VanC++: UMA INTERFACE GENÉRICA PARA PROGRAMAÇÃO PARALELA

MURILO F. SCHMALFUSS[1]; GERSON GERALDO H. CAVALHEIRO[2]
[1]*Universidade Federal de Pelotas – mfschmalfuss@inf.ufpel.edu.br*
[2]*Universidade Federal de Pelotas – gerson.cavalheiro@inf.ufpel.edu.br*

1. INTRODUÇÃO

Arquiteturas *multicore* apresentam uma alternativa para melhorar o desempenho de aplicações, seja esse desempenho relacionado ao tempo de processamento ou relacionado a uma melhor utilização de todos os recursos oferecidos pelos *hardwares* atuais.

Para atender a demanda dos programadores, diversas ferramentas foram desenvolvidas buscando solucionar de maneira eficiente a falta de soluções de alto nível em linguagens de programação. A partir da versão C++11, a linguagem oferece nativamente um conjunto de recursos para programação *multithreaded* (WILLIAMS, 2012). Entre outras ferramentas que podemos citar estão o padrão bastante difundido OpenMP (CHAPMAN, 2007), e também as soluções mais atuais como CilkPlus (BLUMOFE, et al., 1995) e TBB (REINDERS, 2007).

Nas ferramentas apresentadas podemos observar duas abordagens distintas, em C++11 o programador descreve a concorrência e o paralelismo de forma explícita. Nas demais ferramentas a concorrência é descrita de forma explícita, porém delegando ao ambiente de execução a responsabilidade pela exploração dos recursos paralelos disponíveis. Usualmente um mecanismo de escalonamento aplicado em nível de usuário realiza o mapeamento da concorrência, oferecendo ao programador uma abstração do *hardware* disponível.

A consequência da imposição de regras para o desenvolvimento de aplicações paralelas implica em uma restrição da capacidade de expressão da ferramenta. Uma realidade onde programas são compostos de diferentes áreas de paralelismo onde diferentes abordagens podem ser utilizadas para explorar a concorrência, uma abordagem interessante seria a adaptação do ambiente de execução para as necessidades do algoritmo, provendo uma portabilidade do código entre diferentes plataformas.

Neste trabalho é oferecida uma interface de programação única, sendo possível alterar apenas a estratégia de escalonamento aplicada. C++ foi escolhida como linguagem base, havendo o cuidado de manter inteira compatibilidade da interface proposta com os recursos de programação nativos.

2. METODOLOGIA

Focando na flexibilidade, reuso e compatibilidade de código a interface para programação paralela VanC++ propõe uma abordagem diferente, a descrição do paralelismo é desacoplada do mecanismo de escalonamento, permitindo ao programador implementar novos algoritmos de escalonamento que melhor se adaptem ao problema a ser resolvido.

Embora as diversas ferramentas possuam interfaces divergentes entre si, é comum a elas os conceitos de tarefas, *threads* e escalonamento. Estes conceitos são mapeados para classes genéricas em C++ de modo a permitir que programas concorrentes sejam implementados em termo das interfaces dessas abstrações.

A capacidade de expressão de uma ferramenta de programação concorrente depende de como ela suporta as entidades invocáveis da linguagem. Como um dos objetivos da interface desenvolvida neste trabalho é compatibilidade, todos os tipos de invocáveis são suportados.

A classe *Task* é a entidade responsável por representar invocáveis dentro da interface desenvolvida. Sendo assim, é extremamente importante que tal entidade possa ser capaz de executar o código invocável a qualquer momento no tempo. Afinal, o objetivo é que o código da aplicação execute em paralelo, porém, por diversas razões, uma execução concorrente não é determinística. A classe *Task* precisa, de alguma forma, dissociar o momento da execução de um invocável com sua execução de fato. Ocorre que este é um problema recorrente em programação, solucionado pelo padrão *Command*. Esse padrão armazena uma requisição para executar uma sequência de ações em um objeto, permitindo que essa sequência seja executada em um momento específico. A classe *Task* é a entidade que implementa o conceito de *Functor* (ALEXANDRESCU, 2011), de modo a executar estes invocáveis em paralelo. Neste modelo, o programador deve construir tarefas que serão passadas a outra entidade da interface, chamada *Thread* que representa uma execução concorrente.

A interface *Thread* representa um elemento de processamento e é responsável por executar as tarefas de forma concorrente. A classe *Thread* por representar o fluxo de execução concorrente, especifica dois métodos para serem implementados em sua interface. O operador operator()(), o qual recebe uma tarefa por referência, é responsável pela execução concorrente da mesma e o método join(), este método sinaliza o fim da execução e a sincronização.

Uma implementação da classe *Thread* mapeada para núcleos de um processador pode ser feita utilizando a classe std::thread definida no padrão C++11. Não só é possível ser compatível com os *threads* disponíveis na linguagem, como também há a possibilidade de o programador usar essa classe como base para sua própria implementação de *thread* e ainda poder utilizar objetos da classe *Task* como invocáveis.

No contexto desse trabalho, a classe fundamental *SchedulingPolicy* se encarrega de representar uma política de escalonamento genérica, cujo objetivo é servir de plataforma para interações entre tarefas e *threads* definidos pelo programador, tendo tais interações especificadas por um algoritmo de escalonamento, também providenciado pelo programador. Logo, para poder descrever a concorrência da aplicação, o programador deve, obrigatoriamente, parametrizar instâncias de *SchedulingPolicy* com um tipo que representa um algoritmo de escalonamento e, opcionalmente, com outro tipo que represente o *thread* implementado pelo usuário.

Além dessas classes fundamentais, é possível, ainda, expressar a concorrência da aplicação utilizando-se de abstrações de mais alto nível, encapsulados em padrões. Por serem implementados em termo destas classes básicas para expressão de concorrência, estes padrões permitem que o programador não tenha que se preocupar com a sincronização e a criação de *threads* diretamente, nem com seu escalonamento sobre os recursos de processamento disponíveis. Grande parte dessa flexibilidade proporcionada pela ferramenta advém do uso extensivo de técnicas de programação genérica combinadas com conceitos modernos de orientação a objetos.

Devido ao grande potencial de paralelização que laços apresentam, especialmente quando possuem poucas dependências de dados entre iterações, faz sentido tratar tal construção como um padrão. De maneira geral, caso existam poucas dependências entre as iterações, é possível paralelizar um laço for ao

recursivamente dividir seu espaço de iteração e atribuir os subespaços resultantes para *threads* que executarão concorrentemente. Os benefícios que podem ser observados nesta abordagem são a ausência de gerenciamento de *threads* por parte do programador, mesmo ele podendo selecionar qual elemento de processamento executará o padrão, e a possibilidade de otimizar a execução da aplicação por meio do algoritmo de escalonamento apropriado.

Uma operação de redução combina elementos de dados de modo a obter um resultado cujas dimensões são menores do que as dimensões da coleção de entrada. Para isso, esse padrão usa uma função combinatória binária que deve, obrigatoriamente, ser associativa. É esta propriedade que permite que a coleção seja reduzida recursivamente, visto que existe uma independência entre resultados parciais distantes. É justamente sobre a característica associativa da operação de redução que reside a possibilidade de paralelização deste padrão. O algoritmo de redução implementado pela interface toma vantagem da interface especificada pela classe std::vector de C++. Em especial, a implementação do algoritmo faz uso do tamanho da coleção, representado pelo método size() e do método que dá acesso aleatório a elementos dessa coleção, na forma do operador operator[](). Utilizando esses métodos, é possível calcular as partições necessárias que devem ser sucessivamente reduzidas e realizar o armazenamento dos resultados parciais. Essa necessidade de particionamento da coleção decorre das dependências da própria operação de redução. Caso o tamanho do nível seja 1, o primeiro elemento da coleção é retornado, representando a base e o resultado final da recursão.

3. RESULTADOS E DISCUSSÃO

Para a validação da interface foram escolhidos 4 problemas do *benchmark Cowichan* (WILSON; IRVIN, 1995), *Gaussian Elimination* e *Matrix-Vector Product* que são solucionados utilizando o padrão *for* paralelo e *Histogram Thresholding* e *Invasion Percolation* que são solucionados utilizando o padrão de redução paralelo.

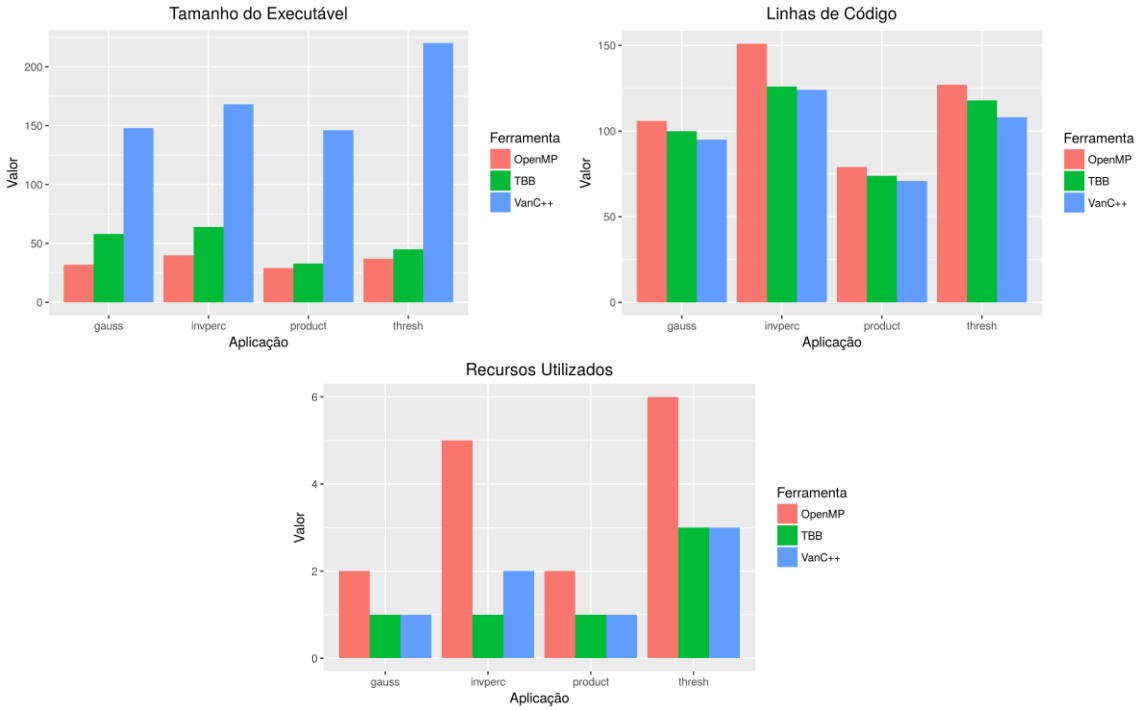

Figura 1: Comparativos entre as ferramentas utilizadas.

Para a coleta dos dados foi utilizado uma máquina com as seguintes configurações: processador Intel Core i5-5200U, sistema operacional Ubuntu 16.04 64 bits e compilador Intel icpc 17.0.3.

Na Figura 1, são apresentadas comparações entre as implementações. Em relação a OpenMP, o número de recursos se manteve acima da média geral, pois, embora de simples utilização, estão presentes em maior número, possibilitando maiores possibilidades com relação ao formato da aplicação. Contudo nesses casos não houve impacto significativo em relação ao número de linhas de código. Já em TBB e VanC++, observa-se um baixo número de recursos utilizados. Isso se deve a sua utilização mais complexa, porém mais abrangente em questão de possibilidades. Contudo a utilização destes recursos não impactou no número de linhas de código. O que variou bastante quando comparado as outras ferramentas, foi o tamanho do executável gerado.

4. CONCLUSÕES

Uma das funcionalidades mais importantes do padrão C++11 é o suporte a *multithreading*. Isto torna possível escrever programas C++ concorrentes sem depender de extensões específicas de plataforma, e assim, permite escrever código portável com comportamento garantido. Neste trabalho apresentou-se a especificação, implementação e validação de uma interface de expressão de paralelismo iterativo que explora e é compatível com as novas funcionalidades da linguagem de programação C++.

O resultado que se observa é a possibilidade da descrição do paralelismo de laços iterativos de forma simplificada. Isso é alcançado pelo uso dos padrões desenvolvidos para a descrição em alto nível de construções concorrentes.

Como trabalhos futuros, pretende-se estudar a construção de estratégias de escalonamento que utilizem a estrutura genérica provida pela interface desenvolvida para a implementação de programas paralelos.

5. REFERÊNCIAS BIBLIOGRÁFICAS

ALEXANDRESCU, A. **Modern C++ Design: Generic Programming and Design Patterns Applied**. Addison-Wesley, 2001.

BLUMOFE, R. D.; JOERG, C. F.; KUSZMAUL, B. C.; LEISERSON, C. E.; RANDALL, K. H.; ZHOU, Y. Cilk: Na Efficient Multithreaded Runtime System. **SIGPLAN Not**, New York, v.30, n.8, p.207-216, 1995.

CHAPMAN, B.; JOST, G.; PAS, R. **Using OpenMP: Portable Shared Memory Parallel Programming**. The MIT Press, 2007.

REINDERS, J. **Intel Threading Building Blocks**. O'Reilly, 2007.

WILLIAMS; A. **The C++ Concurrency in Action: Practical Multithreading**. Manning, 2012.

WILSON, G. V.; IRVIN, R. B. **Assessing and Comparing the Usability of Parallel Programming Systems**. 1995