Análise da Influência do *Runtime* OpenMP no Desempenho de Aplicação com Tarefas

Henrique C. P. da Silva[*], Marcelo C. Milletto[†], Vinicius G. Pinto, Lucas M. Schnorr

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

{hcpsilva,marcelo.miletto,vgpinto,schnorr}@inf.ufrgs.br

Resumo. *Nesse trabalho verificamos o comportamento de cinco* runtimes *numa aplicação com programação baseada em tarefas. Observamos discrepâncias na duração das tarefas em relação ao tempo total de duração de algumas versões, além de dificuldades no gerenciamento com número excessivo de tarefas nas versões LIBKOMP e KStar$_{StarPU}$. A versão StarPU manteve bom desempenho mesmo em cenários com grande número de tarefas.*

1. Introdução e Objetivos

Após décadas trabalhando em sistemas e em um ferramental a par da exponencial demanda por paralelismo, o desenvolvedor de sistemas de alto desempenho tem diversas escolhas de bibliotecas, de *runtimes* e de *APIs* para explorar a fim de utilizar efetivamente o *hardware* desses sistemas. Entretanto, a diferença entre essas é tal que não se trata mais da qualidade da implementação ditar o desempenho desses sistemas, mas sim a escolha de biblioteca utilizada pelo programador. Com isso, surge a questão de definir qual dessas possui melhor desempenho e, sobretudo, o porquê disso.

Podemos citar OpenMP [OpenMP Review Board 2015], a partir da versão 4, e StarPU [Augonnet et al. 2011] como ferramentas de programação paralela que suportam modelos mais sofisticados de paralelismo, como a programação baseada em tarefas. Nessa, a lógica de um programa é organizada em tarefas e o paralelismo é inferido implicitamente pelas dependências de dados entre cada uma das tarefas, o que se traduz em um grafo direcionado acíclico (*DAG*). Assim, o *runtime* se encarrega de escalonar e distribuir essas tarefas dentre seus *workers* respeitando as dependências declaradas.

Avaliamos o desempenho e comportamento de uma fatoração QR em blocos para matriz densa baseada no PLASMA [Buttari et al. 2009] com duas implementações, uma utilizando diretamente as diretivas de tarefas do OpenMP e outra nativa em StarPU. A primeira foi executada com os *runtimes* nativos OpenMP do GCC e do LLVM, o *runtime* da biblioteca LIBKOMP baseado em X-Kaapi e através da transpilação para StarPU via KStar. Logo, nossos objetivos foram comparar o **escalonamento** das tarefas, observar a **ociosidade** dos trabalhadores de cada *runtime* e identificar **anomalias** em quaisquer das bibliotecas avaliadas. O *companion* deste trabalho está publicamente disponível[1].

2. Metodologia de Coleta e Análise de Dados

Nessa seção abordamos sobre a metodologia experimental, assim como o ferramental de *software* utilizado para a análise dos dados coletados.

[*]Bolsa do Projeto da Petrobras;

[†]Bolsa do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPQ);

[1]https://gitlab.com/hcpsilva/companion-erad-2020

Projeto Experimental: A fim de analisar o impacto dos diferentes *runtimes*, definimos como fatores do projeto experimental o tamanho da **matriz** e o tamanho do **bloco** de cada tarefa. O tamanho do bloco varia em fatores de potências de 2 entre $[32, 512]$, resultando, assim, em cinco níveis, enquanto o tamanho da matriz foi fixado em 8192. Através da combinação fatorial completa desses parâmetros de execução [Jain 1991], observamos a tendência do impacto do número das tarefas geradas pela a aplicação. Cada combinação foi repetida 5 vezes para cada experimento em cada nó para a avaliação do *makespan* e 1 vez para a coleta do traço das execuções.

Manipulação de dados com ferramentas modernas de *data science*: Utilizamos a linguagem R em conjunto com o pacote tidyverse através de blocos de códigos embutidos em arquivos Org versionados na ferramenta `git`. Assim, construímos um caderno de laboratório (*labbook*) que centralizou o processo exploratório científico, as anotações decorrentes e todos os experimentos realizados, desde o projeto desses até o código de análise dos dados coletados. Dessa maneira assegura-se a reprodutibilidade e a verificação dos resultados obtidos [Stanisic et al. 2015] sem perder a flexibilidade necessária para que processos de desenvolvimento e de verificação ocorressem paralelamente.

Execução e Coleta dos Dados: A automação foi realizada com *scripts* Shell executados pelo gerenciador de *jobs* Slurm em cada nó de computação detalhado na Tabela 1. Devido a complexidade de dependências dos *runtimes*, desenvolvemos um *script* que abstrai detalhes da plataforma e permite a fácil reprodução de um experimento.

Exploração e Análise dos Dados: A integração das ferramentas previamente citadas no editor Emacs permitiu o uso de blocos de código de análise dos dados em R entrelaçados com notas sobre esses. Uma abordagem similar é a dos *Jupyter Notebooks*, que porém é mais restrita pelo versionamento limitado dos arquivos em *rich text*.

Visualização e Concepção do Artigo: Após análise das observações, as visualizações dessas constavam também nos mesmos blocos de código de análise embutidos no *labbook*. Com a criação das visualizações, utilizamos um arquivo Org secundário para escrever o artigo e exportá-lo para LaTeX através do sistema de exportação da ferramenta.

3. Resultados Experimentais e Observações

Nessa seção apresentamos, com auxilio de visualizações, os fenômenos observados na experimentação. Além disso, detalhamos as plataformas utilizadas.

Tabela 1. Configuração das plataformas utilizadas nos experimentos.

Nome	CPU	L1/L2/LLC	RAM
`draco`	2×8 Xeon E5 2640 v2 2.5GHz	32KB/256KB/20MB	64GB DDR3
`cei`	2×12 Xeon Silver 4116 2.1GHz	32KB/1024KB/16.5MB	93GB DDR4
`hype`	2×10 Xeon E5 2650 v3 2.3GHz	32KB/256KB/25MB	128GB DDR4

Configuração Experimental: Na execução da aplicação, usamos as plataformas descritas na Tabela 1 e as ferramentas da Tabela 2. Todas executam Debian (`10.2`) com *kernel* Linux `4.19.0-6`. O rastreamento da aplicação utilizando a biblioteca $\text{libgomp}_{\text{GCC}}$ foi realizado com a ferramente ScoreP `6.0` e, quando utilizada a biblioteca $\text{libomp}_{\text{LLVM}}$, rastreamos a aplicação com uma biblioteca própria utilizando chamadas conforme a especificação OMPT `4.5`. O rastreamento da biblioteca StarPU e e do compilador $\text{KStar}_{\text{StarPU}}$ foram realizados com a biblioteca FxT `0.3.5` e o rastreamento das versões LIBKOMP foi realizado pelo próprio *runtime*, que implementa chamadas à *API* OMPT.

Tabela 2. Características das versões executadas da aplicação.

Identificador	Fonte	ABI/API Utilizada	Versão
$\text{libgomp}_{\text{GCC}}$		OpenMP/GCC	`8.3.0`
$\text{libomp}_{\text{LLVM}}$		OpenMP/LLVM	`6.0.0`
$\text{KStar}_{\text{StarPU}}$	Diretivas	StarPU (*LWS scheduler*)	$\texttt{master}_{\texttt{bf6af54e57bad130}}$
$\text{LIBKOMP}_{\text{libgomp}}$		OpenMP/LIBKOMP-LLVM	$\texttt{master}_{\texttt{32781b6dab10b1b5}}$
$\text{LIBKOMP}_{\text{libomp}}$		OpenMP/LIBKOMP-GCC	$\texttt{master}_{\texttt{32781b6dab10b1b5}}$
StarPU	Nativo	StarPU (*LWS scheduler*)	`1.3.1`

Diferenças de tempo de execução dos *kernels* em função do *runtime*: Executamos as versões das aplicações descritas na Tabela 2 coletando os tempos de execução que são apresentados nos gráficos da Figura 1. Nas colunas, observamos os diferentes tamanhos de bloco de cada tarefa e na linhas observamos as diferentes máquinas utilizadas no experimento. Apresentamos o valor médio de tempo das 5 observações e seu erro padrão.

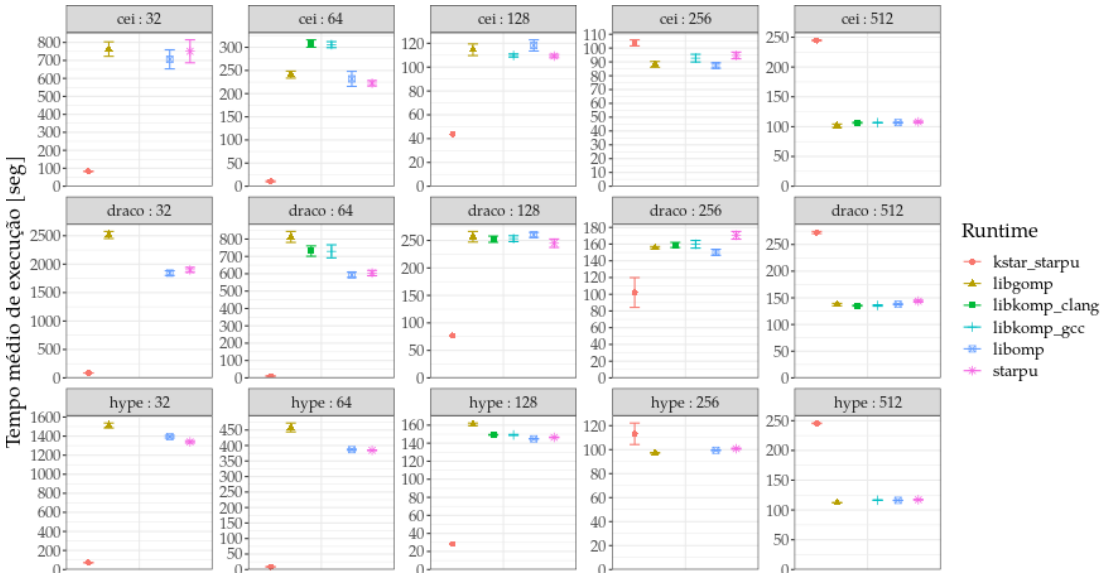

Figura 1. Comparação do *makespan* da execução de cada runtime.

Observamos que a tendência de comportamento se preserva entre as plataformas utilizadas. Sendo assim, analisamos nas próximas seções os detalhes da execução das tarefas na plataforma `cei` com o tamanho 64 de bloco.

Análise de Ociosidade por Trabalhador: Na Figura 2 observamos que, além do caso de tamanho 64, os *runtimes* mantém ociosidade similar para todos os casos. Com um tamanho de bloco 64, um grande número de tarefas é criado, o que estressa a capacidade de escalonamento dos *runtimes*, e, por consequência, o tempo ocioso por *worker* da plataforma pode refletir isso. Pela análise da figura, observamos que esse caso é especialmente interessante, já que ambos $\text{LIBKOMP}_{\text{clang}}$ e $\text{KStar}_{\text{StarPU}}$ apresentam ociosidade média acima de 90% em todos os trabalhadores. Em todos os casos, $\text{libgomp}_{\text{GCC}}$, $\text{libomp}_{\text{LLVM}}$ e StarPU mantém performance compatível em questão da efetividade do escalonamento das tarefas.

Comparação do Escalonamento entre os três *runtimes*: A Figura 3 apresenta o tempo de início da tarefa `dgeqrt`, que é o primeiro procedimento do laço de execução de uma fatoração QR. Ao observar este tempo, verificamos a progressão da implementação no processo de fatoração e a eficiência desse. Com exceção do $\text{KStar}_{\text{StarPU}}$ e StarPU, todas as tarefas foram iniciadas no primeiro segundo de execução. Além disso, observamos que a implementação $\text{KStar}_{\text{StarPU}}$ iniciou suas tarefas quase instantaneamente, indicando

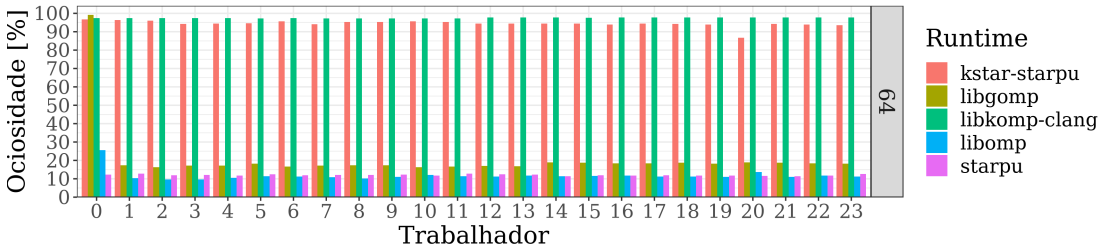

Figura 2. Comparação do *idleness* das tarefas em cada runtime e *worker*.

possível anomalia no rastreamento ou na implementação.

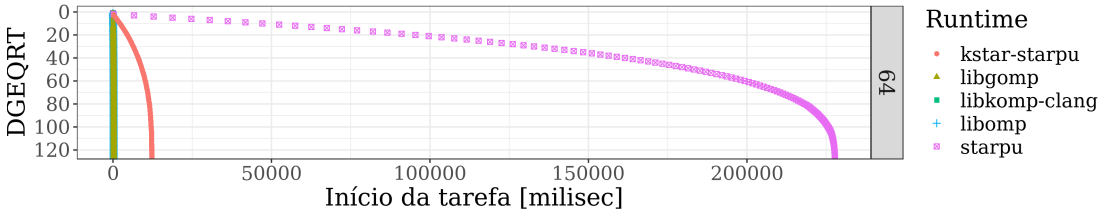

Figura 3. Comparação do escalonamento e progressão de cada *runtime*

4. Conclusão e Trabalhos Futuros

Nesse trabalho analisamos o desempenho e comportamento de cinco *runtimes* implementando uma fatoração QR utilizando tarefas. A partir dessas observações, identificamos que os tempos de duração das tarefas são incompatíveis com a taxa de ociosidade e tempo total observados, como é o caso das implementações $\text{libgomp}_{\text{GCC}}$, $\text{libomp}_{\text{LLVM}}$ e LIBKOMP. Além disso, verificamos que as ferramentas KStar e LIBKOMP não obtiveram desempenho desejável quando o grão de trabalho era pequeno, conforme detalhado na Subseção 3.3. Pela visualização do *makespan* da Figura 1 percebemos que existe um comportamento anômalo da ferramenta $\text{KStar}_{\text{StarPU}}$ em quase todos os casos, o que acreditamos indicar que esta versão não está respeitando as dependências de dados entre as tarefas. As constatações pedem pelo aprofundamento do trabalho, então consideramos implementar a verificação da solução obtida pela execução, adicionar o *runtime* OmpSs [Duran et al. 2011] nos testes e também abranger diferentes arquiteturas de processador.

Referências

Augonnet, C. et al. (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Conc. and Comp.: Pract. and Exp.*

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53.

Duran, A. et al. (2011). OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1st edition.

OpenMP Review Board (2015). OpenMP application program interface version 4.5.

Stanisic, L., Legrand, A., and Danjean, V. (2015). An effective git and org-mode based workflow for reproducible research. *SIGOPS Oper. Syst. Rev.*, 49(1):61–70.