# Implementação Paralela do LU no NPB C++ Utilizando um Pipeline Implícito

**Júnior Löff**[1], **Dalvan Griebler**[1], **Luiz G. Fernandes**[1]

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

junior.loff,dalvan.griebler@acad.pucrs.br,luiz.fernandes@pucrs.br

***Resumo.*** *Neste trabalho, um pipeline implícito com o padrão* $map$ *foi implementado na aplicação LU do NAS Parallel Benchmarks em C++. O LU possui dependência de dados no tempo, o que dificulta a exploração do paralelismo. Ele foi convertido de Fortran para C++, a fim de ser paralelizado com diferentes bibliotecas de sistemas multi-core. O uso desta estratégia com as bibliotecas permitiu ganhos de desempenho de até* $10.6\%$ *em relação a versão original.*

## 1. Introdução

A maior parte das aplicações encontradas na indústria e na academia não permitem uma implementação paralela simples. É necessário refatorar o código para remover a dependência de dados entre rotinas de computação. Um exemplo de aplicação com diversas regiões de intensa computação e dependentes no tempo é o LU (Lower-Upper Gauss-Seidel), que pertence ao conjunto NAS Parallel Benchmarks (NPB) [Bailey et al. 1994]. Esta aplicação resolve um sistema linear de equações utilizando uma variação do método de Gauss-Seidel, conhecida como método $SOR$ (*successive over-relaxation*). Este método é iterativo e requer a decomposição da matriz principal em duas matrizes triangulares: L (*lower*) inferior e U (*upper*) superior. A aplicação LU implementa o método $SSOR$ (*symmetric successive over-relaxation*), que funciona combinando duas execuções do algoritmo $SOR$. O LU computa de forma que cada iteração aproveita a computação anterior para melhorar a aproximação, gerando dependência entre as computações.

Nesta aplicação, é complexo implementar o paralelismo de dados puramente com o padrão `map`, visto que a dependência de dados limita a programação paralela e, consequentemente, o desempenho. Estudos que focaram na exploração do paralelismo da aplicação LU [Jin et al. 1999, Frumkin et al. 1998] descrevem que há dois métodos eficientes para implementar a aplicação: *hyperplane* e *pipelining*. A versão *pipeline* oferece vantagens visto que consegue utilizar melhor a distribuição de dados e possui melhor uso da memória cache [Jin et al. 2009]. A versão *hyperplane* faz uma abordagem matemática, onde todos os pontos de um mesmo *hyperplane* podem ser calculados independentemente.

A contribuição deste trabalho é a implementação paralela da aplicação LU do NPB utilizando diferentes bibliotecas de programação paralela em C++ voltadas para sistemas *multi-core*. A versão original implementa apenas o OpenMP, nós estendemos o paralelismo para as bibliotecas FastFlow, Thread Building Blocks (TBB) e Cilk++. Além disso, contribuímos com uma avaliação de desempenho. Esse estudo é a continuação da pesquisa publicada em [Griebler et al. 2018]. Portanto, o artigo tem como objetivo descrever a implementação do *pipeline* implícito utilizando o padrão `map`. Esta estratégia possibilitou estender o paralelismo para as demais bibliotecas. A conversão de Fortran para C++ do LU foi realizada para permitir o uso do OpenMP, FastFlow, TBB e Cilk++. O trabalho está dividido da seguinte maneira. A Seção 2 detalha a implementação do pipeline implícito. A Seção 3 discute os resultados e a Seção 4 apresenta as conclusões.

## 2. Implementação Paralela do Pipeline Implícito

É importante ressaltar que, apesar da implementação simular um *pipeline*, a aplicação LU não poderia ser implementada em um *pipeline* tradicional sem os devidos controles e gerenciamentos do fluxo de dados. Esta estratégia desenvolvida e aprimorada pelos autores, com base em [Jin et al. 1999], foi essencial para permitir a implementação do paralelismo nas quatro bibliotecas paralelas avaliadas neste trabalho. Anteriormente, a implementação paralela utilizando somente os padrões $maps$ e $reduces$ apresentou um desempenho baixo. Houve uma redução de apenas $15\%$ do tempo de execução em relação ao código serial em uma máquina com $12$ núcleos físicos.

Na aplicação LU, em cada iteração do método $SSOR$, as variáveis do sistema de equações são aproximadas, cada iteração precisa do resultado obtido anteriormente. Isso faz com que exista dependência de dados no tempo, limitando o potencial de paralelismo. Além disso, essa dependência entre os dados é encontrada em todos os níveis ($k$, $j$, $i$) de diversos laços de repetição ao longo do código, impedindo a adição de paralelismo até mesmo em regiões paralelas com menor aproveitamento da *cache*. Um trecho de código com padrões de acesso à dados, onde $\alpha$ recebe $-1$ ou $+1$, pode ser visto a seguir:

```
ldx[j][i][0][m]*v[k][j][i+α][0];
ldy[j][i][0][m]*v[k][j+α][i][0];
ldz[j][i][0][m]*v[k+α][j][i][0];
```

Esta estratégia pode ser nativamente modelada no OpenMP, devido ao suporte oferecido pela biblioteca. A implementação consiste em encapsular o método $SSOR$ em uma região paralela, que irá simular um *pipeline*. Posteriormente, os trechos de código com computação intensa são anotados com `pragma omp for` para computação paralela. O fluxo de computação é gerenciado através de filas bloqueantes que controlam as *threads* de acordo com o *status* da computação anterior. Além disso, periodicamente são executados `pragma omp flush` para sincronizar a visão da memória entre as *threads*. Na implementação para as demais bibliotecas, FastFlow, TBB e Cilk++, utilizamos a biblioteca Pthreads para obter acesso a mecanismos como espera e sinal condicionados.

O nosso mecanismo bloqueante difere da versão original da NASA que faz o controle através do índice `j`. Nós aprimoramos o gerenciamento utilizando o identificador da *thread*. Isto serve para diminuir o *overhead*, visto que agora são feitas `num_threads` trocas ao invés de `jend` trocas. Em resumo, implementamos quatro funções para controlar o fluxo de dados no *pipeline* implícito: `WAIT_SIGNAL_BUTS`, `GO_SIGNAL_BUTS`, `WAIT_SIGNAL_BLTS` e `GO_SIGNAL_BLTS`. Note que há duas variações do fluxo de controle (`wait_buts` e `wait_blts`), pois essas rotinas são opostas, onde `blts` faz acesso ordenado e `buts` faz acesso reverso na matriz, conforme é ilustrado na Figura 1 através dos dois laços do índice $k$.

O bloqueio das *threads* foi implementado logo após as rotinas de decomposição da matriz principal (identificadas pelas flechas de sincronização na Figura 1) em matrizes triangulares inferior (`jacld`) e superior (`jacu`) através das rotinas `WAIT_SIGNAL_BUTS` e `WAIT_SIGNAL_BLTS`. Sempre que uma *thread* termina sua computação, a rotina `GO_SIGNAL_BUTS` ou `GO_SIGNAL_BLTS` é chamada para liberar a computação da *thread* seguinte. Desta forma, o fluxo de execução da aplicação ocorre como mostrado na Figura 1, onde cada *thread* computa um pedaço dos dados no nível $j$. O paralelismo se dá quando cada parte do nível $j$ é computada em um avanço do nível $k$. Como a computação é realizada sobre as matrizes triangulares, apesar da primeira *thread* computar antes, ela terá mais computação que a última *thread*, o que ajuda no balanceamento de carga.

Apesar da aplicação LU ser do domínio de paralelismo de dados, a estratégia uti-

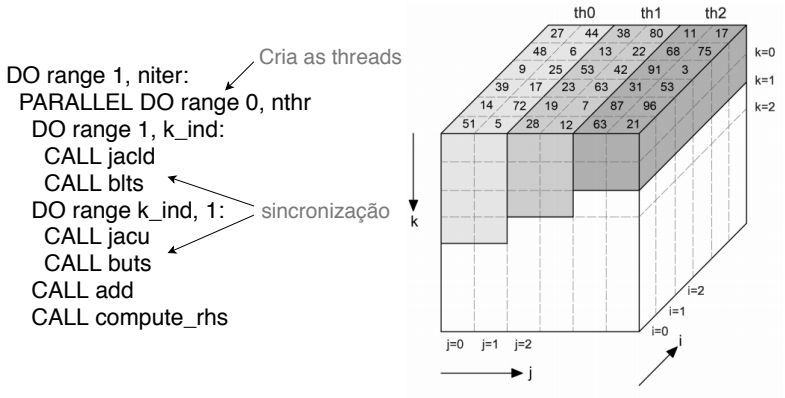


**Figura 1. Descrição do algoritmo e fluxo de execução da aplicação LU**

lizada foge da tradicional implementação $map$. A nossa estratégia utiliza o padrão $map$ somente para criar *threads* iguais à `num_threads`, ao invés de anotar um laço para execução paralela. Ou seja, o `parallel for` irá simular uma região paralela similar à do OpenMP, atribuindo um identificador único `thread_id` para cada *thread*. O gerenciamento e divisão dos dados é processado na região privada de cada *thread*. Isto pode impactar no balanceamento de carga, visto que cada biblioteca possui sua própria otimização na divisão dos dados e que foi substituída na nossa estratégia por uma implementação manual de escalonamento. Durante a implementação, as bibliotecas TBB e Cilk++ apresentaram limitações quanto ao gerenciamento das *threads* devido as abstrações. Como implementamos a criação manual das *threads* com identificador único, o TBB e Cilk++ puderam ser utilizados para paralelizar o LU com o *pipeline* implícito.

## 3. Resultados

Os experimentos foram executados em uma máquina equipada com dois processadores Intel(R) Xeon(R) CPU E5-2620 v3 2.40GHz, totalizando 12 núcleos físicos e 24 *threads* além de 24 GB de memória RAM. O sistema operacional era Ubuntu Server 64 bits com kernel 4.4.0-59-generic. Os programas foram compilados com a versão 7.0 dos compiladores `g++` e `gfortran` utilizando a *flag* de otimização `-O3`. Os testes foram repetidos 5 vezes para cada amostra, onde foi utilizada a média aritmética e calculado o desvio padrão que está plotado no gráfico. Os testes foram executados nas classes B e C, com 250 iterações e matriz de dimensões 102x102x102 e 162x162x162, respectivamente. Os gráficos apresentam o tempo de execução em segundos em relação ao grau de paralelismo.

A versão nomeada de `Original` foi implementada pela NASA em Fortran, as demais versões apresentadas nos gráficos foram implementadas em C++ pelos autores do trabalho. É possível perceber que as versões TBB, FF e CILK que implementam a estratégia do *pipeline* implícito tem desempenho comparado com a biblioteca OpenMP que apresenta suporte nativo para *pipelining*. Na Figura 2 é possível ver que o desempenho entre as bibliotecas é similar. Em 12 *therads* há o fim dos núcleos físicos e início da região de *hyperthreading*. Nesta região as versões FF e OMP apresentam melhor desempenho em relação à versão `Original`. No grau máximo de paralelismo, em 24 *threads*, a versão FF apresenta o melhor desempenho, seguido da versão OMP. Novamente ambas apresentam desempenho superior a versão `Original`. As versões TBB e CILK estão próximas, no entanto apresentam maior *overhead* por conta do escalonamento e *thread affinity* destas bibliotecas. Na Figura 3 vemos que a versão FF apresenta desbalanceamento de carga. Em 4 *threads*, o tempo de execução da versão FF chega próximo de 500

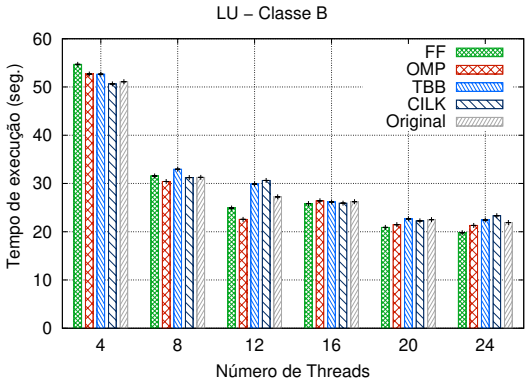


**Figura 2. Aplicação LU na Classe B.**

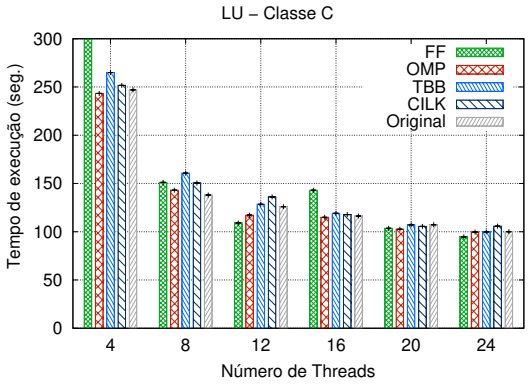


**Figura 3. Aplicação LU na Classe C.**

segundos. Isto é explicado pelo aumento no número de *cache misses*, que contabilizaram o dobro das demais versões. Por conseguinte, no grau máximo de paralelismo, com todas as 24 *threads*, as versões encontram-se com resultados similares. No entanto, a versão FF apresenta melhor desempenho, devido principalmente à inserção manual do *pinning* das *threads* em afinidade com núcleos próximos entre si. O CILK tem escalonamento e mecanismo de *thread affinity* próprios que apresentam *overhead* para cargas de trabalho maiores, o que explica o pior desempenho com 24 *threads*.

## 4. Conclusões

Este trabalho apresentou o estudo sobre uma estratégia de paralelismo na aplicação LU. A estratégia implementa um *pipeline* implícito utilizando o padrão paralelo $map$, o que possibilitou a implementação no FastFlow, TBB e Cilk++. Os resultados demonstram que a implementação tem desempenho superior a versão original. Nossa melhor implementação paralela, que foi a versão utilizando a biblioteca FastFlow, superou o tempo de execução da versão Original em $10.6\%$ na classe B e $5.6\%$ na classe C. Como trabalhos futuros, pretende-se avaliar as demais aplicações do NPB, cujo trabalho já está em andamento.

## Referências

Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. (1994). The NAS Parallel Benchmarks. Technical report, NASA Ames Research Center, Moffett Field, CA - USA.

Frumkin, M. A., Jin, H., and Yan, J. (1998). Implementation of the NAS Parallel Benchmarks in High Performance Fortran. Technical report, NASA.

Griebler, D., Loff, J., Mencagli, G., Danelutto, M., and Fernandes, L. G. (2018). Efficient nas benchmark kernels with c++ parallel programming. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 733–740.

Jin, H., Frumkin, M., and Yan, J. (1999). The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report, NASA.

Jin, H., Hood, R., and Mehrotra, P. (2009). A practical study of upc using the nas parallel benchmarks. In *Proceedings of PGAS*, PGAS '09, pages 8:1–7.