NeurOMP: Paralelização automática de código utilizando Aprendizagem por Reforço

João Saffran[1], **Luís Fabrício W. Góes**[1]

[1]Departamento de Ciência da Computação – Pontifícia Universidade Católica de Minas Gerais

joao.saffran@sga.pucminas.br, lfwgoes@pucminas.br

Abstract. *Building high performance applications is a daunting task. The programmer must be aware of several application development components, including parallelism techniques. Therefore modern compilers try to modify applications code to explore parallelism automatic, using static code analysis. One way to improve this strategy is using Reinforcement Learning (RL). This paper purpose a new compiler transformation, called NeurOMP, which uses RL to generate parallel code automatically using the OpenMP library. Experimental results shown that NeurOMP has an average speedup on CAP Bench of* 1.6*, similar to a human specialist.*

Resumo. *A criação de aplicações que obtenham o máximo de desempenho computacional nas arquiteturas modernas é uma tarefa complexa. Além de utilizar conhecimentos de paralelismo, o programador precisar ter um amplo conhecimento de vários outros aspectos da aplicação. Por este motivo, os compiladores modernos tentam paralelizar algoritmos de maneira automática, utilizando a análise estática do código. Uma maneira de melhorar o processo de tomada de decisão do compilador é utilizando Aprendizagem por Reforço (RL). Este trabalho propõe e avalia uma otimização de compilador, chamada NeurOMP, que utiliza RL para automatizar códigos em C utilizando a biblioteca OpenMP. Os resultados experimentais mostram que o NeurOMP obtém um speedup médio no CAP Bench de* 1.6*, similar a um especialista humano.*

1. Introdução

O paralelismo se tornou um componente essencial em sistemas de computação, aplicações e algoritmos modernos [Ahmed et al. 2017]. A criação de aplicações escaláveis que obtenham o máximo de desempenho exige que o desenvolvedor conheça amplamente da arquitetura, do hardware e do compilador que ele utiliza, além de envolver o conhecimento da aplicação, algoritmos e estruturas de dados envolvidas na solução do problema. Devido a esta complexidade no desenvolvimento de uma aplicação paralela, os compiladores vem se preocupando cada vez mais em otimizar o código fonte, visando evitar que o programador tenha que paralelizar o código explicitamente [Schardl et al. 2017] [Tong et al. 2017].

As otimizações do compilador envolvem várias transformações e mudanças no código de entrada, com o objetivo de remover, adicionar e alterar operações [Triantafyllis et al. 2003]. Um processo que os compiladores tentam automatizar é o processo de paralelização [Moreira et al. 2016]. Isto vem sendo amplamente pesquisado,

onde várias técnicas e ferramentas já foram desenvolvidas para tornar esse processo menos custoso para os programadores. Alguns exemplos são: i) criação de bibliotecas que ajudem o compilador no processo de paralelização, como por exemplo o OpenMP [Dagum and Menon 1998], juntamente com uma API que permite ao programador paralelizar o código utilizando diretivas de compilação; ii) transformações de código usadas para paralelizar o código através da análise estática do código [Bessey et al. 2010].

Estes avanços facilitam o processo de paralelização de código, mas ainda assim este problema não foi resolvido [Chapman et al. 2008]. Os conceitos envolvidos na paralelização de código são complexos para uma parte dos programadores. Um outro aspecto impactante é que essas bibliotecas necessitam que os programas sejam reescritos para utilizá-las no processo de paralelização. Este mesmo problema afeta o compilador na hora de realizar a análise estática. Para que o compilador obtenha a paralelização que causa a maior redução do tempo de execução, várias outras transformações intermediárias devem ser realizadas [Khatami 2017].

O compilador tem um outro empecilho quando realiza as transformações. Uma vez que um programa tem um comprometimento com a corretude e desempenho do código gerado, este fator força o compilador a se comportar de maneira conservadora. Um exemplo disso é quando o compilador faz a vetorização (processo de paralelização de instruções realizado pelo compilador) [Gubner and Boncz 2017]. Se o custo de um conjunto de instruções for baixo, o compilador pode não vetorizar essas instruções, acreditando que essa vetorização não garante nenhum ganho. A solução mais adotada para esse problema é permitir que o programador determine manualmente qual o custo mínimo para que a transformação ocorra [Coelli 1996].

Uma maneira de melhorar as otimizações do compilador seria utilizar algoritmos de inteligência artificial (IA) [Stephenson et al. 2003]. Apesar de não serem tão assertivas quanto a análise estática, estas técnicas ajudam no processo de tomada de decisão do compilador, tornando-o capaz de criar heurísticas mais eficientes que o ajuda a criar códigos com melhor desempenho computacional.

Um campo de IA promissor para ser aplicado dentro do processo de compilação é o Aprendizado por Reforço (RL) [Kulkarni and Cavazos 2012]. Este campo estuda a criação de agentes inteligentes capazes de tomar decisões em um ambiente, que pode ser estocástico ou determinístico, com o propósito de maximar o acúmulo de recompensa a longo prazo.

Uma das vantagens de RL é utilizar um modelo minimalista do mundo. Com um *Processo de decisão de Markov* [Davis 2018], um arcabouço matemático utilizado para modelar problemas de RL, é possível resolver problemas apenas observando as recompensas recebidas pelas ações tomadas pelo agente.

Um algoritmo capaz de fazer o acúmulo de recompensas em longo prazo é o *QLearning*. Este algoritmo descobre uma função que aprende uma distribuição de probabilidade para as ações do agente, sofrendo alterações dependendo do estado do agente e do mundo. Desta forma obtêm-se qual a melhor ação para cada estado do mundo.

O objetivo deste trabalho é a criação de uma otimização de compilador que utiliza aprendizagem por reforço para fazer a paralelização automática de código, *loops* do tipo *for*, utilizando *OpenMP*. Além de implementar a transformação, compara-se o resul-

tado desta otimização com as já existentes e com um especialista em paralelismo. Para mensurar isto, utiliza-se o *CAP Bench*, um *benchmark* [Souza et al. 2017] já paralelizado com *OpenMP* e, além disso, são comparados os processos de análise estática para geração automática de código dos compiladores ICC e GCC.

O restante deste trabalho está dividido em 7 seções. A Seção 2 discute os trabalhos relacionados. Seções 3 e 4 explicam conceitos de Computação Paralela e Aprendizagem por reforço, respectivamente. O processo utilizado pela otimização é apresentado na Seção 5. A Seção 6 apresenta os resultados experimentais. Por fim, na Seção 7 são apresentadas as conclusões deste trabalho.

2. Trabalhos Relacionados

Nesta seção, são discutidos os trabalhos relacionados. Os conceitos abordados nessa seção são: Aplicações de Aprendizagem de Máquina dentro do processo de compilação, técnicas de paralelização automática de código e aplicações de aprendizagem por reforço.

Devido à constante evolução das arquiteturas dos computadores modernos, definir os parâmetros usados nas otimizações internas do compilador se torna uma tarefa complexa para ser realizada por seres humanos. Por isso, o trabalho [Fursin et al. 2008] propõe o uso de Aprendizagem de Máquina (AM) para que o compilador aprenda automaticamente quais os melhores parâmetros para as heurísticas utilizadas no compilador para cada arquitetura.

No trabalho [Moreira et al. 2016], os autores utilizam análise estática para realizar a paralelização do código de maneira automática utilizando diretivas do *OpenACC*. Este trabalho não utiliza ML para tentar paralelizar o código, ao invés disso, são apresentadas várias regras criadas manualmente para paralelizar o código.

Em [Coons et al. 2008], os autores utilizam aprendizagem por reforço para alocar instruções e dados dentro de vários processadores. O trabalho apresenta uma utilização de RL dentro do contexto de otimizações de compiladores, porém o que os autores tentam reduzir o tempo gasto em comunicação.

No trabalho do *DeepMind* [Mnih et al. 2013], os autores usam RL de uma maneira mais tradicional. Eles visam criar um agente inteligente capaz de jogar diversos jogos de *Atari*. Apesar de utilizar aprendizagem por reforço, o trabalho foca em um campo distante da paralelização automática de código.

No artigo [Marinkovic et al. 2018], os autores propõem um algoritmo de paralelização automática de código em nível de instrução. O trabalho foca na paralelização automática de código, mas em nível de código de máquina binário.

Este trabalho se difere dos anteriores, pois utiliza aprendizagem por reforço como mecanismo para obter a paralelização de um determinado código, utilizando as diretivas *OpenMP*. Já os trabalhos anteriores utilizam técnicas de Aprendizagem de Máquina para auxiliar o compilador no seu processo de tomada de decisão.

3. Computação Paralela

A Computação Paralela é uma área de estudo da Ciência da Computação que estuda como criar algoritmos que obtenham o máximo de desempenho dos computadores modernos

[Golub and Ortega 2014], muitas vezes isso inclui explorar arquiteturas não convencionais, como *GPUs* e *FPGAs*. Atualmente, os cientistas tem se preocupado com outros fatores além do tempo de execução de um programa, um exemplo é o consumo de energia do computador [Saffran et al. 2016].

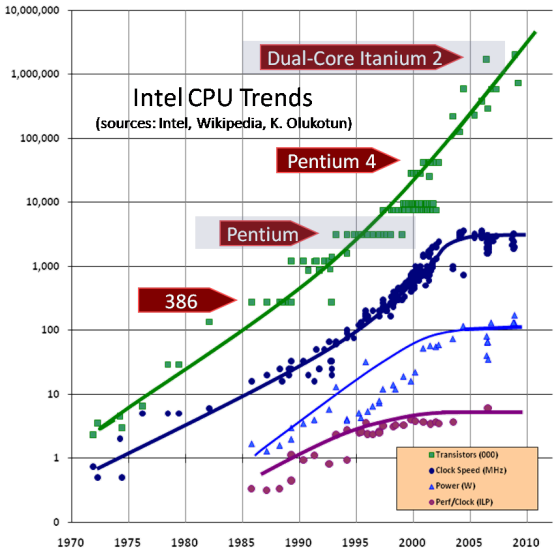

Figura 1. Evolução dos processadores nos últimos anos.

A computação paralela surgiu como solução para os problemas de escalabilidade de processadores monoprocessados. O aumento da frequência, que foi a principal técnica para aumentar o desempenho desses processadores, começou a sobreaquecer os computadores, gerando um elevado consumo energético, como pode ser visto na Figura 1. Para resolver este problema, ao invés de aumentarem o número de transistores, aumentou-se o número de núcleos. Esta estratégia resolveu o problema, mas fez surgir vários outros, uma vez que o desempenho das aplicações *single threaded* agora passam a subutilizar a arquitetura disponível e surgiu a necessidade dos programadores explorarem o paralelismo explicitamente em seus códigos.

Existem várias maneiras de se explorar o paralelismo nas arquiteturas modernas [Costa et al. 2017]. Contudo, as maneiras mais comum são:

- Paralelismo de Instrução: Realizado pelo processador, visa explorar o máximo do *pipeline* de execução. Geralmente, utiliza operações que atuam sobre vários dados ao mesmo tempo (*SIMD*). É um tipo de paralelismo que é realizado pelo compilador de forma transparente para o programador.
- Paralelismo de dados: Paralelismo que aplica um único algoritmo em um conjunto de dados. Ele é realizado utilizando-se várias *threads* programadas explicitamente pelo desenvolvedor do aplicação.
- Paralelismo de Tarefas: Quando são realizadas mais de uma tarefa sobre um determinado conjunto de dados ao mesmo tempo, utilizando *threads* ou outras máquinas.

Apesar de existirem várias formas de se explorar o paralelismo nas arquiteturas modernas, a maioria delas ainda depende do programador [Nickolls et al. 2008]. Por

causa disso os pesquisadores de computação paralela projetaram várias estratégias para facilitar o processo de paralelização de código. Um exemplo é o uso de Padrões de Programação Paralela [McCool et al. 2012], com estes padrões torna-se fácil identificar a melhor estratégia para se paralelizar uma aplicação baseando-se somente no seu funcionamento. Uma vez identificado o padrão é necessário implementá-lo. Para isso existem diversas bibliotecas/*frameworks* que ajudam o programador, alguns exemplos são: OpenMP, OpenACC, CUDA, MPI e pthreads.

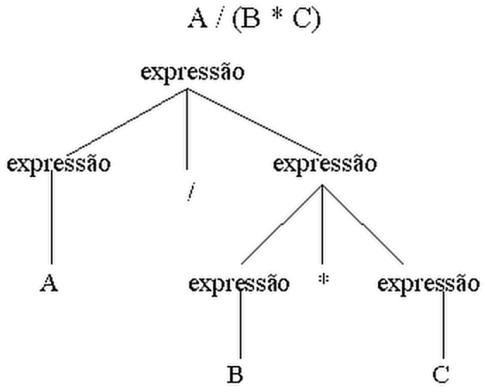

Figura 2. Exemplo de Árvore Sintática.

Uma biblioteca muito utilizada para paralelização de código é o OpenMP [Dagum and Menon 1998]. Esta biblioteca define uma API para auxiliar na execução de operações comuns na paralelização de código, como por exemplo: redução, tratamento de condições de corrida e sincronização de *threads*.

Uma outra maneira de se explorar o paralelismo é através do próprio compilador, que pode tentar gerar um código paralelo de forma transparente para o programador. A técnica mais utilizada para paralelização de código automática é através da análise estática [Lee et al. 2017]. Esta consiste em percorrer a Árvore Sintática (Figura 2) gerada pelo compilador após a análise sintática, procurando por padrões de código que possam indicar que o código é paralelizável.

4. Aprendizagem por Reforço

A Aprendizagem por reforço (RL) é um campo de Aprendizagem de Máquina inspirado na psicologia comportamental. Este campo estuda a criação de agentes inteligentes que tomam decisões em um ambiente com o objetivo de maximizar o acúmulo de recompensas recebidas a longo prazo [Mnih et al. 2013].

A maneira mais comum de modelar um problema de RL é utilizando um *Processo de decisão de Markov* (MDP) [Davis 2018], um *framework* matemático utilizado para modelar situações e processos de decisões nos quais as ações são parte estocásticos e parte determinísticos. A Figura 3 mostra uma representação gráfica de um MDP. Ele é utilizado na modelagem de problemas de otimização que podem ser resolvidos utilizando RL ou programação dinâmica [Sarkale et al. 2018]. Um MDP pode ser definido matematicamente como uma quíntupla $(S, A, P.(.,.), R.(.,.), \gamma))$:

- S: Conjunto de estados, pode ser finito ou infinito.

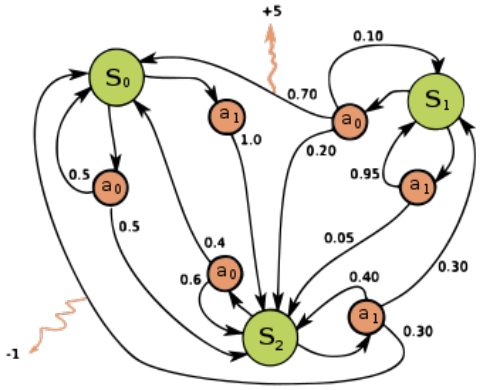

Figura 3. Exemplo de MDP.

- A: Conjunto de ações que podem ser tomados pelo agente.
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$: Função que determina a probabilidade de em um determinado momento no tempo t estando no estado s e tomando a ação a o agente transitar para o estado s'.
- $R_a(s, s')$: Função que determina a recompensa imediata recebida pelo agente por ter realizado a ação a em um estado s e ter transitado para o estado s'.
- $\gamma \in [0, 1]$: constante conhecida como Fator de Desconto. Determina a diferença de importância entre recompensas recentes e futuras.

Algoritmo 1: Q-learning: Aprende a Função $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Entrada: $S, A, P(s, a), R(s, a), \gamma, \alpha$
Saída: Q

início
  Inicializa $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
  enquanto Q *não convergir* **faça**
    Começando do estado inicial $s \in \mathcal{X}$
    enquanto s *não for terminal* **faça**
      Calcula-se π de acordo com Q e com a estratégia de exploração (exemplo: $\pi(x) \leftarrow \arg\max_a Q(x, a)$)
      $a \leftarrow \pi(s)$
      $r \leftarrow R(s, a)$
      $s' \leftarrow P(s, a)$
      $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
      $s \leftarrow s'$
    fim
  fim
  retorna Q
fim

Uma vez definido o MDP, é necessário escolher um algoritmo capaz de determinar uma política, isto é, um mapeamento de qual ação tomar em cada estado para maximar o total de recompensas recebidas. Um exemplo algoritmo utilizado para isso é o *QLearning*

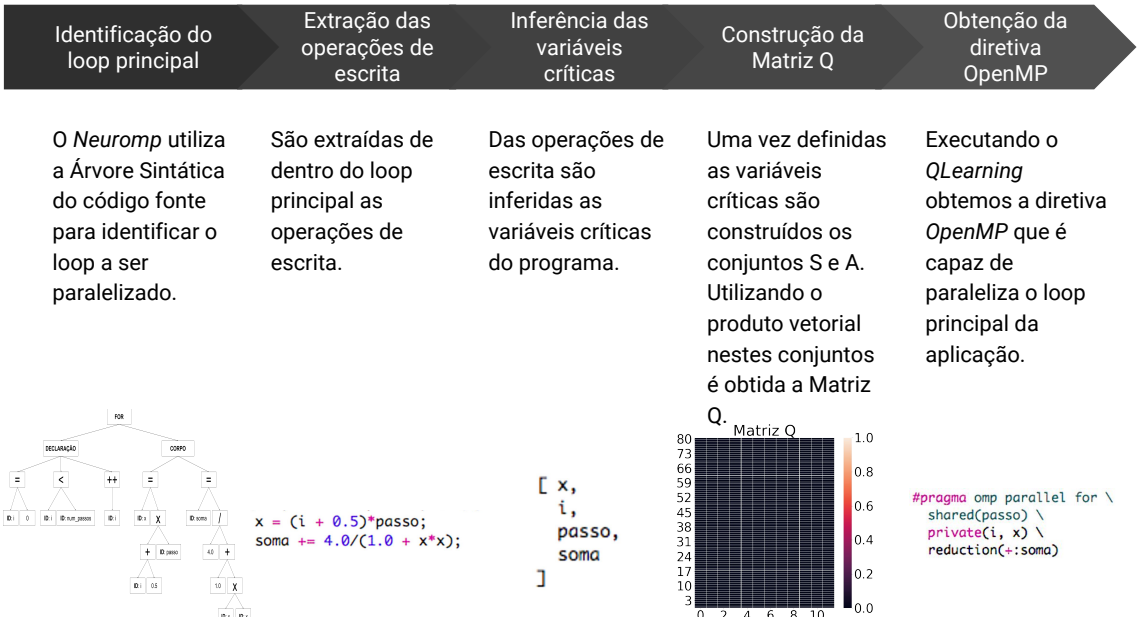

Figura 4. Funcionamento do *NeurOMP*.

[Mnih et al. 2013]. Este algoritmo aprende uma distribuição de probabilidade para cada estado do MDP, chamada de função Q, esta distribuição indica quais ações são mais prováveis de retornar a maior recompensa, este processo pode ser visto no Algoritmo 1.

5. NeurOMP

Nesta seção, é discutido o *NeurOMP*, uma otimização de compilador, aplicável em instruções do tipo *for loop*, que utiliza o algoritmo *QLearning* para definir o pragma *OpenMP* capaz de paralelizar um algoritmo escrito em C. Vale ressaltar que o *NeurOMP* realiza a paralelização em nível de dados e não trata condições que necessitam de exclusão mútua, mas ele pode ser facilmente integrado com o *OpenMP* tornando esse problema mais fácil de ser resolvido pelo programador. O funcionamento do *NeurOMP* está sumarizado na Figura 4.

O primeiro passo para se utilizar um algoritmo de aprendizagem por reforço é definir um MDP. Este arcabouço matemático nos permite definir os comportamentos tanto do mundo quanto do agente, desta forma cria-se um ambiente onde o agente pode aprender a maximizar o recebimento de recompensas a longo prazo. Para o *NeurOMP*, o MDP está definido da seguinte forma:

- S: Conjunto contendo as diretivas *OpenMP* válidas para um determinado *loop* da aplicação.
- A: Todas as transformações de variáveis permitidas pelo *OpenMP*.
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$: Essa função é constante, sendo a probabilidade igual a 1, uma vez que esse ambiente é totalmente determinista.
- $R_a(s, s')$: Esta função está definida na figura 4. Para o cálculo da recompensa são definidas 4 variáveis derivadas de um estado: $T_s s$ tempo de execução do código sequencial, $T_s p$ tempo de execução do código paralelo gerado em um estado s, $R_s s$ resultado da execução do código sequencial e $R_s p$ resultado da execução do código paralelo gerado em um estado s.

- $\gamma \in [0, 1]$: Este valor foi definido de maneira empírica em 0.95.

$$R_a(T_ss, T_sp, R_ss, R_sp) = \begin{cases} -1, & T_sp > T_ss \\ -1, & R_sp \neq R_ss \\ speedup(T_ss, T_sp), & (T_ss, T_sp, R_ss, R_sp) \in \mathbb{R}, \end{cases}$$

Figura 5. Equação que define a recompensa recebida em um determinado estado apos tomar uma ação.

O próximo passo para a execução do *NeurOMP* é a definição da matriz Q, utilizada pelo *QLearning*. Essa matriz é definida por $Q : (S \times A)$. Para obtermos os conjuntos S e A, é necessário percorrer a Árvore Sintática e identificar a operação de *loop* principal da aplicação. Esta é indicada pelo programador usando uma diretiva de compilação similar às do *OpenMP*.

Uma vez determinado o *loop*, o *NeurOMP* identifica as variáveis críticas, variáveis que podem estar envolvidas em alguma condição de corrida. Isto é feito percorrendo mais uma vez a AST e, dentro do *loop*, encontrando as variáveis envolvidas em operações de escrita. Com estas variáveis é possível definir os conjuntos S e A necessários na matriz Q.

Uma vez construída a matriz, executa-se o *QLearning*. O algoritmo utiliza as informações contidas no MDP para encontrar qual a diretiva *OpenMP* capaz de tornar o código paralelo. Porém, este método pode ser demorado para o processo de compilação. Por isso, é utilizado no *NeurOMP*, uma heurística para parar a execução caso o resultado já encontrado seja satisfatório. O programador pode definir um limiar para o *speedup*, caso o *NeurOMP* encontre uma paralelização que tenha um ganho de desempenho maior que o limiar, para-se a execução do *QLearning*.

Após a finalização da execução do *NeurOMP*, tem-se a função Q definida. Desta maneira, partindo do estado inicial da matriz Q e executando as melhores ações para cada estado, obtém-se a diretiva que gerou maior ganho de desempenho, pela otimização do *QLearning*.

6. Resultados Experimentais

Para avaliar a eficiência do *NeurOMP*, foram realizados testes utilizando o *CAP Bench* [Souza et al. 2017]. Uma vez que este *benchmark* já é paralelizado com *OpenMP*, é comparado o desempenho da otimização com o humano. Outra comparação de desempenho realizada é com as técnicas já existentes no compilador para paralelização automática de código, para isso foram utilizados 2 compiladores, o ICC (*Intel Compiler Collection*) e o GCC (GNU Compiler Collection), ambos utilizam análise estática para tentar realizar a paralelização. Em especial, o ICC utiliza diretivas de compilação para identificar quais *loops* paralelizar, estratégia similar ao *NeurOMP*.

Os testes foram realizados em uma arquitetura contendo dois processadores *Intel core i5*, 8GB de memória principal, rodando o sistema operacional Linux Ubuntu 16.04. Para cada aplicação presente no *CAP Bench*, foram geradas 3 variações do código: uma versão sequencial, a versão paralelizada pelo humano e a versão paralelizada pelo

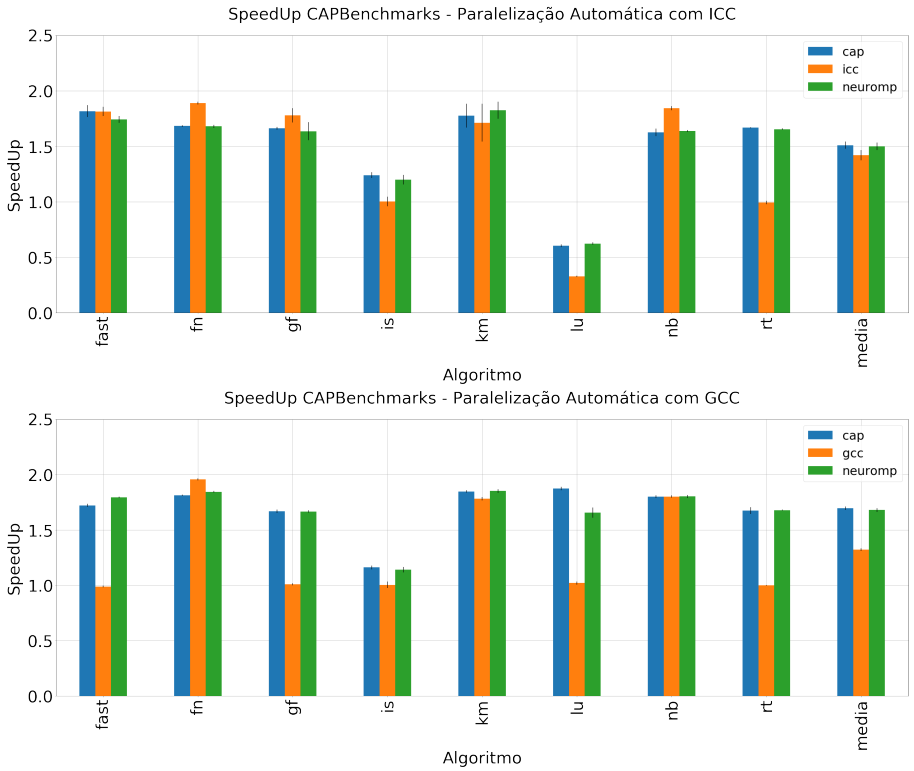

Figura 6. Resultados os testes com o ICC e com GCC.

NeurOMP. Foram realizados 2 testes, um para cada compilador. Os testes consistem em: compilar cada variação, paralelizar o código pelo compilador e por fim calcular o *speedup* em relação ao código sequencial para as paralelizações do compilador, do humano e do *NeurOMP*. Cada aplicação foi executada 10 vezes em cada variação, a média aritmética e o desvio padrão foram calculados, garantindo-se um grau de confiança de 95%.

Como pode ser visto na figura 6, o *NeurOMP* foi capaz de paralelizar todas as aplicações do *CAP Bench* de maneira similar ao especialista humano. Apesar de alguns casos os compiladores conseguirem um desempenho superior, por eles paralelizarem todos os *loops* enquanto o humano e o *NeurOMP* agem somente naqueles computacionalmente mais custosos. Nota-se na figura 6 que para a paralelização da aplicação *LU* no ICC, obteve-se um *speedup* inferior a 1, isso se deve ao fato que o compilador utilizou alguma outra otimização além do paralelismo que tornou o código sequencial significativamente mais rápido, tornando o paralelismo ineficiente. Como o ICC não possui código aberto, não é possível afirmar com certeza qual é esta otimização.

O *NeurOMP* obteve um ganho médio superior a ambos os compiladores, como poder ser visto na Figura 6. O *speedup* médio da otimização foi superior no GCC, em comparação com o ICC. Isto se deve ao fato que o ICC realiza mais otimizações do que o GCC, o que torna o código mais eficiente. Esta diferença do código gerado pode ser observada pela diferença de *speedup* nos dois compiladores. Pelos gráficos, o *NeurOMP* obteve um ganho de desempenho 27.19% maior em comparação com o GCC e 5.57% maior que o ICC.

7. Conclusão

Neste trabalho, foi apresentado o *NeurOMP*, uma otimização de compilador que utiliza o *QLearning*, um algoritmo de aprendizagem por reforço, para realizar a paralelização automática de código utilizando *OpenMP*. Os resultados experimentais demonstram que a otimização obtém um *speedup* médio no *CAP Bench* igual a 1.6, similar a um especialista humano.

Para trabalhos futuros, pode-se aplicar a mesma técnica para paralelizar outras arquiteturas, como GPU ou FPGAs ou aglomerados de computadores. Além disso, pode estender o *NeurOMP* para que ele possa ser capaz de identificar regiões críticas e realizar paralelismo de tarefas.

Referências

Ahmed, J., Siyal, M. Y., Najam, S., and Najam, Z. (2017). Challenges and issues in modern computer architectures. In *Fuzzy Logic Based Power-Efficient Real-Time Multi-Core System*, pages 23–29. Springer.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. (2010). A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75.

Chapman, B., Jost, G., and Van Der Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press.

Coelli, T. J. (1996). A guide to frontier version 4.1: a computer program for stochastic frontier production and cost function estimation. Technical report, CEPA Working papers.

Coons, K. E., Robatmili, B., Taylor, M. E., Maher, B. A., Burger, D., and McKinley, K. S. (2008). Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 32–42. ACM.

Costa, G. A., Bentes, C., Ferreira, R. S., Feitosa, R. Q., and Oliveira, D. A. (2017). Exploiting different types of parallelism in distributed analysis of remote sensing data. *IEEE Geoscience and Remote Sensing Letters*, 14(8):1298–1302.

Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

Davis, M. H. (2018). *Markov models & optimization*. Routledge.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Bonilla, E., Thomson, J., Leather, H., et al. (2008). Milepost gcc: machine learning based research compiler. In *GCC Summit*.

Golub, G. H. and Ortega, J. M. (2014). *Scientific computing: an introduction with parallel computing*. Elsevier.

Gubner, T. and Boncz, P. (2017). Exploring query execution strategies for jit, vectorization and simd.

Khatami, Z. (2017). Compiler and runtime optimization techniques for implementation scalable parallel applications.

Kulkarni, S. and Cavazos, J. (2012). Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices*, 47(10):147–162.

Lee, Y., Jeong, J., and Son, Y. (2017). Design and implementation of the secure compiler and virtual machine for developing secure iot services. *Future Generation Computer Systems*, 76:350–357.

Marinkovic, V., Popovic, M., and Djukic, M. (2018). An automatic instruction-level parallelization of machine code. *Advances in Electrical and Computer Engineering*, 18(1):27–36.

McCool, M. D., Robison, A. D., and Reinders, J. (2012). *Structured parallel programming: patterns for efficient computation*. Elsevier.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Moreira, K. C. A., Mendonça, G. S. D., Guimarães, B., Alves, P., and Pereira, F. M. Q. (2016). Paralelização automática de código com diretivas openacc. *SBLP. SBC*.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 classes*, page 16. ACM.

Saffran, J., Garcia, G., Souza, M. A., Penna, P. H., Castro, M., Góes, L. F., and Freitas, H. C. (2016). A low-cost energy-efficient raspberry pi cluster for data mining algorithms. In *European Conference on Parallel Processing*, pages 788–799. Springer.

Sarkale, Y., Nozhati, S., Chong, E. K., Ellingwood, B., and Mahmoud, H. (2018). Solving markov decision processes for network-level post-hazard recovery via simulation optimization and rollout. *arXiv preprint arXiv:1803.04144*.

Schardl, T. B., Moses, W. S., and Leiserson, C. E. (2017). Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–265. ACM.

Souza, M. A., Penna, P. H., Queiroz, M. M., Pereira, A. D., Góes, L. F. W., Freitas, H. C., Castro, M., Navaux, P. O., and Méhaut, J.-F. (2017). Cap bench: a benchmark suite for performance and energy evaluation of low-power many-core processors. *Concurrency and Computation: Practice and Experience*, 29(4).

Stephenson, M., Amarasinghe, S., Martin, M., and O'Reilly, U.-M. (2003). Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, volume 38, pages 77–90. ACM.

Tong, Y., Zhang, W., Ma, Y.-C., Liu, Y., Liang, Y., Zhang, T., and Luo, H. (2017). Compiler-guided parallelism adaption based on application partition for power-gated ilp processor. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(4):1329–1341.

Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. I. (2003). Compiler optimization-space exploration. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 204–215. IEEE.