Melhorando a Geração Automática de Código Paralelo para o Paradigma de Processamento de Stream em Multi-cores

Júnior Löff[1], **Dalvan Griebler**[1], **Luiz Gustavo Fernandes**[1]

[1] Escola Politécnica, Grupo de Modelagem de Aplicações Paralelas (GMAP), Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS), Porto Alegre, Brasil

junior.loff@edu.pucrs.com;dalvan.griebler,luiz.fernandes@pucrs.br

Abstract. *Parallel programming is still a challenge for programmers since it exhibits too many low level and operational system details. Programmers must deal with scheduling, load balancers and synchronizations. This work contributes with optimizations to a parallel programming abstraction for expressing stream parallelism in multi-cores. This work has extended SPar by including two new attributes to its language, and implementing new optimizations in its compiler for improving better performance to the parallel code automatically generated. Experiments revealed that our new SPar version can abstract parallelism details with similar performance to manually parallelized code.*

Resumo. *A programação paralela ainda é um desafio para desenvolvedores, pois exibe demasiados detalhes de baixo nível e de sistemas operacionais. Programadores precisam lidar com detalhes como escalonamento, balanceamento de carga e sincronizações. Esse trabalho contribui com otimizações para uma abstração de programação paralela para expressar paralelismo de stream em multi-cores. O trabalho estendeu a SPar adicionando dois novos atributos na sua linguagem, e implementou melhorias no seu compilador a fim de proporcionar melhor desempenho ao código paralelo gerado automaticamente. Os experimentos revelaram que a nova versão da SPar consegue abstrair detalhes do paralelismo com desempenho similar às versões paralelizadas manualmente.*

1. Introdução

Processadores *multi-core* têm se popularizado como uma alternativa viável frente a limitações físicas encontradas no aumento da frequência de processadores. Diferente da abordagem sequencial, a extração de desempenho nos *multi-cores* requer uma implementação específica, através do paradigma de programação paralela. Nesse paradigma, fica a cargo do programador fazer o melhor uso da arquitetura paralela para poder aumentar o desempenho. A popularização da programação paralela significaria o mesmo impacto na escrita de código que a programação orientada a objetos teve na programação sequencial.

Dessa forma, diversas metodologias e bibliotecas vem sendo propostas para melhorar a programabilidade no contexto de paralelismo. Na literatura, houve dois movimentos principais que descrevem metodologias de programação paralela: introdução de esqueletos algorítmicos [Cole 1989] e metodologias de programação paralela originárias da engenharia de software [Mattson et al. 2004]. Em meados de 2010, pesquisas e ideias dessas duas áreas convergiram na programação paralela estruturada [McCool et al. 2012].

Desde então, bibliotecas do C++ como OpenMP, Intel TBB, Microsoft PPL e C++17 Parallel STL têm convergido para essa metodologia. Cada uma dessas bibliotecas oferece aos programadores uma interface de programação (API) com padrões paralelos que abstraem complexidades do paralelismo, como escalonamentos, sincronizações e outros.

Por outro lado, escolher o melhor padrão paralelo ainda é um desafio e requer expertise do programador. De acordo com Griebler [Griebler 2016, Griebler et al. 2017a], essas bibliotecas ainda expõem demasiados detalhes paralelos ao programador. Como solução, Griebler apresenta um novo modelo de programação paralela mais abstrato. Para provar o conceito, desenvolveu-se a SPar [1], uma linguagem de domínio específico (DSL) do C++ para expressar paralelismo através de atributos do C++11. O programador utiliza os atributos para anotar o fluxo de dados no código. Nisso, a responsabilidade de geração de código paralelo eficiente é passada para o compilador da SPar.

A geração de código na SPar é limitada ao domínio de paralelismo de *stream*. Nesse domínio, trabalhos já mostraram que é possível aumentar a produtividade sem haver perdas significativas de desempenho [Griebler et al. 2017a, Griebler et al. 2017b, Griebler et al. 2018a]. No entanto, diversas aplicações de processamento de *stream* possuem regiões internas com intensa computação sobre dados, são exemplos: aprendizado de máquina, extração de recursos naturais (petróleo, minerais), monitoramento de incêndios florestais e outros. Nessas aplicações, o desempenho poderia aumentar cominando os padrões do paralelismo de *stream* com paralelismo de dados. O objetivo desse trabalho é investigar a SPar no domínio do paralelismo de dados e, também, combinações do paralelismo de *stream* com dados. Assim, a pergunta que esse trabalho tenta responder é: **A partir de uma única interface de programação paralela, é possível gerar código paralelo automático eficiente para diferentes tipos de paralelismo em multi-cores?**

As contribuições desse trabalho são descritas a seguir:

- **Extensão da linguagem SPar.** Aumentamos a expressividade da linguagem SPar adicionando dois novos atributos sem comprometer a simplicidade: Pure e Impure.
- **Novas definições e regras de transformação.** O objetivo é aumentar a inteligência do compilador para geração de código paralelo mais eficiente.
- **Novo algoritmo para o compilador.** Implementamos um novo algoritmo para geração de código paralelo automático com base nos principais padrões paralelos do paralelismo de *stream*, paralelismo de dados, ou ainda a combinação destes.
- **Avaliação de desempenho.** Executamos um conjunto de experimentos utilizando aplicações reais para avaliar a eficiência do novo algoritmo implementado na SPar.

O restante desse artigo está organizado da seguinte forma. A Seção 3 apresenta a proposta de extensão da linguagem SPar. Na Seção 4 são discutidos os resultados. Finalmente, a Seção 2 apresenta os trabalhos relacionados e a Seção 5 resume as conclusões e trabalhos futuros.

2. Trabalho Relacionados

A Tabela 1 compara os trabalhos relacionados. O trabalho de [Thies et al. 2002] introduz uma DSL chamada StreamIt, também composta por uma linguagem e compilador. En-

[1] https://gmap.pucrs.br/spar

tretanto, o StreamIt requer aprender uma nova sintaxe e linguagem baseada em Java, enquanto isso a SPar utiliza atributos da sintaxe padrão do C++. O trabalho [Astorga et al.] introduz o GrPPI (*Generic Reusable Parallel Pattern Interface*), que como o nome sugere, oferece uma API com padrões paralelos genéricos. Ou seja, o programador implementa os padrões (Map, Pipeline, etc.) apenas uma única vez e escolhe para qual biblioteca a ferramenta deve gerar código paralelo. Diferente da SPar, no GrPPI o programador é responsável por identificar o melhor padrão e implementá-lo manualmente.

Tabela 1. Comparação entre os trabalhos relacionados.

Trabalho	Tipo de Interface	Linguagem de Programação	Runtime	Arquitetura Alvo
StreamIt [Thies et al. 2002]	Linguagem de domínio específico externa	Java	Customizado	multi-cores e clusters
GrPPI [Astorga et al.]	Biblioteca paralela de templates C++	C++	FastFlow, TBB, OpenMP e C++ Parallel STL	multi-cores
OpenStream [Pop and Cohen 2013]	Diretivas de compilação pragma	C/C++	POSIX Threads	multi-cores
OmpSs [Duran et al. 2011]	Diretivas de compilação pragma	C/C++	Customizado	multi-cores, clusters e aceleradores
WindFlow [Mencagli et al. 2019]	Biblioteca paralela de propósito específico	C++	FastFlow	multi-cores e aceleradores
PiCo [Misale et al. 2018]	Linguagem de domínio específico do C++	C++	FastFlow	multi-cores
SPar [Griebler 2016]	Linguagem de domínio específico do C++	C++	FastFlow	multi-cores, clusters e aceleradores

Ambos OpenStream [Pop and Cohen 2013] e OmpSs [Duran et al. 2011] se baseiam em anotações de diretivas #pragma, que não são totalmente suportadas na árvore de sintaxes da linguagem C++. Diferente disso, a SPar utiliza anotações que seguem a sintaxe de duplos colchetes, descrita no padrão C++11. O WindFlow [Mencagli et al. 2019] apresenta uma API específica do domínio de *stream*, que requer conhecer os padrões paralelos oferecidos pela biblioteca e qual deles é melhor em determinada aplicação. Já o PiCo [Misale et al. 2018], apesar de ser uma DSL para Big Data, ainda possui sintaxe muito similar ao FastFlow, a principal diferença está na linguagem, que é mais intuitiva que outras ferramentas de Big Data.

3. Proposta de Extensão da Linguagem SPar

Essa seção apresenta as etapas de desenvolvimento desse estudo, desde o isolamento do problema até a implementação de uma solução. É importante mencionar que o grupo de pesquisa onde os autores atuam (GMAP [2]), já possui uma ferramenta interna para estudo-de-caso, a SPar [Griebler 2016]. Desta forma, iremos utilizar a infraestrutura da SPar para testar a nova geração de código. Além disso, como a SPar gera código intermediário para a biblioteca FastFlow, também iremos gerar código intermediário para essa mesma biblioteca. Futuramente, as técnicas discutidas nesse trabalho podem ser aplicadas em outras linguagens além da SPar. Ademais, a geração de código não é dependente do FastFlow. Portanto, é possível gerar código para outras bibliotecas como OpenMP e TBB.

3.1. Introdução a SPar

A SPar (*stream paralellism*) é uma linguagem de domínio específico (DSL) para expressar paralelismo de *stream* em arquiteturas *multi-core*. A linguagem foi proposta por Griebler

[2]https://gmap.pucrs.br

em sua tese de doutorado [Griebler 2016]. O trabalho apresenta uma manifestação em direção a uma maior abstração do paralelismo. Por exemplo, escolher qual o melhor padrão paralelo ou o escalonamento não são tarefas que o programador de aplicação deve ser responsável. A principal intensão é oferecer um conjunto de anotações da própria linguagem de programação (ex. atributos do C++11) que o programador utiliza para anotar o fluxo de dados da aplicação. Então, o compilador da SPar é encarregado de analisar as anotações e gerar o melhor padrão paralelo para aquela situação. A SPar oferece cinco atributos para expressar o *stream*/fluxo de dados no código: (1) **ToStream** indica onde começa e termina o *stream*; (2) **Stage** indica onde começa e termina um estágio/bloco sequencial do fluxo de dados; (3 e 4) **Input** e **Output** como o nome sugere, são as entradas e saídas de cada bloco de código; (5) **Replicate** é um atributo especial para replicar um bloco sequencial quando não há dependência de dados.

O Código 1 foi paralelizado com a SPar através da anotação dos atributos **ToStream** e **Stage** nas linhas 1 e 3. Com estes atributos, o programador informa que o primeiro laço (linha 2) representa um fluxo de *stream*. Assim, cada elemento gerado por esse laço (cada iteração) será computada por um bloco sequencial de processamento, representado pelo estágio e laço da linha 4. Note que as anotações do exemplo contém um dos atributos adicionados com esse trabalho, o **Pure**. Esse atributo adiciona a informação de que o bloco de processamento é puro. O compilador da SPar utiliza essa informação para gerar código paralelo mais eficiente.

```
1 [[spar::ToStream]]
2 for(long int i=0; i<SIZE; i+=1){
3  [[spar::Stage, spar::Pure, spar::Input(i), spar::Replicate()]]
4  for(long int j=0; j<SIZE; j++){
5   for(long int k=0; k<SIZE; k++){
6    matrix[i][j] += (matrix1[i][k] * matrix2[k][j]);
7 }}}
```

Código 1. Multiplicação de matrizes paralelizado com a SPar.

3.2. Aumentando a Expressividade da SPar

Como já foi mencionado, a SPar suporta apenas o domínio de paralelismo de *stream*, gerando código automático para os dois principais padrões: Pipeline e Farm. Durante o estudo, concluímos que esses atributos não são suficientes para expressar o paralelismo de dados. Em outras palavras, a SPar até consegue modelar aplicações do paralelismo de dados, mas nem sempre gera código eficiente - discussões mais profundas são apresentadas na Seção 4. O principal motivo é que os padrões Pipeline e Farm exibem características para modelar eficientemente aplicações com fluxos contínuos de dados. Já aplicações do domínio de dados são melhor modeladas por padrões como Map e MapReduce.

A seguir discutimos o estudo conduzido para avaliar a viabilidade de gerar código paralelo automático para o paradigma de paralelismo de dados, especificamente Map e MapReduce. O estudo envolveu duas etapas, onde a primeira considerou apenas a sintaxe e a segunda a semântica. Identificamos que os padrões Map e MapReduce podem ser localizados no código e estão associados com laços de repetição do C++. Entretanto, semanticamente ainda é um desafio descobrir se é possível gerar código paralelo. Por exemplo, caso exista dependência de dados, a geração de código paralelo automático produziria código incorreto.

De acordo com [Prema et al. 2019], os compiladores atuais para auto-paralelização ainda apresentam problemas de laços não identificados e geração de código semanticamente incorreto. Seguindo as sugestões desse trabalho, a alternativa é desenvolver ferramentas/compiladores baseados em informações inseridas pelo usuário. Assim, propomos a extensão da linguagem SPar para suportar dois novos atributos, que expressam informações do paralelismo de dados: (1) **Pure** significa que um laço de repetição é puro, ou seja, pode ser executado em paralelo já que não apresenta dependência de dados. (2) **Impure** pode ser utilizado para anotar uma região de código impura, a fim de purificá-la. Atualmente esse atributo resolve apenas reduções (intrínseco ao padrão MapReduce).

3.3. Geração de Código

Essa subseção apresenta em alto nível as modificações implementadas no compilador da SPar para geração automática de código paralelo de diferentes tipos do paralelismo. Com a inclusão dos novos atributos na SPar, implementamos um novo algoritmo e expandimos as definições e regras de transformação. A Figura 1 mostra as etapas modificadas no compilador para permitir a nova geração de código. (1) Na etapa *análise de semântica*, é executada uma varredura estática na árvore sintática abstrata (AST) do C++, gerada pelo compilador da SPar. O objetivo é extrair informações das anotações do programador e relacioná-las entre si e com o restante do código sequencial. Nessa etapa, adicionamos os novos atributos na linguagem da SPar e implementamos as devidas validações. (2) Na etapa *regras de transformação*, modificamos a tabela de definições e criamos novas regras de transformação para convergir no padrão paralelo que melhor modela a aplicação anotada. (3) Na etapa *extração de informações*, tentamos localizar as informações dos padrões Map e MapReduce seguindo a ISO padrão do C++. Por exemplo, valor de início e fim do conjunto de dados, variável de redução e outros. (4) Por fim, na etapa *geração de código*, utilizamos as informações extraídas anteriormente e implementamos a geração de código dos padrões de acordo com a sintaxe do C++ e do FastFlow. Nessa parte, implementamos uma nova estratégia de redução baseada em sobrecarga de operadores (*operator overloading* do C++).

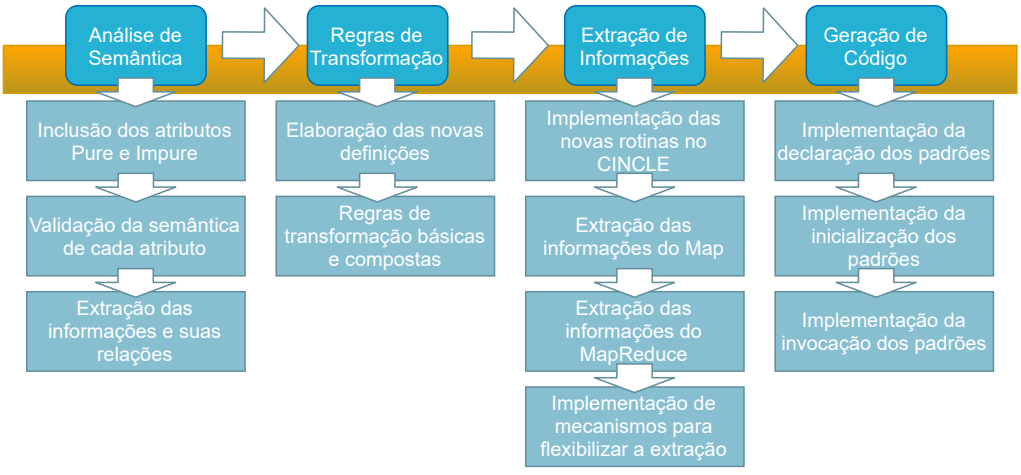


Figura 1. Fluxo das etapas de implementação no compilador da SPar.

4. Experimentos

Os experimentos foram executados utilizando aplicações reais. Os testes foram divididos em duas partes, onde primeiro avaliamos o paradigma de paralelismo de dados e após avaliamos a composição de diferentes paradigmas do paralelismo. A Seção 4.1 avalia o cenário de paralelismo de dados com o NAS Parallel benchmarks (NPB). O NPB contém oito *benchmarks* extraídos do domínio de dinâmica dos fluídos computacional (CFD). A importância do NPB está na sua variada carga computacional, pois as aplicações possuem diferentes métodos de acesso à memória, sobrecarregam diferentes sub-sistemas de *hardware* (por exemplo, unidades de ponto flutuante e *caches*) e apresentam complexas dependências entre dados. Nesse trabalho, focamos em seis *benchmarks*.

A Seção 4.2 apresenta um estudo exploratório da composição de padrões paralelos de *stream* com dados. As aplicações utilizadas para obtermos uma primeira impressão do comportamento da composição de padrões são o Mandelbrot Set e Lane Detection. O Mandelbrot Set é uma aplicação que pertence ao conjunto de visualizações matemáticas. Utiliza-se um fractal no plano complexo para determinar uma cor RGB para cada pixel baseado em sua localização. Em veículos autônomos, identificar a pista depende de imagens capturadas pela câmera. O Lane Detection é uma aplicação utilizada para processar essas imagens. Através das imagens capturadas pelas câmeras, ele detecta os limites da pista utilizando o detector de bordas Canny e a transformada de Hough. Exemplos de saída das aplicações estão representados na Figura 2.

(a) Mandelbrot Set **(b)** Lane Detection

Figura 2. Resultados das aplicações Mandelbrot Set e Lane Detection.

4.1. Avaliando o Paralelismo de Dados

Os experimentos foram executados em uma máquina equipada com 64 GB de memória RAM e um processador Intel(R) Xeon(R) Silver 4108 CPU @ 1.80GHz que possui 8 núcleos físicos com suporte *hyper-threading*, totalizando 16 *threads*. Utilizamos uma máquina com um único processador para evitar o gargalo de comunicação entre processadores. Foi utilizado o sistema operacional Ubuntu 18.04 com *kernel* 4.15.0-123-generic. Nós utilizamos GCC 7.5 com a *flag* de otimização -O3. A versão do FastFlow foi v3.0.0. Nos testes, utilizamos a classe B do NPB, onde os parâmetros podem ser consultados no site oficial [3]. Os testes foram executados de 1 até o grau máximo de paralelismo. A execução foi repetida 5 vezes e os gráficos representam a média desses valores, com desvio padrão representado através de barras de erros em cima das colunas.

[3]https://www.nas.nasa.gov/publications/npb_problem_sizes.html

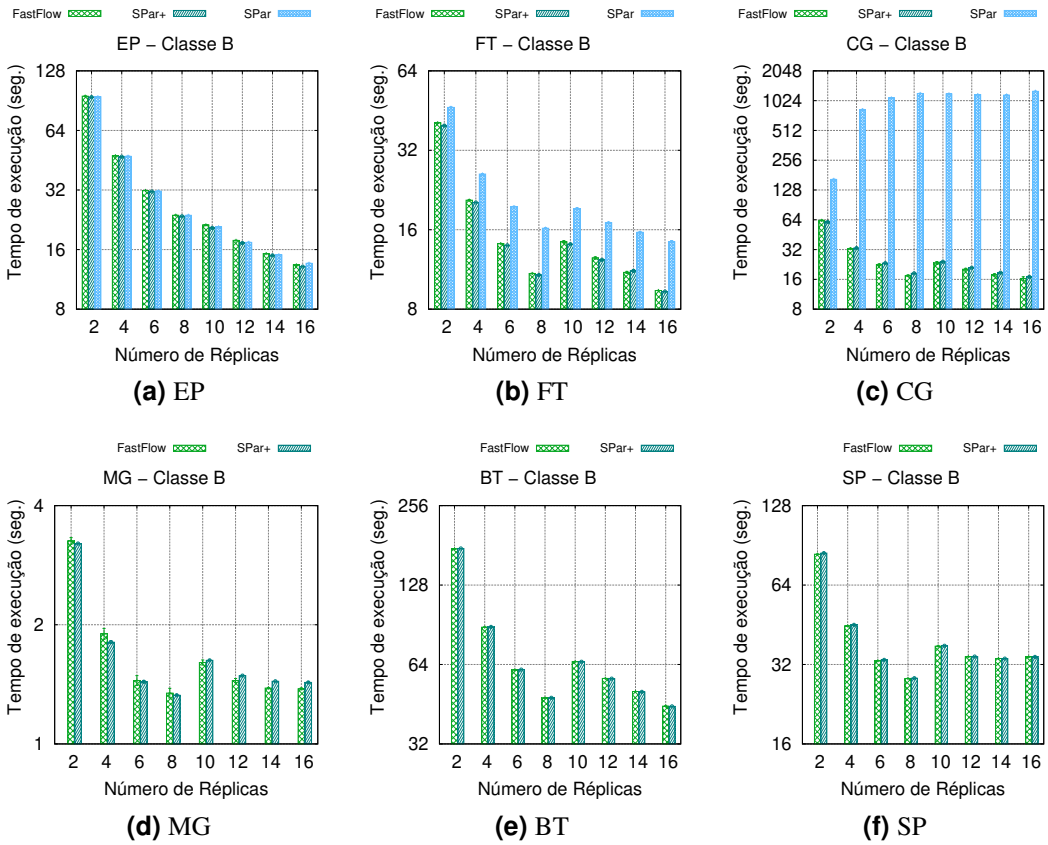


Figura 3. NPB com FastFlow [Griebler et al. 2018b] vs. SPar e SPar+.

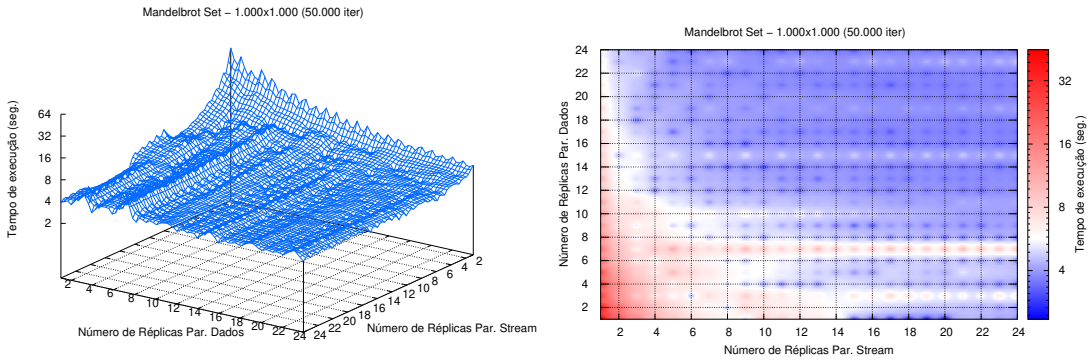
Os gráficos da Figura 3 resumem os resultados. O eixo x mostra o número de réplicas (*threads* responsáveis pela computação intensa) e o eixo y o tempo de execução em segundos (escala logarítmica). Comparamos a SPar original e a nova versão SPar+ estendida nesse trabalho, com uma versão manualmente otimizada do FastFlow obtida de [Griebler et al. 2018b]. Na Figura 3a as diferenças não são significativas, visto que o EP é um *kernel* simples com um único laço de repetição. Nos gráficos das Figuras 3b e 3c é possível perceber que o paralelismo gerado automaticamente pela SPar não é eficiente. O principal problema é que o padrão paralelo gerado pela SPar centraliza o emissor em uma única *thread*. Como as tarefas possuem pouca carga computacional, as *threads* de computação terminam rapidamente e ficam ociosas até receber outra tarefa. Note que o tempo aumenta com mais réplicas, já que há mais *threads* esperando e o tempo para receber uma tarefa aumenta.

Não foi possível implementar as últimas três aplicações com a SPar. Os principais motivos são limitações na paralelização, como: como refatoração de código, captura de variáveis e operações de redução. Em relação a SPar+, paralelizamos os quatro kernels e duas pseudo-aplicações. O resultado das Figuras 3a, 3b, 3e e 3f é muito similar entre a SPar+ e a versão FastFlow. O principal motivo é que o código gerado possui exatamente o mesmo número de laços paralelos. A Figura 3c e a Figura 3d, referentes ao CG e MG, apresentam diferenças maiores. A diferença no MG é que a SPar+ implementa um MapReduce a menos que a versão FastFlow. O motivo, é que atualmente a SPar+ identifica operações de redução somatórias, enquanto que o MG implementa uma operação do tipo

`max`. No CG, a principal diferença é que no FastFlow o escalonamento não é estático, mas dinâmico, com tamanho obtido através de tentativa e erro. Na SPar+ não é possível prever esse número de escalonamento, portanto, no CG a divisão estática apresenta maior desbalanceamento de carga. Em todos os gráficos (exceto da Figura 3a) observamos que o tempo de execução aumenta na transição de 8 para 10 *threads*. O motivo é a tecnologia *hyper-threading*, que pode introduzir desbalanceamento de carga.

4.2. Avaliando o Paralelismo de Stream combinado com Paralelismo de Dados

Essa seção avalia a combinação de diferentes paradigmas do paralelismo. Devido a natureza das aplicações, é conhecido que o desempenho pode diminuir adicionando um nível extra de paralelismo, principalmente por problemas de balanceamento e de granularidade muito fina. Mas o objetivo geral é investigar o comportamento e observar possíveis padrões e tendências, que podem trazer contribuições para essa área de pesquisa. Os experimentos foram executados em uma máquina equipada com 24 GB de memória RAM e dois processadores Intel(R) Xeon(R) CPUE5-2620 v3 @ 2.40GHz que possui 6 núcleos físicos cada e suporte *hyper-threading*, totalizando 24 virtuais. O sistema operacional era Ubuntu 18.04 com *kernel* 4.15.0-112-generic. Nós utilizamos GCC 7.5 com a *flag* de otimização -O3. A execução foi repetida 5 vezes e os gráficos representam a média desses valores. A carga de trabalho utilizada para configurar o Mandelbrot Set é uma imagem com dimensão 1.000x1.000 e limite de iterações 50.000. No Lane Detection, foram utilizadas 936 imagens do *dataset* KITTI 0101.

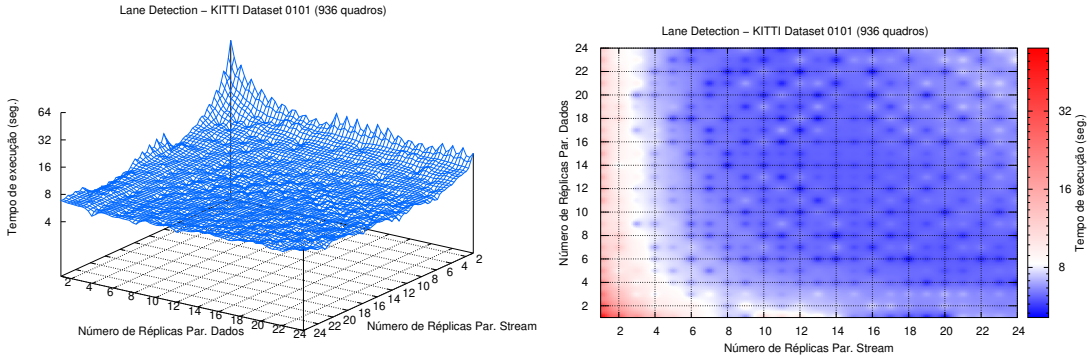


(a) Representação 3D **(b)** Mapa de calor

Figura 4. Desempenho da aplicação Mandelbrot Set na SPar+.

A Figura 4 apresenta os resultados obtidos com a aplicação Mandelbrot Set. Esse gráfico relaciona o número de réplicas do paralelismo de *stream* com o número de réplicas do paralelismo de dados. Foram testadas todas as configurações, onde o tempo de execução é representado no eixo z. Nos gráficos de mapa de calor, o tempo de execução é representado através de cores de intensidade, onde o vermelho representa tempos maiores e o azul menores. A primeira observação nessa aplicação é que o paralelismo de *stream* escala melhor que o de dados. De fato, o melhor resultado de tempo de execução foi obtido com 20 réplicas de *stream* e 1 réplica de dados. Outro comportamento interessante é que o paralelismo de dados limita o tempo total da aplicação, pois mesmo variando o grau de *stream*, o tempo de execução é o mesmo. Isto sugere desbalanceamento de carga.

Esse resultado é interessante, pois sugere que um segundo nível de paralelismo aninhado com o primeiro, poderia balancear a carga de aplicações nativamente desbalanceadas.



(a) Representação 3D **(b)** Mapa de calor

Figura 5. Desempenho da aplicação Lane Detection na SPar+.

A Figura 5 apresenta os resultados obtidos com a aplicação Lane Detection. Os gráficos foram elaborados da mesma forma que os gráficos da figura anterior. Novamente percebe-se que o paralelismo de *stream* escala melhor que o paralelismo de dados. Entretanto, agora o melhor resultado ocorreu com 14 réplicas de *stream* e 8 de dados. Diferente da aplicação anterior, os resultados não são limitados pelo paralelismo de dados. Pelo contrário, há pontos espalhados ao longo da malha com tempos de execução menores que seus vizinhos. Nessa aplicação, não se observou nenhum padrão que explica a diminuição do tempo de execução nesses pontos.

5. Conclusões

Esse trabalho apresentou uma extensão da linguagem SPar para melhorar a geração automática de código paralelo. Isso foi possível devido a adição de dois novos atributos, o Pure e Impure. Além disso, criamos novas definições e regras de transformação e implementamos um novo algoritmo para o compilador. Para testar a eficiência do novo algoritmo, executamos uma avaliação de desempenho utilizando aplicações reais. Os resultados mostraram que com as modificações, a nova versão da SPar é capaz de gerar código automático eficiente para os principais padrões paralelos de *stream* e de dados. Os resultados também mostraram que a geração de código paralelo automático obteve resultados similares à versões implementadas manualmente em aplicações do paralelismo de dados. Conclui-se que este trabalho de pesquisa contribuiu com avanços importantes para a área de programação paralela e todas as sub-áreas que dependem de códigos eficientes para obtenção de desempenho em arquiteturas *multi-core*.

Além disso, experimentos avaliando a composição de diferentes paradigmas de paralelismo mostraram que há oportunidades para explorar melhor essa estratégia. Possíveis trabalhos futuros são: (1) Avaliar a composição de padrões com outras aplicações, possivelmente das áreas de aprendizado de máquina, exploração de recursos naturais e processamento de imagens geo-espaciais. (2) Investigar a viabilidade de incluir novos padrões paralelos, além do Pipeline, Farm, Map e MapReduce. (3) Suportar a geração

de código paralelo automático para diferentes arquiteturas, por exemplo, *clusters* e *multi-cores*, ou *multi-cores* e *GPUs*.

Agradecimentos

Nós agradecemos pelos recursos computacionais do LAD-PUCRS, LARCC-Setrem e do GMAP. Essa pesquisa é parcialmente financiada pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001, pela FAPERGS 05/2019-PQG projeto PARAS (Nº 19/2551-0001895-9), Universal MCTIC/CNPq Nº 28/2018 projeto SPARCLOUD (Nº 437693/2018-0).

Referências

Astorga, D. d. R., Dolz, M. F., Fernández, J., and García, J. D. A generic parallel pattern interface for stream and data processing. *CCPE*, 29(24):e4175.

Cole, M. (1989). Algorithmic skeletons: Structured management of parallel computation.

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011). OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Process. Lett.*, 21(2):173–193.

Griebler, D. (2016). *Domain-Specific Language & Support Tool for High-Level Stream Parallelism*. PhD thesis, PPGCC - PUCRS, Porto Alegre, Brazil.

Griebler, D., Danelutto, M., Torquati, M., and Fernandes, L. G. (2017a). SPar: A DSL for High-Level and Productive Stream Parallelism. *PPL*, 27(01):1740005.

Griebler, D., Hoffmann, R. B., Danelutto, M., and Fernandes, L. G. (2018a). High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming*, 47(1):253–271.

Griebler, D., Hoffmann, R. B., Loff, J., Danelutto, M., and Fernandes, L. G. (2017b). High-Level and Efficient Stream Parallelism on Multi-core Systems with SPar for Data Compression Applications. In *WSCAD*, pages 16–27, Campinas, SP, Brasil. SBC.

Griebler, D., Loff, J., Mencagli, G., Danelutto, M., and Fernandes, L. G. (2018b). Efficient NAS Benchmark Kernels with C++ Parallel Programming. In *PDP*, PDP'18, pages 733–740, Cambridge, UK. IEEE.

Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition.

McCool, M., Robison, A., and Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science.

Mencagli, G., Torquati, M., Griebler, D., Danelutto, M., and Fernandes, L. G. L. (2019). Raising the Parallel Abstraction Level for Streaming Analytics Applications. *IEEE Access*, 7:131944 – 131961.

Misale, C., Drocco, M., Tremblay, G., Martinelli, A. R., and Aldinucci, M. (2018). Pico: High-performance data analytics pipelines in modern c++. *Future Generation Computer Systems*, 87:392 – 403.

Pop, A. and Cohen, A. (2013). Openstream: Expressiveness and data-flow compilation of openmp streaming programs. *ACM Trans. Archit. Code Optim.*, 9(4).

Prema, S., Nasre, R., Jehadeesan, R., and Panigrahi, B. K. (2019). A study on popular auto-parallelization frameworks. *Concurr. Comput. Pract. Exp.*, 31(17).

Thies, W., Karczmarek, M., and Amarasinghe, S. (2002). Streamit: A language for streaming applications. In Horspool, R. N., editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg. Springer Berlin Heidelberg.