o StarPU desenrola implicitamente, tarefa a tarefa, o grafo acíclico dirigido (DAG) completo de dependências, sem que o programador precise expressá-lo manualmente. Em tempo de execução, o runtime escolhe qual das implementações disponíveis será utilizada para cada tarefa, considerando a disponibilidade dos recursos, a distribuição de carga entre os workers e o custo das cópias de dados envolvidas. O gerenciamento de memória nos aceleradores também é responsabilidade do StarPU, que realiza alocação e liberação de buffers na GPU, cópias entre a memória do host e do dispositivo, e a sobreposição dessas transferências com a computação por meio de prefetching. Empregamos dois escalonadores do StarPU: o `dmda` (Deque Model Data Aware), que mantém um modelo de desempenho por par tarefa-recurso, estimado a partir do histórico de execuções, e atribui cada tarefa ao recurso que minimiza o tempo estimado de conclusão, somando o tempo de computação previsto ao custo estimado de transferência de dados; e o `dmdas`, uma variante que processa as tarefas seguindo a ordem de prioridade dada pela distância ao caminho crítico do DAG, ao custo de uma ordenação adicional. Os demais escalonadores do StarPU decidem sem modelo de desempenho e sem estimativa de custo de transferência, e por isso não representam a forma como o runtime é empregado em plataformas heterogêneas.

PaRSEC [Bosilca et al. 2011] é um framework para escalonamento de tarefas voltado a ambientes de memória distribuída compostos por múltiplos nós multicore. Diferentemente de runtimes STF como o StarPU, que desenrolam explicitamente, tarefa a tarefa,

um DAG completo de dependências, o PaRSEC representa o grafo de dependências de forma compacta e simbólica, por meio de um formato conhecido como Parameterized Task Graph (PTG), também chamado de Job Data Flow (JDF). Além do PTG, o PaRSEC oferece a interface Dynamic Task Discovery (DTD) [Hoque et al. 2017], na qual as tarefas são inseridas sequencialmente, à maneira do modelo STF, e as dependências são inferidas dos modos de acesso aos dados. É essa a interface utilizada pelo Chameleon. O escalonamento é totalmente distribuído, cada nó executa sua própria instância do escalonador, que utiliza filas locais por thread e mecanismos de roubo de tarefas sensíveis à hierarquia NUMA para favorecer o reaproveitamento de cache. Assim como o StarPU sobrepõe transferências e computação por meio de prefetching, o PaRSEC sobrepõe computação e comunicação entre nós: as comunicações são implícitas, inferidas diretamente das dependências de dados do grafo, e tratadas de forma assíncrona por uma thread dedicada. Assim como o StarPU, o PaRSEC também oferece suporte a aceleradores por meio de múltiplas implementações para uma mesma tarefa, permitindo que o escalonador decida em tempo de execução em qual unidade. Empregamos dois dos dez escalonadores oferecidos pelo PaRSEC: o `lfq` (Local Flat Queues), que é o escalonador selecionado por padrão pelo runtime, onde cada worker mantém uma fila local ordenada por prioridade e usa roubo de tarefas das filas vizinhas, na ordem de distância na hierarquia de hardware, quando fica ocioso; e o `gd` (Global Dequeue), o mais simples da família, que mantém uma única fila dupla compartilhada pelos workers, com balanceamento de carga mas sem localidade de dados nem roubo de tarefas. Escolhemos o `lfq` por ser o padrão do runtime e o `gd` por representar o extremo oposto, com uma única fila global e sem localidade.

Chameleon [Agullo et al. 2012] é uma biblioteca escrita em C que fornece rotinas para a resolução de sistemas lineares gerais densos, sistemas simétricos positivos definidos e problemas de mínimos quadrados lineares, por meio das fatorações LU, Cholesky, QR e LQ. Assim como as bibliotecas PLASMA e MAGMA, o Chameleon segue a abordagem de algoritmos por blocos [Buttari et al. 2007], em que cada fatoração é decomposta em uma sequência de tarefas de granularidade fina que operam sobre submatrizes quadradas, reduzindo a necessidade de sincronizações globais e expondo um grau de paralelismo bem maior do que o obtido pelas versões tradicionais orientadas a blocos de linha/coluna, como as do LAPACK e do ScaLAPACK. A fatoração de Cholesky por blocos, por exemplo, é expressa como uma sequência de chamadas às rotinas POTRF, TRSM, SYRK e GEMM, cada uma correspondendo a uma tarefa cujas dependências de dados definem um DAG. Em vez de escalonar essas tarefas de forma estática, o Chameleon delega essa responsabilidade a um runtime de escalonamento de tarefas, como o StarPU, o PaRSEC ou o OpenMP, enquanto a implementação interna de cada tarefa reutiliza rotinas clássicas de alto desempenho. Esta separação entre o algoritmo, na forma de um grafo de tarefas, e sua execução, é o que permite ao Chameleon explorar arquiteturas heterogêneas compostas por CPUs e GPUs sem reescrita do algoritmo para cada plataforma.

3. Métodos e Materiais

Apresentamos nossos métodos e materiais, iniciando pelas cargas de trabalhos na forma de algoritmos de fatoração que fazem parte de um grande número de aplicações, pelas adaptações necessárias no Chameleon e PaRSEC para tornar possível este trabalho, seguindo das configurações de hardware e software e o projeto experimental.

3.1. Carga de trabalho: algoritmos de fatoração densa Cholesky e LU

Consideramos dois algoritmos de fatoração disponíveis no Chameleon: a fatoração de Cholesky (`potrf`) e a fatoração LU sem pivotamento (`getrf_nopiv`). A fatoração de Cholesky decompõe uma matriz real simétrica positiva definida A , de ordem $n \times n$, na forma $A = LL^T$, em que L é uma matriz real triangular inferior de ordem $n \times n$ com elementos diagonais positivos. O algoritmo por blocos utiliza quatro tipos de tarefa: POTRF, SYRK, TRSM e GEMM, todas com implementações em CPU e em GPU. Já a fatoração LU sem pivotamento decompõe uma matriz real A , de ordem $n \times n$, na forma $A = LU$, em que L é triangular inferior e U é triangular superior. O projeto desse algoritmo por blocos para sistemas acelerados por GPU é descrito por Agullo et al. [Agullo et al. 2011], e utiliza três tipos de tarefa: GETRF, TRSM e GEMM, das quais apenas GETRF, que fatora o bloco da diagonal, está restrita à CPU.

3.2. Implementação do Suporte à GPU no PaRSEC

O Chameleon utiliza o PaRSEC por meio de um fork mantido por desenvolvedores do próprio Chameleon, que é a base deste trabalho. Nessa base, a comparação entre StarPU e PaRSEC em plataforma heterogênea não era possível: enquanto os codelets do Chameleon para o StarPU já têm suporte maduro em GPU, a interface DTD do fork não suportava a execução de tarefas em GPU, e os codelets do Chameleon para o PaRSEC estavam limitados à CPU.

No PaRSEC, adicionamos uma nova função de inserção de tarefas, que permite registrar, além da implementação em CPU, uma implementação em GPU para a mesma tarefa. Em tempo de execução, a escolha do dispositivo reutiliza o mecanismo de escalonamento de GPU já empregado pelo backend PTG: uma tarefa que escreve um bloco já residente na GPU é enviada a ela; nas demais, o PaRSEC compara a carga pendente da GPU com um peso estático da CPU, ambos derivados do desempenho teórico de cada dispositivo, e só envia a tarefa à CPU quando a GPU está carregada o bastante.

Essa mudança expôs um problema de coerência de dados: quando uma tarefa executada em CPU escrevia um bloco, o caminho de execução da DTD não invalidava corretamente eventuais cópias desse bloco já presentes na memória da GPU. Uma tarefa de GPU subsequente podia então ler dados desatualizados, produzindo resultados numericamente incorretos. Corrigimos o problema invalidando explicitamente as cópias de dispositivo de um bloco sempre que uma tarefa de CPU o modifica.

Com o suporte a GPU já disponível na interface DTD do PaRSEC, implementamos, no lado do Chameleon, versões em GPU dos codelets utilizados pelas fatorações avaliadas neste trabalho. Estas até então só possuíam implementação em CPU quando executadas com o PaRSEC. Cada uma dessas implementações delega o cálculo às bibliotecas cuBLAS e cuSOLVER, de forma análoga aos codelets equivalentes já existentes para o StarPU, associando-as à sua contraparte de CPU por meio da nova função de inserção de tarefas com suporte a CUDA.

3.3. Configuração de Hardware e Software

A Tabela 1 especifica a configuração de hardware das partições do Parque Computacional de Alto Desempenho (PCAD), no INF/UFRGS, utilizadas nos experimentos, ambas equipadas com processadores da Intel e GPUs da NVIDIA. Indicamos a quantidade de

P-cores (núcleos de alto desempenho), sendo que os núcleos restantes até o total informado na coluna CPU são núcleos de eficiência. A última coluna traz o pico teórico de desempenho de cada GPU em FP64.

Tabela 1. Especificação de hardware das partições utilizadas.

Partição	CPU	RAM	GPU	GFLOPS
tupi	i9-14900KF, 24c (8× P-cores)	128 GB	RTX4090, 24GB	1.290
poti	i7-14700KF, 20c (8× P-cores)	96 GB	RTX4070, 12GB	455

Utilizamos o Chameleon na versão 1.4.0, com suporte a CUDA habilitado. Como runtimes de escalonamento, utilizamos o StarPU 1.4.7 e uma versão do PaRSEC derivada de um fork mantido por desenvolvedores do próprio Chameleon, ambos também compilados com suporte a CUDA e com a coleta de rastros de execução habilitada. Para equalizar o número de fluxos de execução em GPU entre os dois runtimes, fixamos `STARPU_NWORKER_PER_CUDA=4` (o que equivale a quatro CUDA *streams*). No PaRSEC, essa quantidade é definida em tempo de compilação (`PARSEC_MAX_STREAMS=6`, das quais quatro são de execução e duas de transferência), sem parâmetro equivalente configurável em tempo de execução. Utilizamos CUDA 12.4, gcc 13.3.0 e Debian 12 como sistema operacional, com driver NVIDIA 590.44.01. A reprodutibilidade de todo o ambiente é garantida por meio do gerenciador de pacotes Nix, com as dependências fixadas em um flake, publicado em <https://github.com/matreagnago/chameleon-runtime-benchmarks/tree/sscad2026>. A análise dos dados ocorre usando ferramentas modernas de ciência de dados junto com o framework StarVZ [Pinto et al. 2021] para a visualização dos rastros de execução coletados pelo StarPU e pelo PaRSEC.

3.4. Projeto Experimental

Realizamos três projetos experimentais complementares para investigar a influência do escalonador, do tamanho do problema e da máquina alvo no desempenho das fatorações de Cholesky (`potrf`) e LU sem pivotamento (`getrf_nopiv`). 1/ O primeiro projeto experimental determina o tamanho de bloco ideal para cada máquina, um parâmetro que precisa estar fixo antes de comparar os runtimes entre si. 2/ O segundo projeto experimental, um projeto fatorial completo, usa esse tamanho de bloco ótimo para comparar o desempenho do StarPU e do PaRSEC em função do tamanho do problema. Por fim, 3/ o terceiro projeto experimental usa rastros de execução para explicar, em nível de tarefa, a origem das diferenças de desempenho observadas no segundo projeto experimental. Em todos eles, a variável de resposta é o GFLOPS em relatório da própria aplicação.

1/ Determinar Tamanho de Bloco Ideal. Realizamos uma varredura sistemática de configurações de bloco em precisão dupla. Fixamos o tamanho da matriz de entrada em $N = 40.000$ e variamos o tamanho do bloco dentro de um intervalo. Esse tamanho de entrada foi escolhido por permitir a saturação dos recursos computacionais e expor com clareza o efeito do tamanho de bloco sobre o desempenho. Cruzamos essas configurações com as duas fatorações (`potrf`, `getrf_nopiv`) e quatro combinações de runtime e escalonador. Os dois escalonadores do StarPU são guiados por um modelo de desempenho medido em tempo de execução, enquanto os dois do PaRSEC decidem o uso da GPU de forma estática. A expectativa que orienta todos os três projetos experimentais,

portanto, é a de que `starpu:dmda` e `starpu:dmdas` superem as duas configurações do PaRSEC. Executamos o mesmo desenho experimental nas duas partições empregadas, considerando as particularidades (número de threads) individuais. A ordem de execução foi aleatória, com cinco repetições por configuração, para caracterizar a variabilidade entre execuções de uma mesma configuração. É o pico de GFLOPS FP64 observado em cada máquina que determina o tamanho de bloco a ser utilizado na sequência.

2/ Efeitos de Interação entre Fatores. O segundo projeto experimental, um projeto fatorial completo, investiga os efeitos de interação entre três fatores que influenciam o desempenho das fatorações: a máquina utilizada, a combinação de runtime e escalonador, e o tamanho do problema. Este tem dez valores entre 5.000 e 60.000, os mesmos nas duas máquinas, de modo que o único fator que muda entre elas é o tamanho de bloco. Para cada uma das duas fatorações, executamos todas as combinações desse desenho fatorial em ordem aleatória, novamente com cinco repetições por configuração.

3/ Rastreamento Comportamental. O objetivo é explorar como cada escalonador se comporta em cada uma das aplicações e máquinas. Empregamos os mecanismos de coleta de rastros de execução do StarPU e do PaRSEC, por meio de oito execuções representativas por máquina, uma para cada combinação entre as duas fatorações e as quatro combinações de runtime e escalonador. Como o StarVZ foi originalmente projetado para consumir rastros do StarPU, adaptamos o pipeline de análise para o PaRSEC, onde os rastros brutos são primeiro convertidos para o formato Paje por meio da ferramenta `dbp2paje` do próprio PaRSEC, preservando os nomes de containers necessários para o StarVZ identificar corretamente workers e streams de GPU. O restante do pipeline de análise é idêntico ao utilizado para o StarPU. Assim, temos um método único para comparar o escalonamento realizado por cada um dos runtimes em ambiente único de análise.

4. Resultados

Esta seção apresenta os resultados dos três projetos experimentais descritos na Seção anterior. A Seção 5.1 usa a varredura de tamanho de bloco para fixar esse parâmetro em cada máquina. A Seção 5.2 usa esse valor para comparar os quatro pares de runtime e escalonador em função do tamanho do problema. A Seção 5.3 recorre aos rastros de execução para explicar, em nível de tarefa, por que o runtime vencedor muda de uma máquina para a outra.

4.1. Escolha do tamanho de bloco

A Figura 1 mostra os resultados que nos permitem determinar o tamanho de bloco ideal para as quatro combinações de runtime e escalonador em precisão dupla, com $N = 40.000$ fixo em cada máquina. A figura traz o GFLOPS médio em função do tamanho de bloco b . Cada ponto é uma configuração medida, e a barra vertical marca o mínimo e o máximo observados entre as repetições. A cor identifica o runtime, onde o StarPU aparece em tons de vermelho (`starpu:dmda` e `starpu:dmdas`) e o PaRSEC em tons de azul (`parsec:lfq` e `parsec:gd`). Os painéis estão dispostos em uma grade: cada linha é uma das duas fatorações e cada coluna é uma das duas máquinas. Na máquina poti, as quatro configurações convergem para um pico em torno de $b = 500$; na máquina tupi, elas atingem um platô a partir de $b = 1.000$, valor em que as duas configurações do StarPU alcançam seu máximo e a partir do qual as do PaRSEC variam

pouco. Os tamanhos de bloco adotados nas duas etapas seguintes são os que maximizam o GFLOPS em cada máquina: $b = 1.000$ na máquina tupi e $b = 500$ na máquina poti.

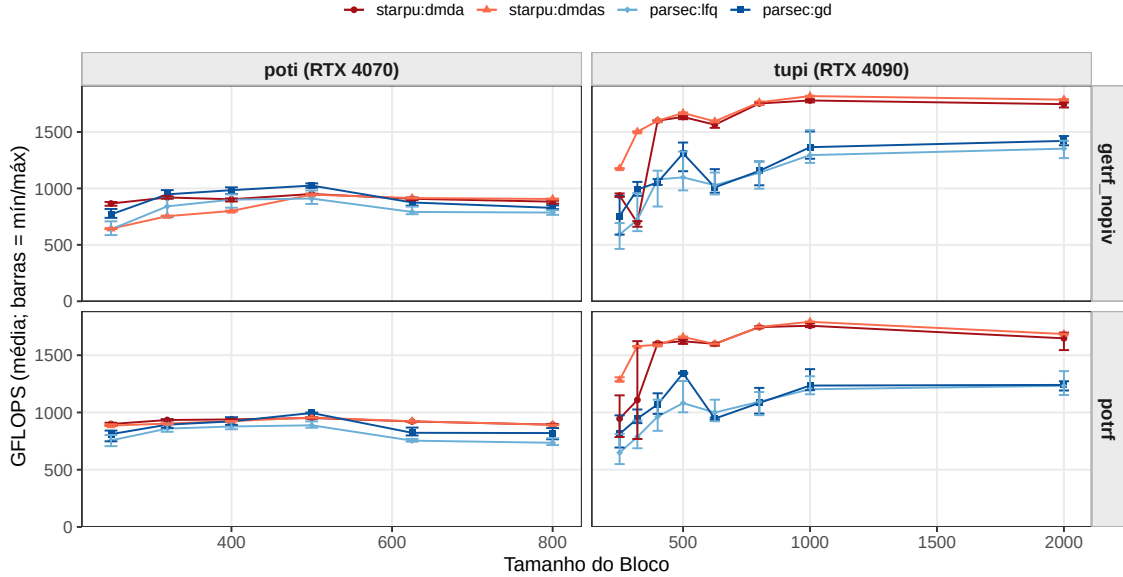


Figura 1. GFLOPS em função do tamanho de bloco b , por algoritmo e máquina.

4.2. Desempenho em função do tamanho do problema

A Figura 2 apresenta o resultado do segundo projeto experimental. A figura traz o GFLOPS médio em função da ordem N da matriz de entrada, com cada máquina rodando no seu tamanho de bloco ótimo. Cada ponto é a média das repetições daquela configuração naquele N e a barra vertical corresponde a um desvio-padrão para cada lado. O padrão que aparece na Figura 2 é a observação central que organiza o restante desta seção. Na máquina poti, o StarPU atinge um desempenho melhor até $N \approx 10.000$, mas a partir de 20.000 o `parsec:gd` obteve o melhor resultado. Na máquina tupi, por sua vez, o StarPU obteve o melhor resultado em toda a faixa de N , com a vantagem sobre o PaRSEC aumentando à medida que N cresce. Esse resultado contraria a intuição de que o StarPU, por manter um modelo de desempenho aprendido e refinado em tempo de execução, deveria sempre obter um desempenho melhor do que o PaRSEC, que decide o escalonamento de GPU de forma estática desconsiderando modelos de desempenho, sem medir tempos reais de execução.

4.3. Análise de rastros de execução

Para entender o resultado observado na seção anterior, analisamos em detalhe os rastros de execução coletados para a fatoração de Cholesky com $N = 40.000$ e o bloco ideal de cada máquina: $b = 1.000$ na máquina tupi e $b = 500$ na máquina poti. A Figura 3 decompõe, nas duas máquinas, o desempenho de cada configuração nas parcelas entregues por CPU e por GPU, calculadas como as operações de ponto flutuante executadas por cada classe divididas pelo *makespan*. As duas parcelas empilhadas somam o desempenho da execução, indicado acima de cada barra.

Na poti (coluna à esquerda da Figura 3), a parcela entregue pela GPU é praticamente a mesma nas quatro configurações, entre 364 e 399 GFLOPS na Cholesky. A ordem

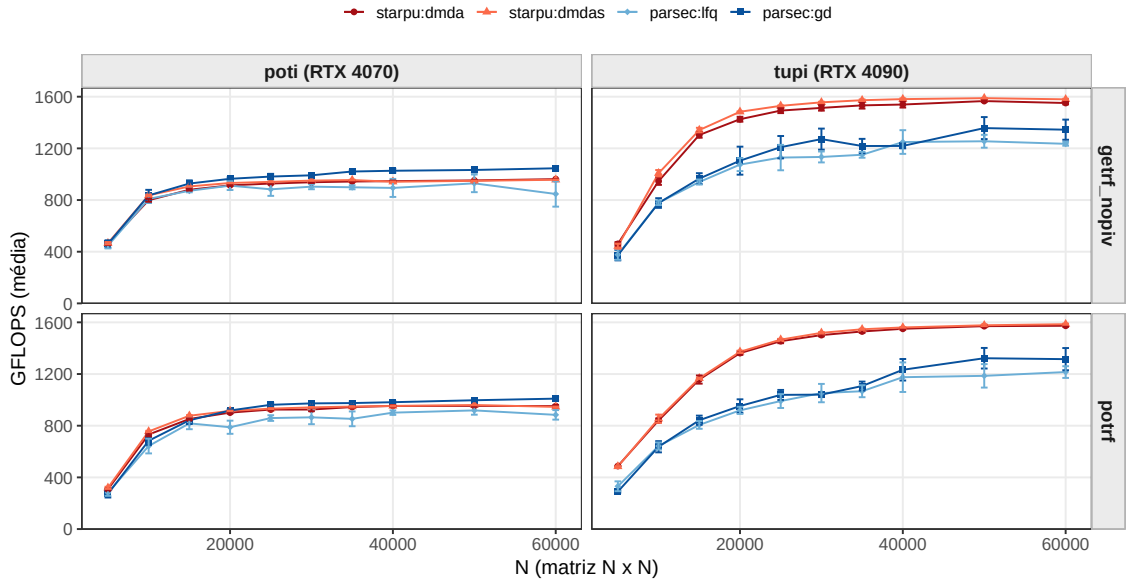


Figura 2. GFLOPS médio em função de N , por algoritmo e máquina.

das barras é, portanto, decidida na CPU: o `parsec:gd` entrega 607 GFLOPS contra 542 do `starpu:dmda`, e esses 65 GFLOPS a mais compensam com folga os 26 a menos que ele entrega na GPU. O mesmo padrão se repete na fatoração LU. O `parsec:gd` vence porque preserva a CPU. Os dois runtimes recebem os mesmos 19 *workers* de CPU, mas o StarPU cria, além deles, uma thread dedicada para cada um dos quatro *workers* CUDA, enquanto o PaRSEC despacha a GPU de forma oportunista, a partir do próprio *worker* que obtém acesso ao dispositivo, sem nenhuma thread reservada para essa gestão.

Na tupi (coluna à direita da Figura 3), esse mecanismo deixa de compensar. A GPU passa a ser o recurso dominante, e as duas configurações do StarPU extraem dela mais de 1.100 GFLOPS na Cholesky, contra 942 do `parsec:gd`. Mais decisivo, porém, é que o PaRSEC deixa de vencer também na CPU, invertendo o que se observa na poti: 371 GFLOPS do `parsec:gd` contra 669 do `starpu:dmda`. Os rastros mostram a razão: ali os *workers* de CPU do `parsec:gd` ficam ociosos em metade do *makespan*, contra menos de 10% nas duas configurações do StarPU, porque a heurística estática de dispositivo do PaRSEC concentra o trabalho na GPU e esgota o que resta para a CPU antes do fim da execução.

Painéis espaço-tempo e progressão do DAG. As Figuras 4 e 5 são painéis do StarVZ [Pinto et al. 2021, Garcia Pinto et al. 2018]. Ambas trazem o tempo de execução no eixo horizontal, compartilhado entre os quatro painéis para que os *makespans* possam ser comparados diretamente. Elas trazem apenas os rastros da poti, por ser a máquina mais representativa para esta análise, pois é nela que ocorre a inversão de desempenho que motiva a investigação. Na Figura 4, cada linha do eixo vertical é um recurso de execução: os *workers* de CPU e, abaixo deles, as quatro *streams* de GPU. Cada retângulo é uma tarefa, posicionada no recurso em que executou e colorida pelo seu tipo. A faixa cinza vertical no início de cada painel marca o CPB (*Critical Path Bound*), o limite inferior de tempo imposto pelo caminho crítico do DAG, e o número rotacionado à direita é o

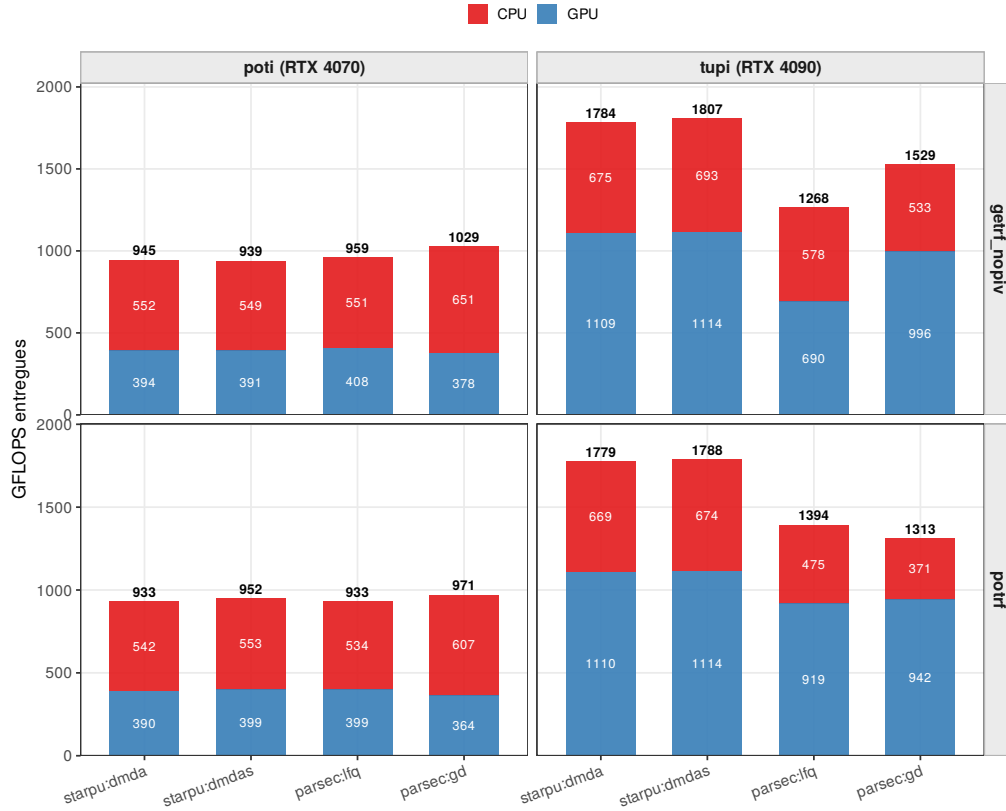


Figura 3. GFLOPS entregas por classe de recurso (CPU/GPU) na poti e na tupi, por algoritmo e configuração.

makespan da execução. A figura dá forma visual à decomposição da seção anterior. Os dois runtimes executam tarefas em quatro *streams* de GPU: no StarPU, uma por *worker* CUDA; no PaRSEC, quatro das seis *streams* CUDA, já que as outras duas são reservadas às transferências entre *host* e dispositivo. Nas faixas de CPU, porém, as configurações se distinguem: as duas do StarPU e o `parsec:gd` aparecem saturadas, com 1,6% a 8,7% de ociosidade agregada nos *workers*, enquanto o `parsec:lfq` exibe lacunas brancas largas e recorrentes ao longo de toda a execução, que somam 13,9%. É essa ociosidade que explica por que o `parsec:lfq` entrega apenas 534 GFLOPS de CPU na Figura 3, a menor parcela entre as quatro configurações, mesmo com tarefas de CPU tão baratas quanto as do `parsec:gd`.

A Figura 5 usa o mesmo eixo de tempo, mas traz no eixo vertical o índice k da iteração à qual cada tarefa pertence. O `starpu:dmda` mantém as linhas coladas, com 1,9 iterações simultâneas em média, concluindo uma iteração antes de avançar para a seguinte, enquanto o `starpu:dmdas` abre um envelope largo, com 28,7 iterações simultâneas em média, porque sua priorização pelo caminho crítico adianta tarefas de iterações futuras. As duas configurações do PaRSEC progridem em degraus, com 7,7 e 9,3 iterações em média, mas por outro motivo: são tarefas retardatárias de iterações antigas retidas nas filas. Estas diferenças observadas na forma como o escalonamento das tarefas por iteração ocorre acaba sendo uma assinatura do comportamento de cada escalonador. Podemos perceber que ainda que as diferenças de tempo sejam relativamente pequenas, estas assinaturas permitem compreender as estratégias de escalonamento e podem trazer

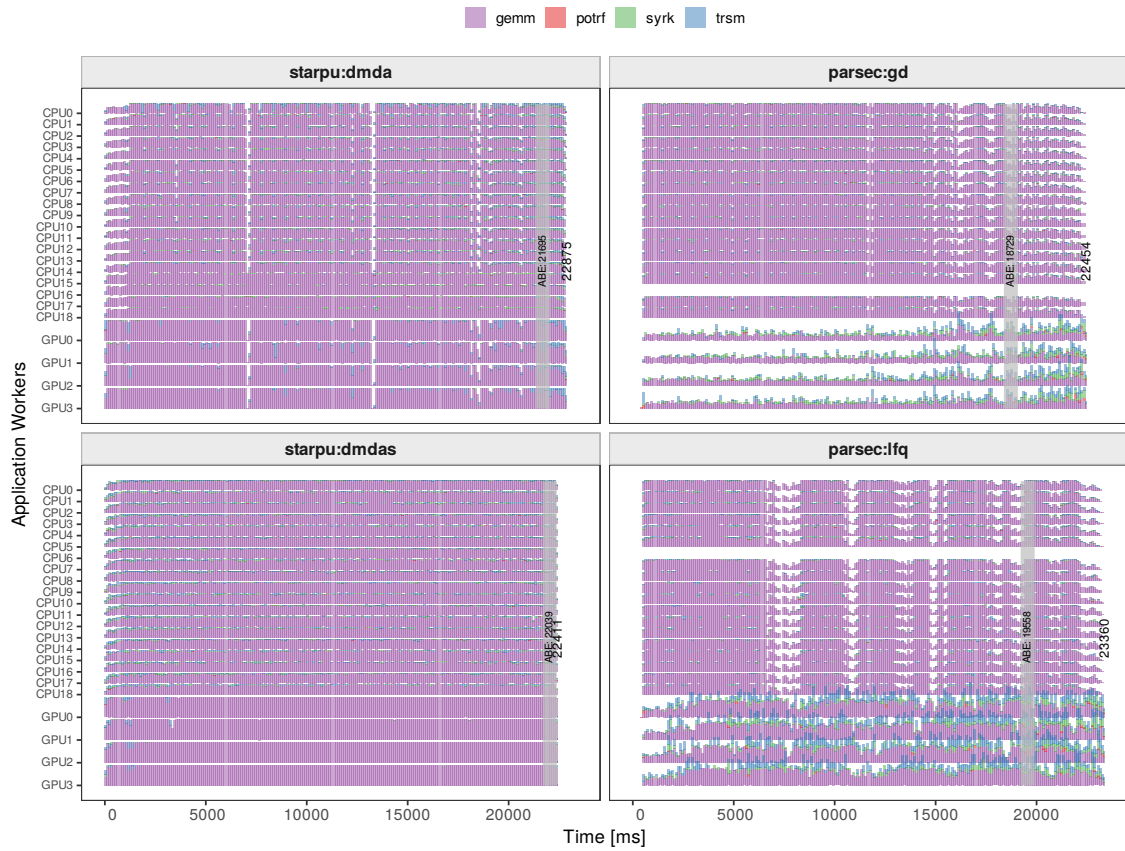


Figura 4. Painéis espaço-tempo de StarPU e ParSEC na poti.

indícios fundamentais na melhoria das estratégias de escalonamento.

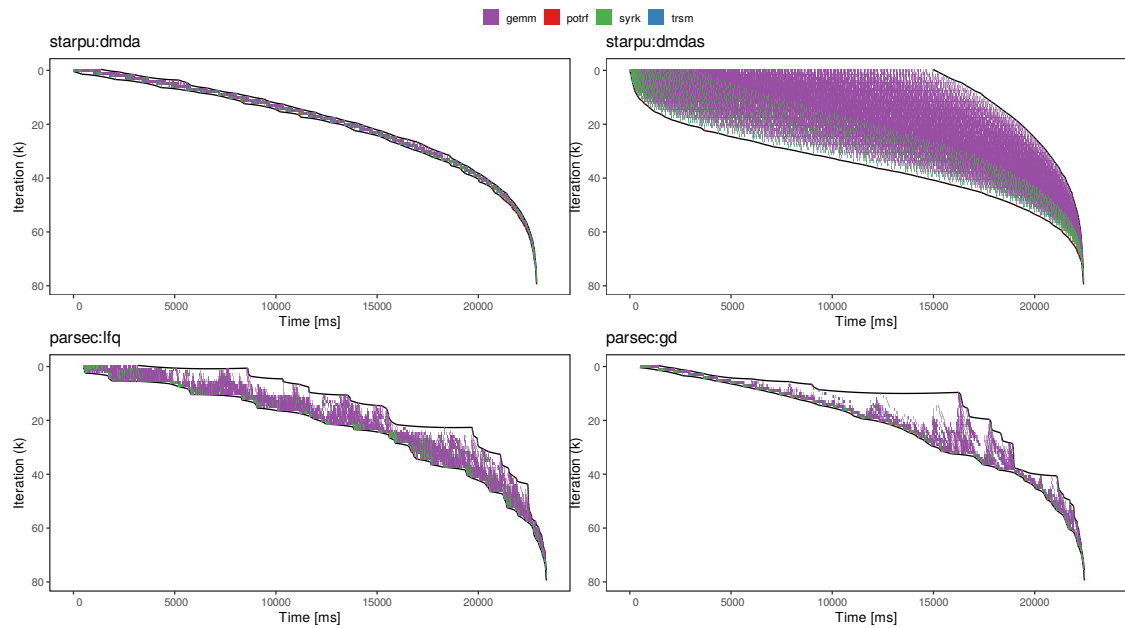


Figura 5. Progressão das iterações Cholesky ao longo do tempo, na poti.

5. Conclusão

Este trabalho comparou o desempenho dos runtimes StarPU e PaRSEC nas fatorações de Cholesky e LU sem pivotamento da biblioteca Chameleon, mantendo fixa a biblioteca de álgebra linear para isolar o efeito do runtime sobre o desempenho. Para viabilizar essa comparação, estendemos a interface DTD do PaRSEC com suporte à execução de tarefas em GPU e implementamos os codelets de GPU correspondentes no Chameleon, capacidades até então restritas ao caminho do StarPU.

Os resultados mostram que o StarPU é superior ou equivalente na maior parte dos casos, e que a exceção depende do equilíbrio entre a capacidade de CPU e de GPU de cada máquina. Na poti, cuja RTX 4070 (455 GFLOPS em FP64) vale pouco mais da metade da capacidade agregada da CPU, o `parsec:gd` obteve vantagem modesta, de 4% a 8% para $N \geq 20.000$, por despachar a GPU de forma oportunista, sem reservar threads de CPU para gerenciá-la. Na tupi, cuja RTX 4090 (1.290 GFLOPS) vale quase o dobro da CPU, o StarPU venceu em toda a faixa, com cerca de 30% de vantagem em $N = 40.000$ mesmo no tamanho de bloco ótimo de cada runtime. A análise dos rastros de execução mostrou que a inversão não se decide pela eficiência com que cada runtime executa uma tarefa na GPU: na poti, as quatro configurações extraem essencialmente o mesmo desempenho do dispositivo, e o que as separa é o quanto sobra de CPU. O StarPU dedica uma thread por *worker* CUDA à gestão da GPU, o que, na poti, encarece em cerca de 25% cada tarefa executada em CPU, e recupera esse custo apenas quando a capacidade da GPU é grande o bastante para justificá-lo.

Como trabalho futuro, pretendemos: (i) instrumentar o PaRSEC para medir a duração de tarefas de GPU diretamente no dispositivo, em vez de no *host*, eliminando a sobreposição entre *streams* que hoje impede comparar durações de tarefas de GPU entre os dois runtimes; (ii) estender a avaliação a um espectro maior de relações entre capacidade de CPU e de GPU, com outras GPUs e CPUs, para caracterizar com mais precisão o limiar em que a vantagem se inverte entre os runtimes; (iii) avaliar o desempenho em ambientes distribuídos multi-nó e multi-GPU, nos quais o escalonamento totalmente distribuído do PaRSEC foi projetado para ter vantagem sobre o modelo de gerenciamento por nó do StarPU; e (iv) estender as extensões de GPU desenvolvidas na interface DTD do PaRSEC a outras fatorações do Chameleon, como QR.

Acknowledgments

Todos os experimentos deste trabalho foram realizados na plataforma PCAD, <http://pcad.inf.ufrgs.br>, do INF/UFRGS. Este trabalho foi parcialmente financiado pela Petrobras, por meio de um acordo com a UFRGS, com os projetos 2020/00182-5 e 2018/00263-5. Este estudo foi financiado pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brasil - Código de Financiamento 001.

Referências

- Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Langou, J., Ltaief, H., and Tomov, S. (2011). Lu factorization for accelerator-based systems. In *2011 9th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA)*, pages 217–224.
- Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., and Tomov, S. (2012). Chapter 34 - a hybridization methodology for high-performance linear algebra

- software for gpus. In mei W. Hwu, W., editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 473–484. Morgan Kaufmann, Boston.
- Agullo, E., Beaumont, O., Eyraud-Dubois, L., Herrmann, J., Kumar, S., Marchal, L., and Thibault, S. (2015). Bridging the gap between performance and bounds of cholesky factorization on heterogeneous platforms. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 34–45.
- Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.
- Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemariner, P., and Dongarra, J. (2011). DAGuE: a generic distributed DAG engine for high performance computing. In *Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011 Workshops)*, pages 1151–1158, Anchorage, Alaska, USA. IEEE.
- Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. J. (2007). A class of parallel tiled linear algebra algorithms for multicore architectures. *CoRR*, abs/0709.1272.
- Garcia Pinto, V., Mello Schnorr, L., Stanisic, L., Legrand, A., Thibault, S., and Dan-jean, V. (2018). A visual performance analysis framework for task-based parallel applications running on hybrid clusters. *Concurrency and Computation: Practice and Experience*, 30(18):e4472.
- Hoque, R., Herault, T., Bosilca, G., and Dongarra, J. (2017). Dynamic task discovery in parsec: a data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, Scala '17*, New York, NY, USA. Association for Computing Machinery.
- Lacoste, X., Faverge, M., Bosilca, G., Ramet, P., and Thibault, S. (2014). Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 29–38.
- Lisito, A., Faverge, M., Kuhn, M., Pruvost, F., and Ramet, P. (2025). Scalable and portable lu factorization with partial pivoting on top of runtime systems. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1011–1022.
- Pei, Y., Bosilca, G., and Dongarra, J. (2022). Sequential task flow runtime model improvements and limitations. In *2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, pages 1–8.
- Pinto, V. G., Nesi, L. L., Miletto, M. C., and Schnorr, L. M. (2021). Providing in-depth performance analysis for heterogeneous task-based applications with starvz. *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 16–25.
- Thibault, S. (2018). *On Runtime Systems for Task-based Programming on Heterogeneous Platforms*. Habilitation à diriger des recherches (hdr), Université de Bordeaux. HAL Id: tel-01959127.