WSCAD-WIC 2018

Workshop de Iniciação Científica em Arquitetura de Computadores e Computação de Alto Desempenho

Conteúdo

WSCAD-WIC Artigos

WIC S1

WIC S2

WIC S3

Controle de Acesso Baseado em Papéis em Ambientes Assistidos

Gabriel R. Goulart[1], **Mário A. R. Dantas**[2]

[1]Departamento de Informática e Estatística (INE) - Universidade Federal de Santa Catarina (UFSC)

[2]Instituto de Ciências Exatas (ICE) - Universidade Federal de Juiz de Fora (UFJF)

gabriel.r.goulart94@gmail.com, mario.dantas@ice.ufjf.br

Abstract. *This work presents an access control system based on user and environment roles, using context information to compose the access rules. To validate the functioning of this system, 4 individuals from different age groups performed tests during a two-week period in a home-based environment, where RFID tags were used to identify users in a non-intrusive way. After the tests, it was identified that all access attempts were processed correctly, regardless of the changes of roles or access rules, which shows that the proposed system does indeed work in a ambient assisted living.*

Resumo. *Este trabalho apresenta um sistema de controle de acesso baseado em papéis de usuário e ambiente, utilizando informações de contexto para compor as regras de acessos. Para validar o funcionamento desse sistema, 4 indivíduos de diferentes faixas etárias realizaram testes durante o período de duas semanas em um ambiente assistido domiciliar, onde tags RFID foram utilizadas para identificar os usuários de maneira não intrusiva. Após os testes, identificou-se que todas as tentativas de acesso foram processadas corretamente, independentemente das mudanças dos papéis ou das regras de acesso, o que mostra que o sistema proposto de fato funciona em um ambiente assistido.*

1. Introdução

Segundo uma pesquisa das Nações Unidas, a população idosa dobrará até 2050 [ONU 2017]. Com isso o consumo de serviços voltados para essa população também aumentará, porém se soluções inovadoras não forem encontradas e aplicadas, esses serviços sofrerão com um défice muito grande para suprir as necessidades da sociedade.

Nesse contexto soluções como a de ambientes assistidos são aplicadas com o objetivo de fornecer uma ajuda, e complementar os serviços voltados a saúde e bem estar, não só dos idosos, mas de qualquer indivíduo que precise ser assistido. A utilização dessa solução provê um maior conforto para o indivíduo, pois ele poderá viver no seu ambiente domiciliar e mesmo assim continuar sendo acompanhado pelo o seu médico por exemplo. E se alguma anormalidade acontecer, imediatamente todos os envolvidos no cuidado do indivíduo serão notificados e as ações necessárias serão tomadas.

Com o crescimento e popularização dos ambientes assistidos, não se pode deixar de pensar na segurança desses ambientes, e por este motivo o controle de acesso é de

extrema importância, pois garante acesso ao ambiente, acesso físico, apenas para pessoas aptas a acessá-lo, utilizando diversas abordagens como por exemplo a baseada em papéis.

Seguindo essa linha, este trabalho apresentará um sistema para realizar o controle de acesso baseado em papéis, papéis de usuário e ambiente, juntamente com informações de contexto, ou seja, informações que o ambiente pode prover para o sistema, com o objetivo de realizar um controle de acesso inteligente e sensível ao ambiente.

Este trabalho está dividido da seguinte forma: na seção 2 é realizado uma análise dos trabalhos correlatos, na seção 3 é apresentado a proposta deste trabalho, já na seção 4 os resultados são discutidos, e por fim a conclusão e trabalhos futuros, apresentados na seção 5.

2. Trabalhos Correlatos

Diversas alternativas para realizar o controle de acesso baseado em papéis são encontradas. [Zhang et al. 2004] utilizam máquinas de estados para fazer o controle de papéis ativos e permissões atribuídas aos papéis. Como a aplicação é consciente de contexto, um agente de contexto coleta as informações e gera eventos que disparam transições nas máquinas de estados.

Outra abordagem consciente de contexto é a de [Covington et al. 2001], que traz o conceito de papel de ambiente, o que não é um elemento do padrão de controle de acesso baseado em papéis. Sendo assim, com essa nova atribuição, as permissões são associadas tanto aos papéis de usuários quanto aos papéis de ambiente, provendo uma maior flexibilidade para o sistema como um todo.

[Kayes et al. 2017] utilizam as informações de contexto para ativar o papel do usuário, semelhante aos trabalhos apresentados anteriormente. A utilização do contexto para ativar um papel de usuário é realizada através de expressões contextuais, ou seja, uma composição de contextos, onde se pode utilizar informações como a localização do usuário, dias da semana ou até mesmo as escalas de trabalho, por exemplo.

3. Proposta

Através da análise dos trabalhos correlatos constatou-se que os modelos de controle de acesso propostos não deixam claro a sua aplicabilidade em ambientes assistidos. Portanto com o objetivo de explorar essa questão, este trabalho propõe um sistema para controlar o acesso físico em ambientes assistidos de maneira não intrusiva, utilizando papéis de usuários e ambientes juntamente com as informações de contexto para construir regras de acesso.

A Figura 1 apresenta um exemplo do funcionamento do sistema. Supondo que um usuário X desempenha o papel de usuário EMPREGADO e está tentando acessar um ambiente Y com papel de ambiente SALA DO CHEFE, e a regra de acesso associada ao papel de usuário e ambiente utilize as seguintes informações de contexto : horário, data e se o chefe se encontra em sua sala. Para que o usuário X consiga acessar a sala do chefe as informações de contexto precisam ser verdadeiras.

A identificação do usuário é realizada de maneira não intrusiva utilizando tags RFID (*Radio Frequency Identification*). As tags são associadas a papéis de usuário, que

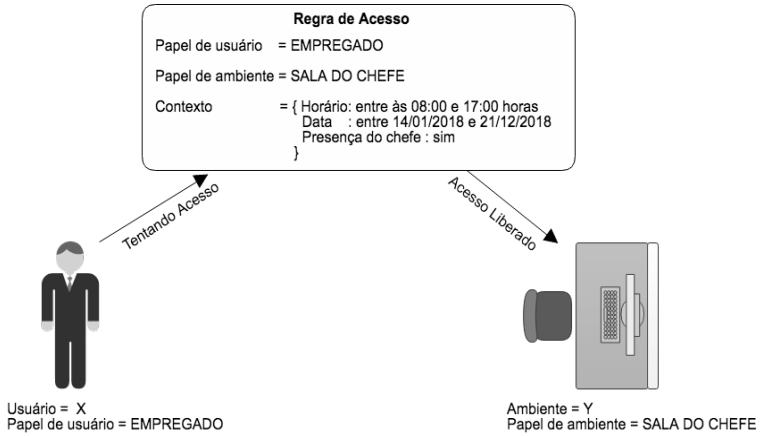


Figura 1. Exemplo do Sistema

por sua vez integram a regra de acesso. As informações e associações, como por exemplo entre usuário e papel de usuário, são armazenadas em um banco de dados MySQL. Importante ressaltar que o sistema foi implementado utilizando as linguagens Python e C para Arduino junto com o protocolo de comunicação MQTT (*Message Queuing Telemetry Transport*).

3.1. Controle de Acesso Baseado em Papéis

Neste trabalho o controle de acesso utilizado será o baseado em papéis, o RBAC (*Role Based Access Control*), porém só será utilizado as funções básicas, as quais garantem o funcionamento do controle de acesso.

Para que o controle de acesso aproveite todos os recursos que um ambiente assistido pode oferecer, se baseando na expansão do RBAC feita por [Covington et al. 2001], será adicionado ao modelo o conceito de ambiente e papel do ambiente, o que permitirá uma maior flexibilidade no sistema, pois a regra de acesso não dependerá apenas do papel do usuário, mas também do papel do ambiente. A Figura 2 apresenta a expansão do RBAC que será utilizado neste trabalho.

Especificações para a expansão do RBAC:

- Usuário (US), Papel de Usuário (PA), Ambiente (AM) , Papel de Ambiente (PM), Sessão (SE), Permissão (PERMS), Contexto (CON).
- $PAUAM = 2^{(\text{PA X PM})}$, conjunto de papéis de usuário associados a papéis de ambiente.
- $AU \subseteq$ US X PAUAM, relação de atribuição muitos para muitos entre usuário e papéis de usuário relacionados com papéis de ambiente.
- $AA =$ relação de um para muitos entre ambientes e papéis de ambiente.
- $AP \subseteq$ PERMS X PAUAM, relação de atribuição um para muitos entre papéis de usuário associados a papéis de ambiente e permissões.
- $Sessão\_usuário(s : S) \rightarrow US$, mapeamento da sessão s em um usuário correspondente.
- $Sessão\_ambiente(s : SE) \rightarrow AM$, mapeamento da sessão s em um ambiente correspondente.

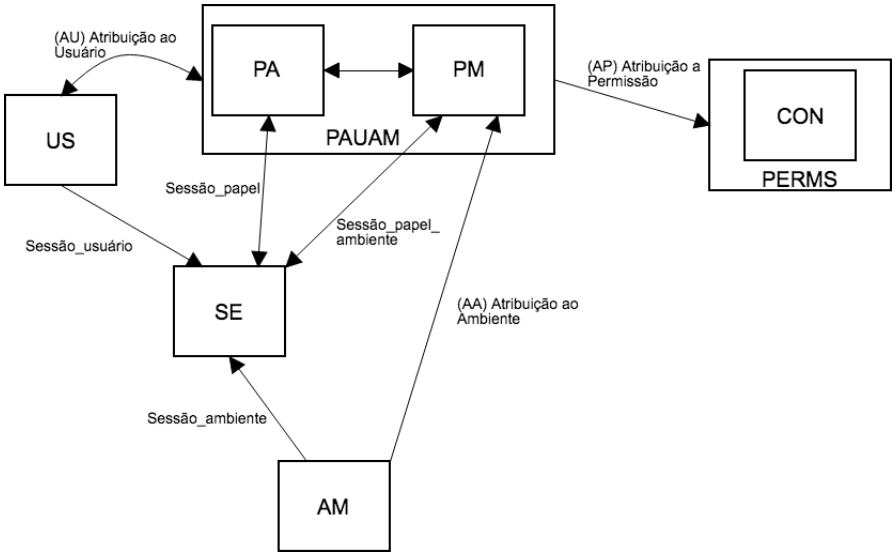


Figura 2. Extensão do RBAC

- $Sessão\_papel(s : S) \rightarrow 2^{PA}$, mapeamento da sessão s em um conjunto de papéis.
- $Sessão\_papel\_ambiente(s : SE) \rightarrow 2^{PM}$, mapeamento da sessão s em um conjunto de papéis de ambiente.

Outra adaptação importante a ser feita no RBAC, é que a sessão representará que o usuário está registrado em um ambiente, ou seja, representará que o usuário está no ambiente. Caso o usuário não tenha uma sessão ativa para um certo ambiente, significa que o usuário não está no ambiente.

4. Ambiente e Resultados Experimentais

Nesta seção serão apresentados os resultados obtidos através dos experimentos realizados. O sistema de controle de acesso foi aplicado em dois ambientes, sendo testado por quatro usuários durante o período de duas semanas.

4.1. Resultados

Ao longo de duas semanas de teste, resultados expressivos foram obtidos a fim de mostrar a validade do sistema. Nas 284 tentativas de acesso, o sistema se comportou de acordo com o esperado, lendo a tag do usuário, capturando o contexto do ambiente, processando as regras de acesso e verificando se o usuário poderia ou não acessar o ambiente.

A Figura 3 expõe o gráfico de utilização do sistema através da quantidade de eventos capturados. Esses eventos consistem em tentativas de acesso nos ambientes por parte dos usuários. Durante o período de teste foram registrados 284 eventos, sendo 240 acessos que foram garantidos pelas regras de acesso, e 44 que foram negados.

Já a Figura 4 apresenta os eventos registrados no sistema agrupados por horário. Essa informação é interessante, pois se consegue ter uma noção da utilização do sistema por horário, e assim se obter conclusões. Por exemplo, pode-se observar que por volta das 19 horas o sistema passou por um pico de eventos, o que significa que houveram muitas tentativas de acesso.

Procurando deixar mais explícito o funcionamento e a dinamicidade do sistema de controle de acesso, mudanças no papel de ambiente do ambiente 1 foram realizadas. Em um primeiro momento o ambiente desempenhou o papel de ambiente 1 (Quarto Filho), e após o papel de ambiente 3 (Sala de Estar).

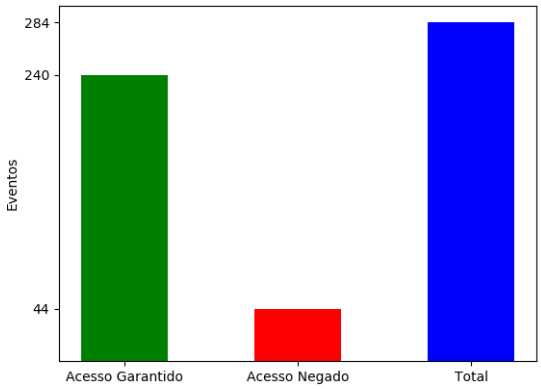


Figura 3. Eventos no sistema

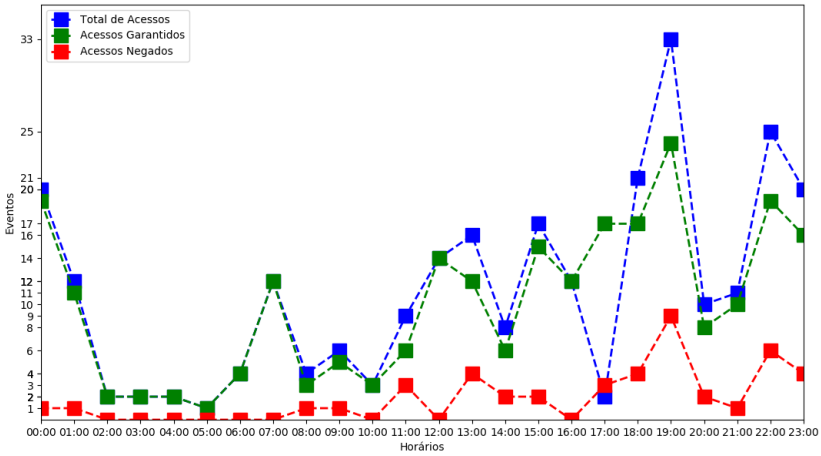


Figura 4. Eventos no sistema por horário

Na Figura 5 se pode observar os acessos no ambiente 1 agrupados por papéis de ambiente e horário, o que comprova que a mudança de papel de ambiente também alterou a forma que o ambiente é utilizado, por exemplo, entre as 2 e 5 da manhã, desempenhando o papel de ambiente 1, houveram por volta de 6 acessos, enquanto que desempenhando o papel de ambiente 3 foram 0 acessos. Esse resultado mostra que o controle de acesso funcionou de maneira correta mesmo com alterações nas configurações.

5. Conclusão e Trabalhos Futuros

Com este trabalho foi possível analisar diversos aspectos relacionados à área de controle de acesso. Sendo assim identificou-se que a abordagem baseada em papéis utilizando

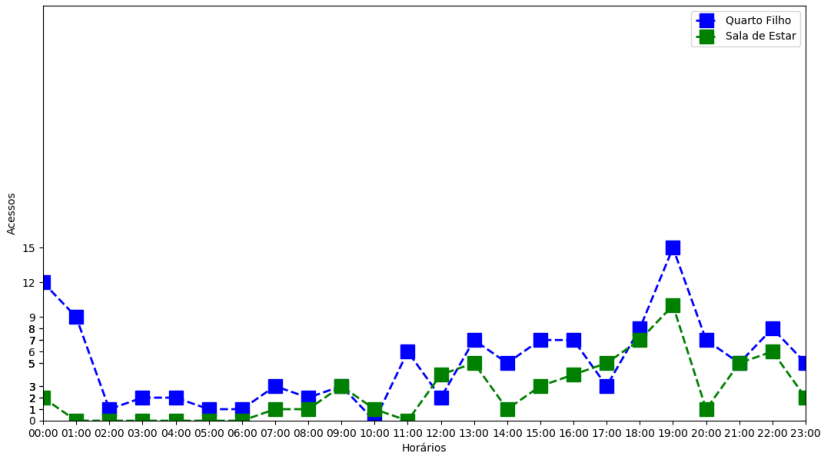


Figura 5. Acessos no ambiente 1 por horário e por papel de ambiente

informações de contexto, é pouco empregada e explorada em ambientes assistidos. Por este motivo decidiu-se utilizar essa abordagem neste trabalho, implementando um sistema capaz de controlar o acesso, se baseando em papéis de usuário e ambiente em conjunto com informações de contexto.

De acordo com os resultados apresentados na seção 4, este trabalho atingiu o seu objetivo, pois o sistema realizou o controle de acesso, utilizando os papéis e informações que o ambiente pôde prover, funcionando de maneira dinâmica e adaptativa, se adequando as mudanças de configuração e estado dos ambientes controlados.

Para trabalhos futuros é sugerido adicionar ao sistema os demais módulos do padrão de controle de acesso baseado em papéis, como por exemplo a hierarquia de papéis. Também é sugerido a implementação de um sistema de gerenciamento, onde se poderá adicionar, editar e excluir papéis, regras de acesso, usuários e ambientes.

Referências

Covington, M. J., Long, W., Srinivasan, S., Dev, A. K., Ahamad, M., and Abowd, G. D. (2001). Securing context-aware applications using environment roles. *Proceedings of the sixth ACM symposium on Access control models and technologies*, (January):pp. 10–20.

Ferraiolo, D. F. and Kuhn, R. D. (2004). Role based access control.

Kayes, A. S. M., Han, J., Rahayu, W., Islam, M. S., and Colman, A. (2017). A Policy Model and Framework for Context-Aware Access Control to Information Resources.

ONU (2017). World population ageing.

Sandhu, R. S. and Samarati, P. (1994). Access control - principles and pratice.

Zhang, G., Zhang, G., Parashar, M., and Parashar, M. (2004). Context-aware dynamic access control for pervasive applications. *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 21–30.

Sistema de Recomendação para a Saúde baseado em Computação Ubíqua

Gabriel Di iorio Silva, Victor Ströele, Mario Dantas

[1]Departamento de Ciência da Computação – Universidade Federal de Juiz de Fora (UFJF)

{iorio,victor.stroele,mario.dantas}@ice.ufjf.br

Resumo. *Com a evolução tecnológica, as pessoas buscam, cada vez mais, por soluções que auxiliem nas suas atividades diárias e, além disso, possam agregar qualidade de vida. Neste sentido, este trabalho apresenta o modelo HARA-RS, um sistema de recomendação baseado no monitoramento de atividades humanas. Através do uso de sensores e monitoramento de movimentos, é definido o perfil e contexto de usuários para a recomendação de conteúdos relevantes para a sua saúde. Discernir as atividades que um indivíduo realiza é importante, pois assim é possível identificar anomalias em sua rotina ou sugerir atividades para diminuir o sedentarismo. Estudos iniciais mostram que a proposta é viável e pode ser considerada como um diferencial de pesquisa.*

1. Introdução

Segundo pesquisas realizadas em 2016 pelo IBGE, o número de idosos no Brasil triplicará em 40 anos e passará de 10% da população brasileira, em 2010, para 29,3% em 2050 [IBGE 2016]. Com o advento da internet e tecnologias móveis cada vez mais presentes em nosso cotidiano, termos como computação ubíqua foram utilizados para designar a onipresença da informática em nossas rotinas. Somando isso ao cenário atual de envelhecimento da população brasileira, tecnologias que permitam uma melhor qualidade de vida e que possam auxiliar na saúde tornam-se promissoras, permitindo monitorar pessoas em seus ambientes domiciliares, os chamados *ambientes domiciliares assistidos*.

Em geral, essas abordagens utilizam sensores para monitorar os usuários. Neste trabalho, para interpretar os dados recolhidos dos acelerômetros, foi utilizado o modelo HARA (*Human Activity Recognition with Accelerometer*) [Amaral and Dantas 2017], que consiste no armazenamento dos dados de posicionamento na residência, movimentação e tempo, sendo possível inferir as atividades diárias de um indivíduo, e assim entender sua rotina. Portanto, baseando-se nos dados coletados pelos acelerômetros e interpretados pelo modelo HARA, seremos capazes de recomendar ações, como a sugestão de atividades para redução de sedentarismo, e, até mesmo, sugerir que o usuário procure um especialista da área de saúde, aumentando as chances de um tratamento bem-sucedido, pela detecção precoce de uma doença.

Os Sistemas de Recomendação (SR) buscam apresentar informações relevantes e personalizadas para os usuários considerando suas necessidades. No contexto da saúde, esses sistemas podem ser utilizados para recomendar conteúdos relevantes para a saúde dos usuários. Os sistemas de recomendação também podem ser utilizados no ramo acadêmico para recomendar artigos, trabalhos e vídeos de acordo com a necessidade, ramo de pesquisa e objetivo do usuário. Além disso, podemos observar sistemas

de recomendação voltados para o lazer, seja na recomendação de músicas ou filmes de acordo com os gostos interpretados e analisados, tornando assim sua experiência mais agradável e prática na plataforma.

Portanto, dado esse contexto e a ampla utilização dos sistemas de recomendação, o objetivo desse trabalho é desenvolver um SR que atue conforme as atividades identificadas pelos acelerômetros e interpretadas pelo modelo HARA, consiga enviar relatórios para especialistas médicos, além de recomendar devidas ações para usuários que apresentarem atividades anômalas para prevenir possíveis agravações de doenças, como também melhorar a qualidade de vida dos usuários.

Este trabalho está organizado como segue: na Seção 2 são apresentados os trabalhos correlatos, na Seção 3 é apresentada a arquitetura do modelo HARA-RS; na Seção 4 uma avaliação preliminar é conduzida e, na Seção 5, são feitas as considerações finais.

2. Trabalhos Correlatos

Ultimamente, muitos trabalhos passaram a abordar o tema de SR voltados para a saúde. Por se tratar de um tema relativamente novo e levando em consideração o cenário futuro, com a expectativa de vida se elevando cada vez. Dessa forma, alguns trabalhos como [Wiesner and Pfeifer 2014], apresentam sistemas que, com base no perfil do usuário, recomendam artigos e textos que ajudam a entender seu estado clínico ou até mesmo recomendar textos que abordam temas na saúde, porém com uma linguagem de fácil entendimento de acordo com o usuário.

Outros trabalhos utilizam sensores para analisar a saúde do usuário e coletar dados que são enviados para um especialista ou para a análise por parte do próprio indivíduo [Amay J Bandodkar 2014]. Entretanto, são poucos trabalhos que focam em soluções que utilizam sensores com o intuito de coletar e transmitir dados para o usuário final. [Lee and Chung 2009] propuseram o uso de camisas inteligentes, compostas por alguns sensores, que coletam os dados do indivíduo por meio de acelerômetros. Tal sistema é baseado em uma arquitetura na qual os dados coletados são enviados para um repositório único com todas as informações obtidas pelos sensores. Outros sistemas utilizam variados tipos de sensores, capazes de enviar os dados coletados para uma pessoa especializada ou para que o próprio usuário realize a análise dos dados coletados, tendo arquiteturas variadas de acordo com a necessidade do usuário [Jung et al. 2008].

O uso de sensores é amplamente difundido em pesquisas científicas, pois, dessa forma, o sistema se torna ainda mais personalizado para o usuário. Neste sentido, este trabalho utiliza os sensores de forma a criar um ambiente assistido e realizar o reconhecimento das atividades dos usuários, além de enviar a devida recomendação de acordo com a interpretação das atividades reconhecidas pelo sistema. Diferencia-se essa pesquisa de outros trabalhos relacionados, que monitoram atividades como o Apple Watch e Jawbone Pulseira UP, posto que efetua-se recomendação de acordo com o contexto identificado.

3. HARA-RS: *Recommender System based on Human Activity Recognition with Accelerometer*

Este trabalho propõe o HARA-RS, um Sistema de Recomendação baseado nos dados obtidos e analisados do modelo HARA para que, por meio das atividades realizadas por um

usuário monitorado em casa, o sistema seja capaz de recomendar, através das anomalias detectadas pelo modelo, que o indivíduo se dirija a um hospital próximo, que seu tempo de sono está irregular, problemas relacionadas ao sistema gastrointestinal, sedentarismo, entre outros tipos de recomendação. Escolheu-se o modelo HARA para o monitoramento e interpretação das atividades em razão de que o mesmo foi desenvolvido pelo nosso grupo de pesquisa.

A Figura 1 apresenta a arquitetura em camadas do HARA-RS. Os dados de atividades dos usuários são coletados e transmitidos, por *Bluetooth*, para o modelo HARA. Esses dados são interpretados de modo a definir o perfil e o contexto do usuário, que são usados pelo sistema de recomendação para apresentar conteúdos personalizados ao mesmo. Cada uma das camadas dessa arquitetura são descritas detalhadamente nas próximas subseções.

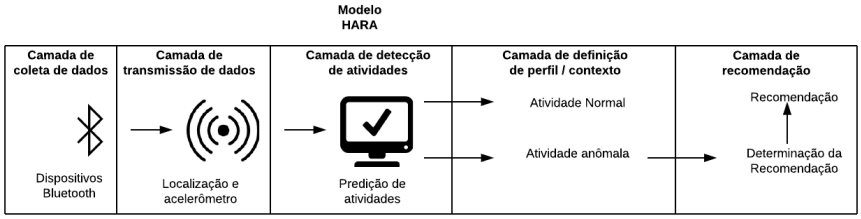


Figura 1. Sistema de recomendação - Visão geral

3.1. Camada de Coleta de dados

Esta camada é responsável por coletar os dados dos usuários que irão receber recomendações. A coleta dos dados é feita por meio de dispositivos *bluetooth* espalhados pela casa (ambiente assistido). Portanto, é possível mapear a intensidade do sinal enviado pelo sensor corporal e definir em qual cômodo a pessoa está. Neste artigo foram utilizados os dados já coletados previamente no experimento de [Amaral and Dantas 2017].

3.2. Cama de Transmissão

Esta camada realiza a transmissão dos dados de movimentação e localização coletados, os mesmos foram transmitidos para o microcontrolador e computador (Raspberry Pi 3) para que assim fossem interpretados pelo modelo HARA-RS.

3.3. Camada de Detecção das Atividades

Esta camada faz a interpretação dos dados coletados, com o intuito de classificar os cômodos da casa e reconhecer as atividades realizadas pelos usuários. Para identificar a localização do indivíduo na casa, dispositivos Bluetooth foram espalhados pelo ambiente. Tais dispositivos são responsáveis por verificar a intensidade do sinal em relação a distância do dispositivo com o acelerômetro usado no corpo do usuário. Nessa etapa, foi necessário configurar o sistema manualmente para que fique salvo em uma base de dados uma relação entre intensidade de sinal e cômodo da casa, para que assim possamos relacionar tais dados e fazermos as recomendações devidas.

Para o reconhecimento das atividades dos usuários, a técnica utilizada pelo modelo HARA é o Modelo Oculto de Markov (HMM), seguindo sua arquitetura. A escolha pelo

HMM se deve ao fato de ser possível realizar infinitas sequências com uma quantidade finita de termos. Analogamente, teremos uma sequência infinita de ações a partir de uma quantidade finita de estados. Ademais, é implementado um modelo que avalie a qualidade de contexto para decidir se os dados coletados devem ser descartados ou devem gerar um alerta.

3.4. Camada de Definição do Perfil e Contexto

Esta camada realiza a definição do perfil do usuário com base em dados previamente especificados como idade, sexo, peso, problemas de saúde, dentre outras características. Conjuntamente com os dados que foram interpretados pelo modelo HARA, o sistema define o contexto em que o usuário se encontra; que atividades ele realizou e se houveram atividades anômalas encontradas de acordo com sua rotina e com seu perfil, para assim estabelecer as possíveis recomendações para esse usuário.

O sistema de recomendação irá classificar a atividade a partir da interpretação dos dados, realizada pelo modelo HARA. Seguindo a devida classificação o sistema será capaz de reconhecer a atividade anômala e fazer a recomendação adequada. A partir do momento em que tais atividades anômalas se tornam corriqueiras, detectadas diversas vezes na semana ou no dia, o sistema irá detectar tal comportamento e efetuar um outro tipo de recomendação, dada a possível gravidade ou risco desses comportamentos. Logo, o sistema irá realizar uma filtragem baseada em conteúdo, definindo o perfil e o contexto de acordo com a atividade detectada.

3.5. Camada de Recomendação

Nesta camada é executado um Sistema de Recomendação considerando diferentes instanciações da arquitetura HARA-RS. A proposta deste trabalho é possibilitar que o sistema seja utilizado por diferentes demandas do domínio de saúde. Basicamente, os usuários finais, que receberão as recomendações podem ser classificados em: especialistas da área e pessoas com necessidade ou interesse em algum tipo acompanhamento.

Independentemente do usuário final, todas as outras camadas são utilizadas da mesma forma, apenas a camada de recomendação é configurada para filtragem dos conteúdos adequados ao usuário que receberá a recomendação. Essa configuração evita que conteúdos destinados a especialistas sejam recebidos por usuários comuns. Alguns exemplos de uso da arquitetura são descritos a seguir.

A arquitetura pode ser instanciada considerando que quem recebe a recomendação é **um usuário comum sem a presença de especialista médico**. Nesse caso a recomendação é feita de forma a indicar para o usuário soluções simples e diretas, levando-se em consideração a falta de um especialista para recomendações que não cabem a um sistema sem suporte de um profissional médico.

Outra forma de instanciação pode ser considerada quando tratamos de **um usuário comum com um especialista médico**. Assim, a recomendação é feita de forma a distinguir quais recomendações são feitas para o profissional e quais são destinadas ao usuário comum. Portanto, o especialista é capaz de, com base nas informações e recomendações recebidas, determinar qual a melhor forma de tratamento ou prevenção para aquele usuário.

Uma terceira instanciação pode ser verificada ao tratarmos de **um usuário menor de idade ou idoso com necessidades especiais**. Diante disso, a recomendação é feita de forma a enviar os dados de recomendação para o indivíduo responsável pelo usuário, uma vez que podemos tratar de crianças ou idosos que não possuem condições de receberem tais recomendações indicadas pelo sistema e realizar o que o sistema sugere.

Ademais, podemos tratar de outros casos nos quais usuários comuns têm o interesse de serem monitorados para algum propósito. Podemos considerar, por exemplo, **um usuário que deseja acompanhar sua queima de calorias**. Nesse cenário, a recomendação é feita com base na quantidade de calorias que o usuário deseja perder por dia, definindo assim mais atividades para serem realizadas caso não se alcance a meta esperada, de modo a auxiliar o mesmo em seu objetivo.

4. Avaliação preliminar

Para avaliar o modelo HARA-RS, foi conduzido um experimento preliminar, instanciando a arquitetura considerando um usuário que deseja monitorar seu gasto diário de calorias. Os dados foram monitorados, transmitidos e as atividades foram detectadas através do modelo HARA. Com base nessas atividades e nas informações dos usuários, o sistema identifica o perfil e o contexto do mesmo. As características de perfil identificadas foram: usuário do sexo masculino, idade entre 20 e 30 anos, que não possui, declaradamente, nenhum problema de saúde, e tem por objetivo gastar 1000kcal por dia. O contexto foi definido através do monitoramento e consolidação das atividades diárias do usuário. A Figura 2 lista as atividades identificadas e os seus respectivos gastos calóricos[1], compondo o contexto do usuário.

Atividade	Calorias gastas
Cozinhar	168 kcal/hora
Dormir	77 kcal/hora
Andar	5,5 kcal/min
Lavar louça	60 kcal/hora
Escrever	10 a 20 kcal/hora
Usar computador	95 kcal/hora
Escovar os dentes	80 kcal/hora
Comer	105 kcal/hora
Arrumar a cama	132 kcal/hora
Varrer	214 kcal/hora

Figura 2. Tabela de atividades e gastos energéticos

Portanto, de acordo com a Figura 3, que ilustra a instanciação desse modelo, os dados de movimentação e localização foram coletados pelos dispositivos *bluetooth* espalhados pelo ambiente assistido e pelos acelerômetros. Logo em seguida foram transmitidos para a camada de detecção de atividades que reconheceu e interpretou essas atividades e então, esses dados foram enviados para a camada de recomendação. Com os dados coletados nesse experimento, o perfil e o contexto do usuário foi consolidado, considerando

[1] *www.bemestar.globo.com*

um dia de atividades: **Dormir**(2 horas), **Arrumar a cama**(20 minutos), **Lavar louça**(1 hora), **Escovar os dentes**(10 minutos), **Usar o computador**(3 horas), **Cozinhar**(1 hora).

Foi identificado que o usuário gastou, considerando seu contexto, aproximadamente 676,33kcal. Desta forma, a camada de recomendação sugeriu que o usuário praticasse alguma atividade leve para que ele alcançasse o seu objetivo de gasto calórico.

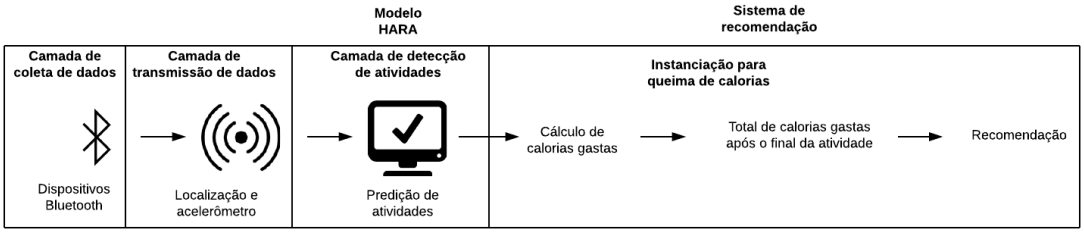


Figura 3. Instanciação para gasto de calorias - Visão geral

5. Considerações Finais

Este trabalho apresentou o modelo HARA-RS, capaz de identificar as atividades realizadas por pessoas em um ambiente assistido, recomendando ações para essas pessoas. Uma avaliação preliminar foi conduzida mostrando a viabilidade do modelo em indicar para o usuário medidas a serem tomadas conforme as atividades detectadas. O sistema também aborda a ideia de várias instanciações da arquitetura para atender diversas demandas de acordo com as possíveis necessidades até agora previstas. Como trabalhos futuros pretende-se avaliar o sistema em outros ambientes assistidos (domiciliar ou hospitalar), monitorando usuários com interesse em receber recomendações de conteúdo personalizado relacionados a saúde.

Referências

Amaral, W. and Dantas, M. (2017). Um modelo de reconhecimento de atividades humanas baseado no uso de acelerometro com qoc. In *Workshop de Iniciação Científica em Arquitetura de Computadores e Computação de Alto Desempenho (WSCAD-WIC 2017)*, pages 45–50. Porto Alegre: SBC.

Amay J Bandodkar, J. W. (2014). Non-invasive wearable electrochemical sensors: A review. *Trends in Biotechnology*, 32(7):363–371.

IBGE (2016). Projeção da população do brasil. *Revisão 2016*.

Jung, J., Ha, K., Kim, J. L. Y., and Kim, D. (2008). Wireless body area network in a ubiquitous healthcare system for physiological signal monitoring and health consulting. *International Journal of Signal Processing, Image Processing and Pattern Recognition*.

Lee, Y.-D. and Chung, W.-Y. (2009). Wireless sensor network based wearable smart shirt for ubiquitous health and activity monitoring. *Sensors and Actuators B: Chemical*, 140(2):390–395. https://doi.org/10.1016/j.snb.2009.04.040.

Wiesner, M. and Pfeifer, D. (2014). Health recommender systems: Concepts, requirements, technical basics and challenges. *International Journal of Environmental Research and Public Health (IJERPH)*, 11(3):2580–2607. https://doi.org/10.3390/ijerph110302580.

ReonV: a RISC-V derivative of SPARC LEON3 processor

Lucas Castro, Rodolfo Azevedo

lcbc.lucascastro@gmail.com, rodolfo@ic.unicamp.br

Institute of Computing – State University of Campinas (UNICAMP)
Av. Albert Einstein, 1251 – 13083-852 – Campinas – SP – Brazil

Abstract. *This paper endorses the importance of reuse and contribution for open source hardware, offering as example the development of a RISC-V soft-core processor, named ReonV, which was developed by reusing all IP cores from a well designed SPARC 32-bit processor changing only its ISA in order to obtain a fully operational RISC-V processor that inherits all other modules and Board Support Package (BSP) from the original SPARC core.*

1. Introduction

Computing community is very familiar with large scale software reuse, which is one of the advantages of developing open source software. However, when it comes to hardware, specially soft-core processors, reusage is often limited to small modules or extending the ISA with new instructions; normally, new cores development are made from the ground, creating the entire core and its Board Support Package (BSP). This approach raises unneeded obstacles such as recreating already existing modules, incompatibility problems when supporting new peripherals and adapting the BSP to new FPGA boards or running environments.

That motivated this research to build a new RISC-V [1] soft-core, named ReonV, reusing the IP cores from GRLIB, an IP Core Library containing a SPARC V8 processor named LEON3 [2, 3, 4]. The taken approach was to change only the pipeline of the processor in order to implement the RISC-V ISA instead of the original SPARC, inheriting all other already existing modules from LEON3, such as memory, memory controllers, peripherals support, debug support unit, BSP and synthesis scripts for a great number of FPGAs of different vendors. Thus, this work intended to show an example of how entire processors can be reused when developing new ones, adapting only the necessary to implement a different ISA. Even more, this study also aimed to make a contribution to the RISC-V community releasing a RISC-V version of a widely used and tested soft-core processor originally compliant with the SPARC ISA.

2. GRLIB and Leon3

The GRLIB IP Library is an integrated set of reusable IP cores, designed for system-on-chip (SOC) development released under GNU GPL license by Aeroflex Incorporated. The IP cores are centered around the common on-chip bus, and use a coherent method for simulation and synthesis. It also contains template designs for several FPGA boards and scripts in order to facilitate synthesis to supported devices [3, 4, 5].

GRLIB also contains LEON3, which is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. The model has a great number of configurable options and is specially aimed for system-on-a-chip (SOC) designs [3, 4, 6].

Figure 1 shows a representation of the LEON3 processor. It shows how flexile LEON3 is, since it can synthesized with a range of at least 5 crucial modules with minimalist options to a total of 17 highly configurable modules [3, 6].

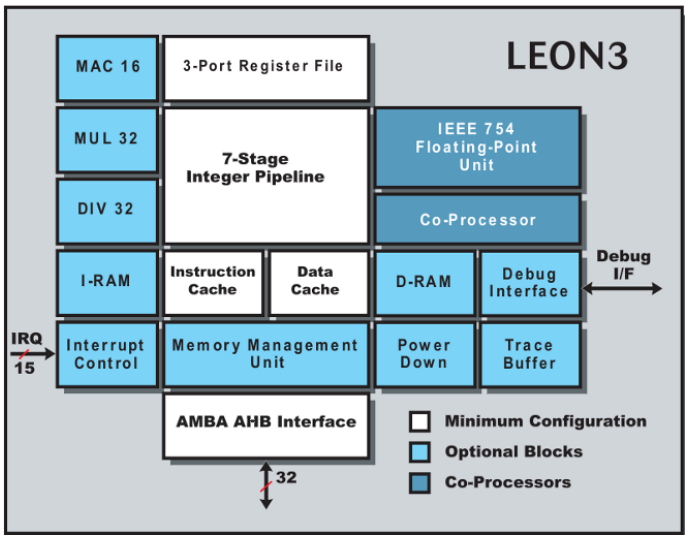


Figure 1. Representation of the LEON3 processor [7].

3. ReonV - A RISC-V version of the LEON3 processor

ReonV is the name given to the 32-bit RISC-V processor obtained after changing the LEON3's ISA from SPARC to RISC-V. It has an open repository[1] and is released under GNU GPL License [8].

ReonV was developed by reusing LEON3 processor and modifying its 7-stage integer pipeline (refer to Figure 1) in order to implement the ISA RV32I [1] without privileged instructions instead of the original SPARC V8, maintaining all other IP cores and resources provided by GRLIB untouched. With this, we aimed to obtain a RISC-V processor which provides all the support to synthesis, FPGAs and peripherals LEON3 has without the need to develop the whole core and its BSP. As a consequence, performance was left aside at this first moment, but there is space for future work in this regard.

4. Methodology and Results

As previously stated, the ReonV development was focused on reusing as much of the original SPARC LEON3 processor as possible. We managed to restrict changes to only modify the 7-stage Integer Pipeline in order to implement RISC-V ISA in the place of SPARC. Since there was no modification outside the pipeline, we intended to automatically gain the support already built for LEON3 to our RISC-V processor.

That allowed us to also use auxiliar software developed for LEON3 such as its synthesis tools, scripts and its debug monitor, named GRMON, to auxiliate the communication with the processor and its development [9]. GRMON has an interface which interacts with the processor's Debug Support Unit (DSU), allowing to easily load and run programs, dump memory, read registers values and other useful debug operations [6, 9].

[1]Repository available at: https://github.com/lcbcFoo/ReonV

While GRMON is a debug monitor for a SPARC processor which brings some expected incompatibility problems with the RISC-V ISA, it proved to be very useful to have such a tool on development stage for the new ISA.

4.1. Changing the ISA

The ISA change took advantage of the fact that the pipeline was already correctly implemented, thus any modifications could be easily tested, specially having the DSU and GRMON on hands. We modified the least necessary to have a working processor running the RV32I ISA, thus the pipeline continued with its original 7 stages (Fetch, Decode, Registers Access, Execute, Memory Access, Exception and Write Back). Most of the signals and modules were reused, with the proper modifications, but there were removed instructions and others completely included, accordingly to the new ISA.

We found problems with incompatible instructions or conventions, such as endianness and branches. ReonV development managed to deal with most of them, but branches required unwanted measures. Since branches on SPARC use already set status flags (Z, C, N, V) to only decide to take the branch or not [2] and, on the other hand, RISC-V branches read 2 registers, compute the condition and only then decide to take the branch or not [1], there was two main options to implement these instructions: make large changes on the pipeline structure and its stages or maintaining the current structure inserting unwanted stalls and without branch prediction. Since the project initially focused more on guaranteeing correctness than on performance, the second option was chosen, the processor's branch prediction was disabled and stalls were inserted on the pipeline to implement RISC-V branches.

After these changes, all instructions except branches take the same amount of cycles as their equivalent do on SPARC, as shown on Table 1. BP refers to Branch Prediction and the instructions abbreviations' follow the RISC-V specifications [1]. Store instructions take 2 cycles on both models, because the pipeline design needs one cycle to load the memory position and another to send the data.

Type	Instructions	Cycles needed (ReonV / Leon3)
PC relative	AUIPC	1 / not implemented
Control Flow	JALR, JAL	1 / 1
Conditional Control Flow	BEQ, BNE, BLT, BGE, BLTU, BGEU	4 (without BP) / 1 (with BP)
Memory Manipulation	LB, LBU, LH, LHU, LW	1 / 1
Memory Manipulation	SB, SH, SW	2 / 2
Logic and Arithmetic	ADD(I), SUB, XOR(I), OR(I), AND(I), SLL(I), SRL(I), SRA(I), SLT(I), SLTU(I), LUI	1 / 1

Table 1. ReonV instructions and the comparison of cycles needed to equivalent SPARC instructions implemented by LEON3.

The project used the Xilinx Vivado Design Suite [10] and the Nexys4DDR FPGA from AVNET [11] to synthesize and run the processor design. Table 2 compares the resource utilization between LEON3 and ReonV on Nexys4DDR with the same synthesis configuration with clock frequency of 40MHz . We can note that while ReonV uses less LUT, it demands more flip-flops. However, since the difference is very small, we consider both designs to have near the same size.

Resource	ReonV utilization	LEON3 utilization	Available at Nexys4DDR	Utilization % (Reonv / LEON3)
LUT	8665	8820	63400	13.67 / 13.91
LUTRAM	97	98	19000	0.51 / 0.52
FF	5346	4977	126800	4.22 / 3.93
BRAM	23	23	135	17.04
I/O	81	81	210	38.57
BUFG	11	11	32	34.38
PLL	5	5	6	83.33

Table 2. Comparison of board resources utilization between ReonV and Leon3. Available collumn refers to the Nexys4ddr FPGA

4.2. Correctness Benchmarks

In order to certify the correctness of the processor, we set up an automated compilation and running environment to execute tests from the set of benchmarks from WCET [12], adapted to ReonV, since each one of them clearly states what structures it validates.

ReonV was tested and correctly executed each of the described programs of Table 3. All tests were executed running the processor's synthesised design as described on Section 4.1. The source code of each test, as all other piece of software needed to run these benchmarks, are available at the project repository [8] and were compiled with the official RISC-V Toolchain [13]. Table 3 also shows which structures are tested on each program accordingly to the following label:

- L - Has loops
- N - Has nested loops
- B - Bit Manipulation
- A - Uses arrays or matrices
- R - Has recursion

5. Discussion

All tests executed successfully and we consider that the pipeline change was successful and a RV32I processor was obtained reusing the LEON3 IP cores with very few modifications.

It's also important to note that the tested version of ReonV still does not implement a few important instructions such as multiplication, division and FP operations. However,

File	Description	L	N	A	R	B
bs.c	Binary search	✓		✓		
bsort.c	Bubble sort	✓	✓	✓		
cover.c	Distinct program flow	✓				
expint.c	Integer exponentiation	✓	✓			
fac.c	Factorial	✓			✓	
fibcall.c	Fibonacci	✓				
insertsort.c	Insertion Sort	✓	✓	✓		
complex.c	Nested loops	✓	✓			
matmult.c	Matrix Multiplication	✓	✓	✓		
ndes.c	Bit Manipulation	✓		✓		✓
prime.c	Prime check	✓				
qsort-exam.c	Quick Sort	✓	✓	✓		
recursion.c	Recursion				✓	

Table 3. Test programs used to validate ReonV. All of them are adaptations of the WCET benchmarks [12].

LEON3 already have modules for executing these instructions [6], suggesting that extending the current implemented ISA will only be another step of hardware reuse and pipeline modification.

Moreover, reusing LEON3 IP core also allowed to use a debug monitor and a DSU since the beginning of the development what was a significant advantage to the project. Also, there was conviction that the other modules were correct since they were not modified and LEON3 is a well established and largely used processor, restricting eventual debug analysis to the changes on pipeline.

Although problems with incompatibility over SPARC and RISC-V ISAs existed and made the initial version of ReonV poorly efficient, specially with branches, it can be improved on future work, some already on going at Unicamp to improve branch instructions and enable the existing processor's MUL, DIV and FP units, which are currently disabled. Furthermore those incompatibilities are ISA specific and may not occur on related development approaches for other architectures, for example MIPS and RISC-V would not face the same branches incompatibility since MIPS branches are similar to RISC-V ones [1, 14]. Thus, the approach proposed here did not bring any unavoidable or irreversible disadvantage for ReonV development.

6. Conclusion

This work concludes releasing an open source VHDL model of a operational RISC-V soft-core developed reusing another soft-core IP Core Library. This is a successful example of how open source hardware has potential for reuse and what advantages this approach may bring to development.

During development, incompatibility and efficiency problems existed, however they were expected since both ISAs have different instructions and conventions. Furthermore, none of them were irreversible and can be fixed or at least minimized on future work. There is also room for forthcoming projects expanding the current instructions

supported by ReonV, such as implementing multiplication, division, float point and the privileged instructions.

Finally, this work offers to the RISC-V community the possibility for further development of a RISC-V soft-core processor which brings with it all the support of a well established IP Core Library as a consequence of its development focus: Hardware Reuse.

7. Acknowledgments

We acknowledge all the support given to this research by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) (process 2017/04018-2), CNPq and CAPES (process 2966/2014).

References

[1] *The RISC-V Instruction Set Manual - Volume I: User-Level ISA - Document Version 2.2*. RISC-V Foundation, 1300 Henley Court, Pullman, WA 99163, 509.334.6306. URL `https://riscv.org/specifications/`. Editors Andrew Waterman and Krste Asanovic. Accessed on 07/2018.

[2] *The SPARC Architecture Manual, Version 8*. Sparc International Inc., 535 Middlefield Road - Suite 210 - Menlo Park - CA 94025.

[3] *GRLIB IP Library User's Manual - Version 2018.1*. Cobham Gaisler AB, Kungsgatan 12, 411 19 Gothenburg, Sweden, . URL `www.cobham.com/gaisler/products/grlib/grlib.pdf`. Accessed on 07/2018.

[4] Cobham Gaisler AB. Grlib ip library. URL `www.gaisler.com/index.php/products/ipcores/soclibrary`. Accessed on 07/2018.

[5] *GRLIB IP Core User's Manual - Version 2018.1*. Cobham Gaisler AB, Kungsgatan 12, 411 19 Gothenburg, Sweden, . URL `www.cobham.com/gaisler/products/grlib/grip.pdf`. Accessed on 07/2018.

[6] *Configuration and Development Guide - Version 2018.1*. Cobham Gaisler AB, Kungsgatan 12, 411 19 Gothenburg, Sweden, . URL `www.cobham.com/gaisler/products/grlib/guide.pdf`. Accessed on 07/2018.

[7] Sven Åke Andersson. Leon3 32-bit processor core. URL `www.rte.se/blog/blogg-modesty-corex/leon3-32-bit-processor-core/1.5`. Accessed on 07/2018.

[8] Lucas Castro. ReonV - A RISC-V version of LEON3. URL `www.github.com/lcbcFoo/ReonV`. Accessed on 07/2018.

[9] *GRMON2 User's Manual*. Cobham Gaisler AB, Kungsgatan 12, 411 19 Gothenburg, Sweden, . URL `www.gaisler.com/doc/grmon2.pdf`. Accessed on 07/2018.

[10] *Vivado Design Suite User Guide - Release Notes, Installation, and Licensing*. Xilinx, Inc. URL `www.xilinx.com`.

[11] *Nexys 4™ FPGA Board Reference Manual*. Digilent Inc, 1300 Henley Court, Pullman, WA 99163, 509.334.6306.

[12] Mälardalen WCET research group. Wcet benchmark. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`. Acessado em 30/04/2018.

[13] RISC-V Foundation. Software Tools. URL `www.riscv.org/software-tools`. Accessed on 07/2018.

[14] *MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual*. Imagination Technologies LTD.

Analysis of a Mechanism to Reduce Prefetcher-Caused Cache Pollution

Arthur Mittmann Krause[1], **Francis Birck Moreira**[1],
Eduardo Henrique Molina da Cruz[2], **Philippe Olivier Alexandre Navaux**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{amkrause,fbmoreira,navaux}@inf.ufrgs.br

[2]Campus Paranavaí – Instituto Federal do Paraná (IFPR)
Rua José Felipe Tequinha, 1400 – CEP 87703-536 – Paranavaí – PR – Brazil

eduardo.cruz@ifpr.edu.br

Abstract. *Caching and prefetching are effective techniques used to mask the latencies of the memory subsystem on current processors, but the prefetcher may jeopardize performance by polluting the cache. This paper presents techniques to reduce cache pollution and analyses the effects of a mechanism that reduces the priority of prefetched blocks aiming to mitigate pollution, exploring the reasons for performance improvements or degradations on different applications.*

1. Introduction

The speed of processors is evolving at a much faster pace than that of the memory. Previously with the increase of processor frequencies, and currently with the rapid expansion of the number of cores, modern processors demand more bandwidth than memories can supply, and the trend is that this problem will only increase. This characteristic of current architectures is known as the memory wall, characterized by the high relative memory-to-CPU latency causing the performance of a system to be mainly determined by the time it takes to bring the data from the memory to the CPU.

To mitigate this performance limitation, designers sought to hide the memory latency. CPU caches and prefetchers are commonly employed to do so. A CPU cache is a small and fast memory that resides between the CPU and the main memory. It stores data that was recently requested by the processor so that, if it is needed again, the processor will not need to wait for the slow DRAM. The prefetcher tries to predict the addresses that will be requested by the processor in the future and loads the data into the cache, hiding the latency of the DRAM request.

The smart management of the stored data in the cache is essential for efficient usage. When data that will not be useful replace data that would be needed by the processor, a situation termed *cache pollution* is defined. The prefetcher is a relevant source of cache pollution. When it prefetches data into the cache, it may displace valuable data in favor of the prefetched blocks that are usually not as crucial for performance.

Many authors suggest modifications to the prefetcher or to the cache insertion policy aiming to reduce the pollution caused by the prefetcher. We present some of the most influential contributions on this matter and implement one of the mechanisms on a simulator, extending the analysis of its effects on the memory subsystem.

2. Cache Pollution

When a cache evicts valuable data to receive useless data, it suffers from *cache pollution*. This situation may happen under different circumstances. For instance, a thread requesting data may cause the cache to evict blocks from another thread. We are interested in the case where an inaccurate prefetcher fetches useless data to the cache. This situation is far worse because the pollution generates more cache misses, which will trigger more prefetches, which in turn will increase the pollution even more.

A prefetcher that is not aggressive enough will not yield all the potential performance in its capability. Blocks may be loaded too late, thus not effectively hiding latency. On the other hand, a prefetcher that is too aggressive can jeopardize system performance. Blocks may arrive in the cache too early and, by the time the processor requests it, the cache already evicted it. Blocks might also not be needed at all, resulting in low prefetcher accuracy. Prefetchers that fetch blocks into the cache are the traditional approach in modern processors. They might suffer from poor timing or low accuracy. These can result in cache pollution, degrading the performance of cache-sensitive applications.

With these concepts in mind, we can infer that a prefetch pollutes the cache if the processor does not request its data. In Figure 1, we illustrate the distribution of processor requests to prefetched blocks of all SPEC_CPU 2006 [Henning 2006] applications, using simulations detailed in section 4.1.

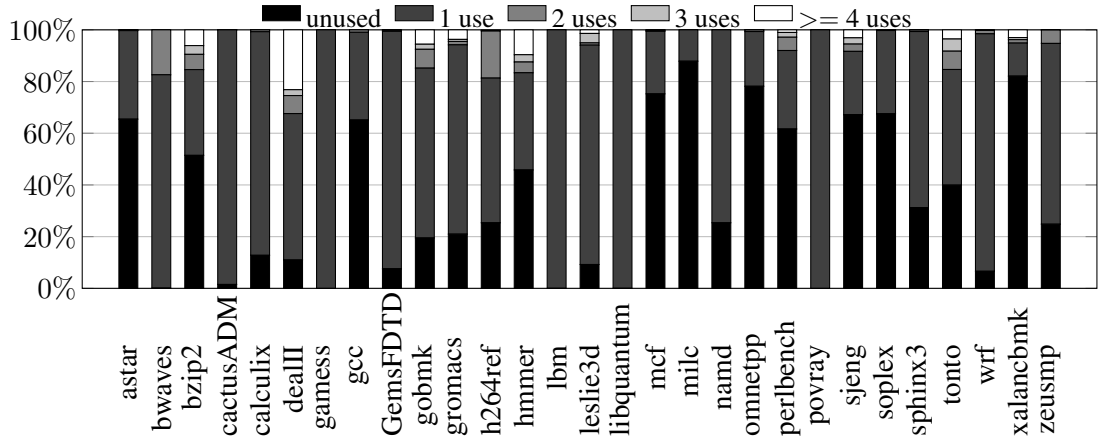


Figure 1. Distribution of line usage per prefetch line.

As seen in the Figure, the effectiveness of the prefetcher may vary depending on the application. The application *astar* does not use over 65% of the prefetches, and only uses prefetches once when they are accurate. On the other hand, the application *hmmer* has a large variance on the number of uses per prefetch line, using over 10% of its prefetches four or more times.

3. Related Work

There are two main approaches to mitigate cache pollution. Some mechanisms modify the prefetcher, others change the cache insertion policy, and certain papers use a combination of both. We analyzed the most influential mechanisms in the literature, but only a fraction of them is described in this paper due to space restraints.

Khan et al. [Khan et al. 2014] describes a mechanism that reduces the cache pollution introduced by prefetch data. It works by using software prefetching instead of hardware prefetching. They analyze the memory accesses using sampling and identify what they call delinquent loads, which are load instructions that often cause cache misses. For each delinquent load, they analyze the stride pattern and insert software prefetches to prefetch the data before the execution of the load instruction. Khan et al. [Khan et al. 2015] later extends this work by developing a runtime system that can dynamically combine any available software and hardware prefetching technique.

Jimenez et al. [Jiménez et al. 2012] propose an adaptive prefetch mechanism for the IBM Power7 processor. The Power7 allows the software to configure the behavior of prefetching, including the prefetch depth (how many lines in advance to prefetch), prefetch on stores (whether to prefetch store operations) and whether to prefetch streams with a stride larger than one cache block. They implement their mechanism in both a user-level runtime library and a privileged-level implementation in the operating system.

The Sandbox prefetcher [Pugsley et al. 2014] is a mechanism to determine at runtime the most appropriate prefetch mechanism for the executing program. Sandbox Prefetching evaluates prefetchers at runtime by adding the prefetch address to a Bloom filter, instead of fetching the data into the cache. In subsequent cache accesses, the mechanism verifies the Bloom filter to check if the prefetcher under evaluation could accurately prefetch the data and test for the existence of prefetchable streams.

Seshadri et al. [Seshadri et al. 2015] observe a pattern across multiple applications, in which prefetched blocks are used only once 95% of the time. As a result of this observation, the authors introduce a mechanism called ICP, which is a combination of two techniques. The first is ICP-D, which demotes a prefetched block to the LRU position when it receives its first demand request, predicting that it will not be reused and allegedly reducing the cache pollution generated by the prefetcher. The authors also propose ICP-AP. ICP-AP monitors the accuracy of each prefetch stream and modifies the cache replacement policy depending on the stream accuracy. If the cache services a prefetch request, ICP-AP promotes the block to a high priority only if it predicts the prefetch stream to be accurate. Otherwise, the block stays in the same position on the queue. When the cache is unable to service a prefetch request, ICP-AP inserts the line read from the next memory level with the highest priority only if it predicts the prefetch stream to be accurate. Else, it places the block at the lowest priority.

4. Simulation

To verify the validity of the presented solutions, we performed a simulation of one of the most recent and influential mechanisms, ICP, which the authors show to outperform previous methods. Although the comparative analysis on the paper is excellent, the effects of the mechanism on the memory hierarchy and how it obtains performance improvements are not fully explored. Therefore, we aim to extend their analysis and investigate the effects of the mechanism in the cache.

4.1. Methodology

We implement the modifications from ICP on a x86 event-driven simulator [Seshadri 2014]. We model a single core system with a configuration represented in Table

1. We tried to replicate the system used in the original paper, on the same simulator used by the authors. The proposed mechanisms were tested separately (ICP-D and ICP-AP) and combined (ICP) with the benchmarks from SPEC 2006 [Henning 2006], which the authors used on their paper. We exclude the benchmarks with a count of misses per 1000-instructions (MPKI) of less than 5 on the L2 cache from our analysis since they do not exert enough pressure on the LLC, and thus the mechanism has little effect over them. We collected traces with 200 million instructions from the most representative portions of each application [Sherwood et al. 2002].

Table 1. Default simulation parameters

Core	in-order/out-of-order x86
L1-D Cache	Private, 32KB, 2-way associative, 1 cycle load-to-use latency
L2 Cache	Private, 256KB, 8-way associative, 8 cycles load-to-use latency
L3 Cache	Shared, 1MB, 16-way associative, 27 cycles load-to-use latency
Prefetcher	16 streams/core, Degree = 4, Distance = 24
M. Controller	First-Come-First-Served-Scheduler
Memory	8 banks, 200-cycle bank access latency

4.2. Results

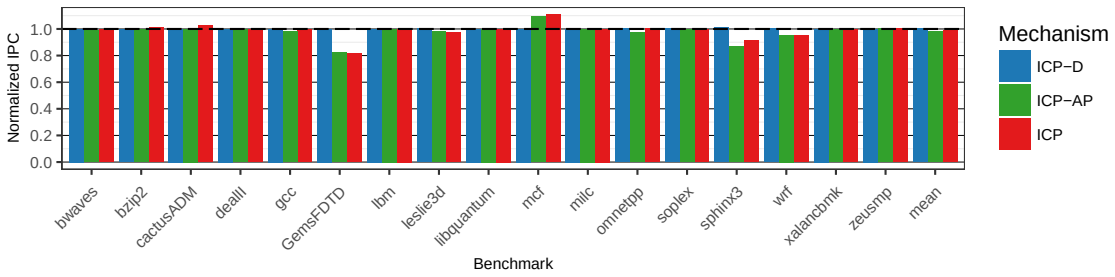


Figure 2. Effect on IPC of the different variations of ICP

Figure 2 shows the instructions-per-cycle (IPC) of the benchmarks under different versions of the mechanism normalized to the LRU policy. Our results show that ICP degrades the overall performance, with few exceptions. On average, ICP-D does not improve IPC, ICP-AP reduces performance by 2.3%, and their combination lowered the performance by 1.7%. However, on *mcf*, the accuracy-aware prefetch insertion increases the IPC by more than 8%. In combination with ICP-D, the mechanism improves performance by 10%. The only other application profiting from the mechanism is *cactusADM*. In *cactusADM* neither ICP-D or ICP-AP alone deliver any performance upgrade, but their combination yields 3% more instructions-per-cycle. ICP-D never improves or degrades performance by more than 1%. Many applications suffer performance degradation under ICP-AP. Remarkably, *GemsFDTD* and *sphinx3* lose 18% and 13.5% of IPC, respectively.

These performance variations are compatible with the changes in the number of misses per 1000-instructions (MPKI) shown in Figure 3. When applications obtain reductions on MPKI, they exhibit better performance. If the mechanism increases the occurrence of misses, it diminishes performance. This is untrue for *libquantum* and *dealII*, which do not pressure the LLC. In these benchmarks, a reduction in the frequency of misses yields no significant performance improvement.

On average, the mechanism does not succeed on its goal of reducing cache misses. This reduction was supposed to originate from a mitigation on prefetcher-caused cache

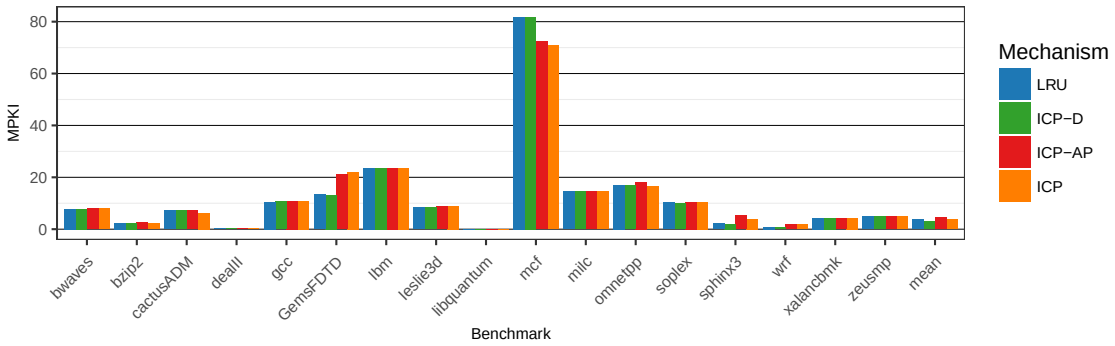


Figure 3. LLC misses per 1000-instructions under the distinct mechanisms

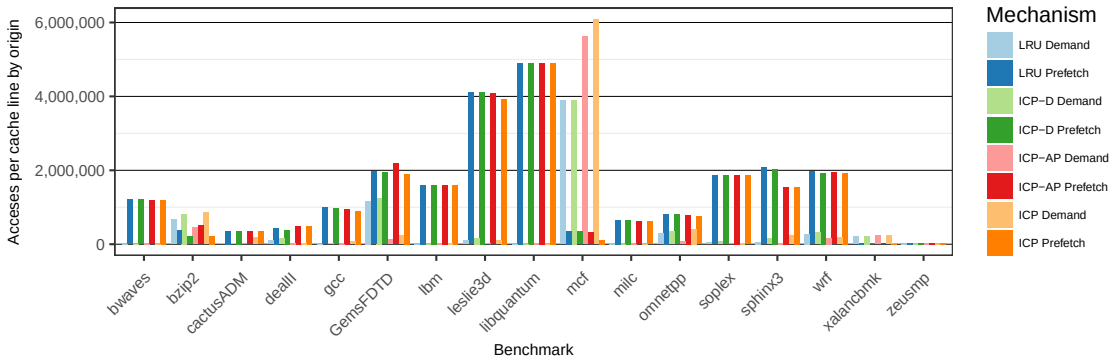


Figure 4. Number of demand requests for demand-fetched and prefetched lines

pollution, by avoiding evictions of useful demand-fetched blocks removing prefetched blocks as soon as they are used. Our results show that for most applications, a significant portion of accesses are to prefetched blocks, as seen in Figure 4. When the mechanism demotes these blocks from the cache, it generates more cache misses. Applications that have a less predictable memory access pattern, such as *mcf*, perform better under the mechanism because the amount of accesses to prefetched blocks is lower and thus the mechanism effectively reduces prefetcher-caused cache pollution when these blocks are removed earlier from the cache.

The results obtained on this experiment are not consistent with the results presented on the original paper [Seshadri et al. 2015]. This can be due to multiple variables on the methodology, specially on the origin of the traces. The authors used traces of one billion instructions, while we used only 200 million. Both traces were generated using Simpoints, but the exact parameters the authors used are not known, which potentially could make the traces for the same benchmakrs drastically different. Some details of the implementation were not specified on the original paper, specially the threshold of accuracy that is used to classify a prefetcher as accurate or inaccurate. The authors also used the *art* benchmark from SPEC 2000, in which their simulation showed a 24% improvement in performance, much greater than in any other benchmark.

5. Conclusion

Prefetching is an important technique to reduce the memory access latency. However, the prefetcher can jeopardize the performance of the cache when it causes the removal of valuable data from it, replacing them with blocks that are less crucial for performance. The state-of-the-art techniques to reduce this cache pollution modify the cache replace-

ment policy to prioritize demand requests over speculative prefetch requests.

Our work shows that these techniques can improve the performance of some applications and degrade it of others, depending heavily on the application's memory access pattern and working set size. On ICP, the performance is improved in applications where the stream prefetcher is inaccurate, up to 10% in *mcf*. On the other hand, the mechanism increases cache misses when the prefetcher is accurate, compromising the performance by 18% in *GemsFDTD*.

As a future work, we intend to test ICP and other mechanisms with different configurations, such as prefetcher aggressiveness, cache sizes, and accuracy threshold on a multicore processor running parallel applications.

References

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Jiménez, V., Gioiosa, R., Cazorla, F. J., Buyuktosunoglu, A., Bose, P., and Connell, F. P. O. (2012). Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *Parallel Architectures and Compilation Techniques (PACT)*.

Khan, M., Laurenzano, M. A., Mars, J., Hagersten, E., and Black-Schaffer, D. (2015). AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance. In *Parallel Architectures and Compilation Techniques (PACT)*.

Khan, M., Sandberg, A., and Hagersten, E. (2014). A case for resource efficient prefetching in multicores. In *International Conference on Parallel Processing (ICPP)*.

Pugsley, S. H., Chishti, Z., Wilkerson, C., Chuang, P. F., Scott, R. L., Jaleel, A., Lu, S. L., Chow, K., and Balasubramonian, R. (2014). Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers. In *International Symposium on High Performance Computer Architecture (HPCA)*.

Seshadri, V. (2014). Source code for Mem-Sim. Retrieved May 30, 2018 from www.ece.cmu.edu/ safari/tools.html.

Seshadri, V., Yedkar, S., Xin, H., Mutlu, O., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. (2015). Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):51.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. *ACM SIGARCH Computer Architecture News*, 30(5):45–57.

Análise Comparativa de MPI e OpenMP em Implementações de Transposição de Matrizes para Arquitetura com Memória Compartilhada

Lucas R. de Araujo, Crístian M. Weber
Fernando E. Puntel, Andrea S. Charão, João Vicente F. Lima

[1] Laboratório de Sistemas de Computação
Universidade Federal de Santa Maria

Abstract. *This work presents two implementations of a matrix transposition application, with MPI or OpenMP, to investigate the impact of these tools on performance in shared memory architecture. The results obtained from the analyzes show which variables influence the performance of the application and in what form they cause this impact.*

Resumo. *Este trabalho apresenta duas implementações de uma aplicação de transposição de matrizes, com MPI ou OpenMP, para investigar o impacto dessas ferramentas no desempenho em arquitetura de memória compartilhada. Os resultados obtidos a partir das análises realizadas mostram quais as variáveis influenciam no desempenho da aplicação e de que forma causam esse impacto.*

1. Introdução

A computação de alto desempenho motiva programadores a explorarem técnicas de paralelização de seus códigos para obtenção de execuções mais velozes. Um dos desafios da programação paralela se refere à maneira como os processos e/ou *threads* se comunicam. Esse fator é dependente da arquitetura em que o código a ser desenvolvido será executado. Arquiteturas de memória compartilhada entram nesse contexto e funcionam de maneira que todos os processos e *threads* possuem acesso direto aos dados.

No âmbito de programação paralela, duas ferramentas que se destacam são MPI e OpenMP. OpenMP é originalmente voltada para arquiteturas paralelas com memória compartilhada, enquanto MPI foi concebida para arquiteturas com memória distribuída, com possíveis otimizações em arquiteturas multiprocessadas. Em arquiteturas comuns em servidores atuais, com múltiplos processadores ou núcleos compartilhando memória, é possível executar aplicações desenvolvidas com OpenMP e/ou MPI.

Em um trabalho anterior [de Araujo et al. 2018], realizou-se experimentos com uma aplicação de computação científica denominada Incompact3d[1], que utiliza MPI e foi originalmente concebida para execução em *clusters*. Ao executá-la em arquitetura com memória compartilhada, que é atualmente mais acessível a alguns de seus usuários, observou-se *overhead* significativo na comunicação entre processos, durante sequências de operações de transposição de matrizes. Alguns autores já compararam MPI e OpenMP e revelaram casos em que OpenMP ou MPI obtiveram melhor desempenho. Isso motivou uma investigação comparativa de desempenho entre MPI e OpenMP para a aplicação em questão, em arquitetura com memória compartilhada.

[1] https://www.incompact3d.com

Neste trabalho, busca-se explorar essa possibilidade por meio de dois programas que reproduzem as sequências de transposições realizadas no Incompact3d: um com MPI e outro com OpenMP. Por meio dessas implementações, objetiva-se obter métricas de desempenho sobre o uso de MPI e OpenMP, investigando causas de alguma implementação apresentar desempenho inferior a outra, principalmente no que se refere à comunicação de dados. Optou-se por externalizar as etapas do Incompact3d devido ao grande porte da aplicação (aproximadamente 30000 linhas de código) e, dessa forma, evitar inúmeras alterações sem ter indícios prévios de melhoria no desempenho ao implementar a aplicação de outra forma.

2. Trabalhos Relacionados

Diversos trabalhos na literatura abordam a análise de aplicações paralelas utilizando MPI e OpenMP de forma comparativa. Uma implementação bastante comum é a híbrida, que utiliza recursos das duas bibliotecas para paralelização. A implementação híbrida é a que tem aparecido em mais trabalhos ao lado da implementação em MPI. Um possível motivo para uso reduzido de OpenMP puro é devido a maneira como as *threads* acessam a memória nessa implementação (*fine-grained memory access*), então o desempenho acaba sendo afetado [Jin et al. 2011].

Em um dos trabalhos que exploram OpenMP puro [Krawezik e Cappello 2003], os autores avaliam alguns *benchmarks* implementados em MPI e em 3 estilos de implementação OpenMP: *loop level*, *loop level with large parallel sections* e SPMD (*Single Program Multiple Data*). Eles concluem que o estilo de implementação em OpenMP é determinante para um melhor desempenho da aplicação em relação a aplicação em MPI.

Já em um trabalho mais recente [Xu e Zhang 2015], os autores utilizaram uma arquitetura SMP (*Symmetric Multiprocessing*) e obtiveram êxito ao conseguir melhor desempenho usando uma abordagem híbrida. Esse caso mostra a maneira mais comum em que a implementação híbrida é utilizada: com MPI para comunicação entre os nós da máquina e OpenMP para comunicação dentro dos próprios nós. Basicamente, o MPI assume a comunicação em um nível de sistema mais distribuído e o OpenMP é utilizado na paralelização em nível compartilhado [Jin et al. 2011].

Apesar da predominância da implementação híbrida em trabalhos recentes, optou-se pela implementação em OpenMP puro (em comparação com MPI puro) pelo fato das execuções serem realizadas em uma arquitetura de memória compartilhada e possivelmente beneficiarem-se de tal arquitetura. Autores também citam implementação pelo compilador, implementação pouco intrusiva e a facilidade para controlar as diretivas como motivos para utilização do OpenMP [Bird et al. 2017].

3. Desenvolvimento

As duas implementações foram desenvolvidas na linguagem C e têm o mesmo esqueleto, que está representado em forma de fluxograma na Figura 1. Na etapa inicial, a aplicação cria uma estrutura de dados referente à distribuição das matrizes entre os processos (MPI) ou *threads* (OpenMP) e então aloca e inicializa as matrizes para sequência da execução. Na fase seguinte, ocorrem diversas iterações em que realizam-se etapas de cálculo e de transposição, alternadamente. É na etapa de transposição que se localizam as principais diferenças entre as implementações e que, devido às particularidades de cada uma, são

propagadas alterações nas outras partes do código, principalmente nas variáveis utilizadas e no compartilhamento de dados entre elas.

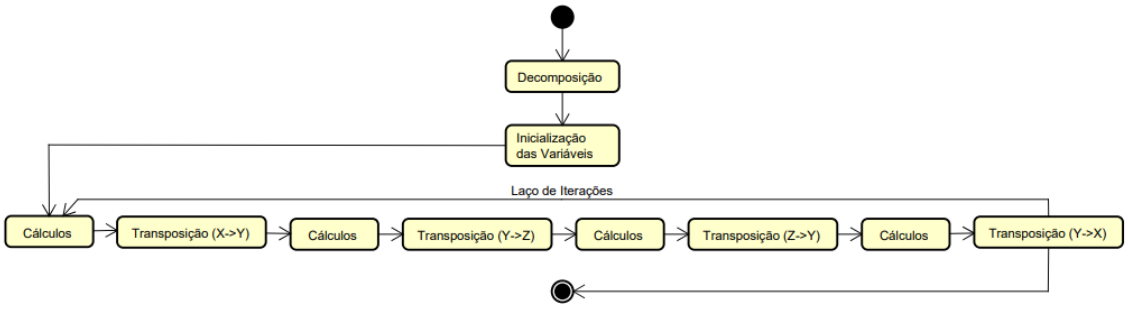


Figura 1. Fluxograma representando a execução da aplicação.

O desenvolvimento utilizando MPI teve como base o código do Incompact3d, passando por um processo de simplificação, mas sem perder a essência do código. Seu processo de transposição, assim como no Incompact3d, acaba tendo grande auxílio de estruturas de dados e de funções que o MPI disponibiliza. Os comunicadores criados durante a decomposição são estruturas que facilitam a identificação dos processos que devem compartilhar dados entre si em determinada transposição. A utilização de MPI_Alltoallv para o compartilhamento de dados acaba ocultando o funcionamento da troca de informações entre os processos, que se dá pelo método de passagem de mensagens.

Já a implementação desenvolvida em OpenMP teve o código baseado na aplicação de mesmo esqueleto em MPI. A transformação do código passou pela retirada e substituição de funções e estruturas de dados do MPI e a inserção de uma estrutura de dados compartilhada. O processo de troca de dados para realização da transposição fica bem mais claro no próprio código da aplicação e funciona de forma em que as *threads* inserem todos os seus dados em uma estrutura de dados compartilhada. Há a utilização de uma barreira para que todos os dados necessários estejam disponíveis e então as *threads* acessam a estrutura de dados em memória compartilhada para leitura dos dados necessários. Nesse processo de leitura dos dados, as *threads* utilizam índices calculados no particionamento original da matriz para acesso às posições corretas.

Ambas as implementações dispõem de um arquivo de parâmetros que facilita a instrumentação de cada execução. Nesse arquivo é possível definir o tamanho da matriz, o número de processos ou *threads*, a divisão da matriz, o número de iterações e o número de repetições dos laços de cálculos. Parte dessas variáveis serão exploradas nas execuções apresentadas na seção 5.

4. Metodologia

O ambiente de execução da aplicação foi um servidor NUMA SGI UV2000 com 48 núcleos distribuídos em 8 processadores Intel® Xeon® CPU E5-4617 (2.90GHz de frequência) de 6 núcleos cada e 512GB de memória RAM. A memória cache da máquina se organiza da seguinte forma: cache L1 de dados e de instruções (32K cada), cache L2 de 256K e cache L3 de 15360K.

Os testes inicialmente visaram medidas de tempo obtidas através da própria

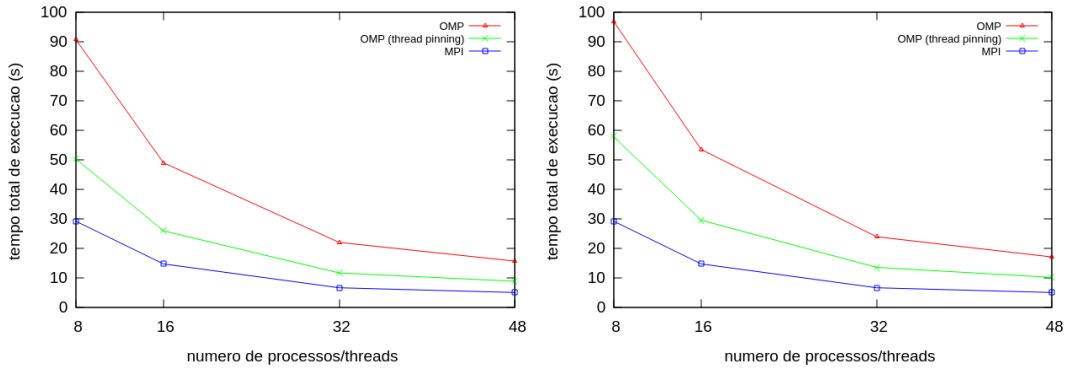
aplicação com a utilização de *clock_gettime* da biblioteca time.h. Posteriormente utilizou-se a ferramenta *perf*[2] (versão 4.9.30) para extração de dados de eventos de *software* e de *hardware* da máquina.

A ferramenta Score-P[3] (versão 4.0) foi utilizada para obter *logs* de informações do tempo ocupado pelas diretivas de barreira na implementação em OpenMP. Esses *logs* foram lidos na ferramenta Cube[4] (versão 4.4). Na fase final de testes, a aplicação LIKWID[5] (versão 4.3.0) também foi utilizada para a tarefa de *thread pinning* da implementação OpenMP através de `likwid-pin`.

5. Resultados e Discussão

Após a implementação e revisão de ambos os códigos iniciou-se a fase de execuções. As execuções iniciais foram usadas para avaliar o desempenho das duas implementações, variando o número de cálculos e o número de processos ou *threads*. Os 2 parâmetros de cálculo utilizados foram 5 e 10, enquanto o número de processos ou *threads* variou nos valores 8, 16, 32 e 48. Já o número de iterações manteve-se fixo em 100, assim como tamanho da matriz manteve-se em 4194304 nós (128*256*128).

Os gráficos nas Figuras 2(a) e 2(b) representam as execuções da aplicação em MPI e OpenMP com as condições citadas acima. Além disso, os gráficos trazem um caso de OpenMP com *thread pinning* que será abordado a seguir. Os gráficos mostram que há um desempenho inferior da implementação em OpenMP em relação à desenvolvida anteriormente utilizando MPI. Enquanto a aplicação em MPI varia de 29,2 segundos com 8 processos para 5,02 segundos com 48 processos, a aplicação em OpenMP (sem *thread pinning*) varia de 90,8 segundos com 8 *threads* para 15,5 segundos com 48 *threads*. Apesar da diferença de tempo entre as implementações, o *speedup* com o aumento do número de processos/*threads* nas duas implementações e nos dois casos onde há variação da quantidade de cálculos é bem semelhante e varia entre 5,7 e 5,8.



(a) Parâmetro de cálculos: 5 repetições (b) Parâmetro de cálculos: 10 repetições

Figura 2. Tempo de execução por número de processos (MPI) ou *threads* (OpenMP).

[2]https://perf.wiki.kernel.org/index.php/Main_Page
[3]http://www.vi-hps.org/projects/score-p/
[4]http://www.scalasca.org/software/cube-4.x/
[5]https://github.com/RRZE-HPC/likwid

A partir do desempenho inferior da aplicação em OpenMP, utilizou-se a ferramenta *perf* para realizar o monitoramento de alguns eventos da máquina durante a execução da aplicação. Inicialmente, o objetivo de utilizar o *perf* era monitorar a memória cache durante a execução, devido ao grande volume de acesso e escrita em variáveis que ocorre no decorrer da execução. Porém, os resultados coletados em relação à memória cache não justificavam os tempos de execução obtidos anteriormente, visto que eram bastante semelhantes.

Dessa forma, outras duas variáveis observadas nos *logs* do *perf* apresentaram uma discrepância maior: *context switches* e *cpu-migrations*. A Tabela 1 apresenta as duas medidas para o caso utilizando 5 cálculos. É possível observar na maioria dos casos um aumento de *context switches* e de *cpu-migrations* da aplicação em MPI para a aplicação em OpenMP.

Implementação	Número de Processos	context-switches	cpu-migrations
MPI	8	6442	80
MPI	16	15995	127
MPI	32	49957	251
MPI	48	126206	400
OpenMP	8	80876	15
OpenMP	16	100425	771
OpenMP	32	100003	5951
OpenMP	48	104063	5404
OpenMP (*thread pinning*)	8	96303	59
OpenMP (*thread pinning*)	16	126223	68
OpenMP (*thread pinning*)	32	124878	84
OpenMP (*thread pinning*)	48	116888	100

Tabela 1. *context-switches* e *cpu-migrations* (Parâmetro de cálculos: 5)

Como possível solução para o problema de *cpu-migrations* elevado na aplicação em OpenMP, buscou-se utilizar a ferramenta `likwid-pin` que permite ao usuário o *thread pinning* sem alterar o código da aplicação. Os resultados obtidos com a utilização de `likwid-pin` estão presentes nas quatro últimas linhas da Tabela 1 e apresentam diminuição para casos com 16, 32 e 48 *threads*. Ainda que no caso de 8 *threads* haja aumento no número de *cpu-migrations*, na Figura 2(a) é possível observar que o tempo de execução utilizando `likwid-pin` também é inferior ao caso em que não se utiliza.

O melhor desempenho nas execuções utilizando `likwid-pin` são consequência da redução de troca de núcleo das *threads* do OpenMP. Os resultados alcançados representam uma melhora no tempo de execução da aplicação em OpenMP, que varia de 53% nos casos de 16 e 32 *threads* a 56,4% no caso de 48 *threads*. Essa comparação é em relação aos casos onde não se utilizou o *thread pinning*.

Outra variável que influencia no tempo da implementação em OpenMP são as barreiras necessárias para a sincronização das *threads*. As barreiras são chamadas através da diretiva `#pragma omp barrier` e se localizam após declarações de variáveis, durante o manuseio da estrutura de dados compartilhada na transposição e antes da liberação de memória. A visualização do tempo ocupado pelas barreiras, na ferramenta Cube,

possibilita afirmar que nas execuções com *thread pinning* o tempo ocupado pelas barreiras corresponde ao *overhead* em relação as execuções da implementação em MPI. Sem o *thread pinning*, os tempos de execução em OpenMP são mais elevados e o *overhead* também é causado pelo número de *cpu-migrations*, como visto anteriormente.

6. Considerações Finais

Com base nos resultados obtidos é possível observar a influência de *cpu-migrations* no baixo desempenho inicial da implementação em OpenMP. A técnica de *thread pinning* usada com a ferramenta LIKWID foi efetiva ao diminuir esse número de *cpu-migrations* e consequentemente os tempos de execução.

Além disso, as barreiras utilizadas para a sincronização se tornam um problema necessário na implementação em OpenMP, principalmente no que se refere ao acesso a estrutura de dados compartilhada. O acesso a essa estrutura necessita de sincronização para que haja apenas a disponibilidade de dados corretos para a utilização.

Os resultados obtidos nessa pesquisa possibilitam que em trabalhos futuros investiguem-se outras maneiras de melhorar o desempenho da aplicação em OpenMP, seja através da diminuição de *context-switches*, que possivelmente influencia no desempenho, quanto em uma maneira de implementar barreiras mais seletivas dentro do código, onde as *threads* não necessitem aguardar todas as outras para seguir a execução, mas aguardem somente as *threads* com as quais compartilharão dados. Também espera-se retornar ao Incompact3d para realizar melhorias na questão da comunicação.

7. Agradecimentos

Este trabalho foi parcialmente financiado pelo projeto “GREEN-CLOUD: Computação em Cloud com Computação Sustentável” (#162551-0000 488-9), no programa FAPERGS-CNPq PRONEX 12/2014.

Referências

Bird, A., Long, D., e Dobson, G. (2017). Implementing shared memory parallelism in MCBEND. In *EPJ Web of Conferences*, volume 153, page 07042. EDP Sciences.

de Araujo, L. R., Weber, C. M., Puntel, F. E., Charão, A. S., e Lima, J. V. F. (2018). Análise de desempenho do software Incompact3D em uma arquitetura com múltiplos núcleos. *Fórum de Iniciação Científica da Escola Regional de Alto Desempenho do Rio Grande do Sul (ERAD RS)*, (2/2018).

Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., e Chapman, B. (2011). High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*, 37(9):562–575.

Krawezik, G. e Cappello, F. (2003). Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127. ACM.

Xu, Y. e Zhang, T. (2015). A hybrid Open MP/MPI parallel computing model design on the SMP cluster. In *Power Electronics Systems and Applications (PESA), 2015 6th International Conference on*, pages 1–5. IEEE.

Melhorando o Desempenho de uma Aplicação de Simulação Numérica Direta de Uma Camada De Mistura Utilizando *Loop Interchange* e *OpenMP*

Sherlon A. da Silva[1], **Matheus C. N. da Silva** [2], **Claudio Schepke**[1], **Cesar Cristaldo** [2]

[1]Laboratório de Estudos Avançados Universidade Federal do Pampa (UNIPAMPA)
97546-550, Alegrete – RS – Brasil

sherlonalmeida@alunos.unipampa.edu.br, claudioschepke@unipampa.edu.br

[2]Grupo de Fenômenos de Transporte Avançado
Universidade Federal do Pampa (UNIPAMPA)
97546-550, Alegrete – RS – Brasil

mathe1s.castro@gmail.com,cesarcristaldo@unipampa.edu.br

Abstract. *The study of hydrodynamic stability finds application in many branches of engineering as dispersion of pollutants, combustion systems, among others. Computational Fluid Dynamics (CFD) makes possible to simulate phenomena that are not always reproducible in the laboratory. However, numerical solutions require a large amount of computations on data, making the execution time long enough to make unfeasible such study. In order to mitigate the execution time of a simulation application of a mixture layer, this work evaluates optimization and parallelization alternatives such as loop interchange and OpenMP. The results show an improvement of up to 2.5× in execution time.*

Resumo. *O estudo da estabilidade hidrodinâmica encontra aplicação em muitos ramos da engenharia como dispersão de poluentes, sistemas de combustão, entre outros. A Dinâmica de Fluidos Computacional (DFC) possibilita simular fenômenos que nem sempre são passíveis de reproduzir em laboratório. Porém, as soluções numéricas requerem grande quantidade de cálculos sobre dados, tornando o tempo de execução longo o suficiente para inviabilizar tal estudo. A fim de mitigar o tempo de execução de uma aplicação de simulação de uma camada de mistura, este trabalho avalia alternativas de otimização e paralelização como loop interchange e OpenMP. Os resultados obtidos mostram melhora de até 2.5× no tempo de execução.*

1. Introdução

A simulação computacional de fenômenos naturais é amplamente difundida [Jamshed 2015]. A representação discreta de um fenômeno possibilita a sua simulação utilizando métodos numéricos adequados. Desta forma, pode-se compreender o fenômeno de maneira acurada, identificando padrões no comportamento para controlá-lo e/ou predizer sua ocorrência.

Um problema recorrente de simulações computacionais é a grande quantidade de cálculos envolvendo os dados. Isso causa uma demora na obtenção dos resultados

desejados. Em contrapartida, existem máquinas com diversos núcleos de processamento, chamadas de arquiteturas *multi-core* e *many-core*, que são capazes de realizar cálculos de maneira paralela e agilizar a execução.

Neste trabalho, deseja-se mitigar o tempo de execução de uma aplicação de simulação de escoamento de fluidos em uma camada de mistura. Para tanto, o código é modificado com alternativas de otimização como *loop interchange* [Kowarschik and Weiß 2003] e paralelização com OpenMP para arquiteturas *multi-core* [Chapman et al. 2008, Chandra et al. 2001].

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta a aplicação de simulação computacional utilizada. Na Seção 3 são apresentados detalhes de implementação, otimização e paralelização. Os resultados experimentais são mostrados na Seção 4. Finalmente, a Seção 5 traz a conclusão e trabalhos futuros.

2. Simulação Numérica Direta de uma Camada de Mistura

Neste trabalho, está sendo utilizada uma aplicação desenvolvida por [Manco 2014]. O intuito da aplicação é obter, via Simulação Numérica Direta (DNS) [Quirino and Mendonça 2007, Moin and Mahesh 1998], uma condição que seja favorável para a formação de vórtices de Kelvin-Helmholtz, ou seja, uma condição que amplifique as perturbações impostas no escoamento em uma camada de mistura, onde é desejável que o regime seja turbulento.

Pensando numa interface oxidante-combustível, a turbulência aumenta a área de contato entre os dois fluidos. A difusão das espécies químicas é acelerada e o processo de combustão ocorre em tempo oportuno. Um exemplo de aplicação é utilizado no sistema de propulsão Scramjet da NASA, onde um veículo não tripulado atingiu uma velocidade de aproximadamente 12.144 km/h por 300 segundos.

Na Figura 1 são mostrados os vórtices gerados em uma simulação. O eixo y representa a vorticidade adimensional e o eixo x representa o domínio físico da câmara de mistura. Vale ressaltar que cada imagem gerada na simulação é uma representação adimensional da vorticidade no instante de tempo em que se encontra.

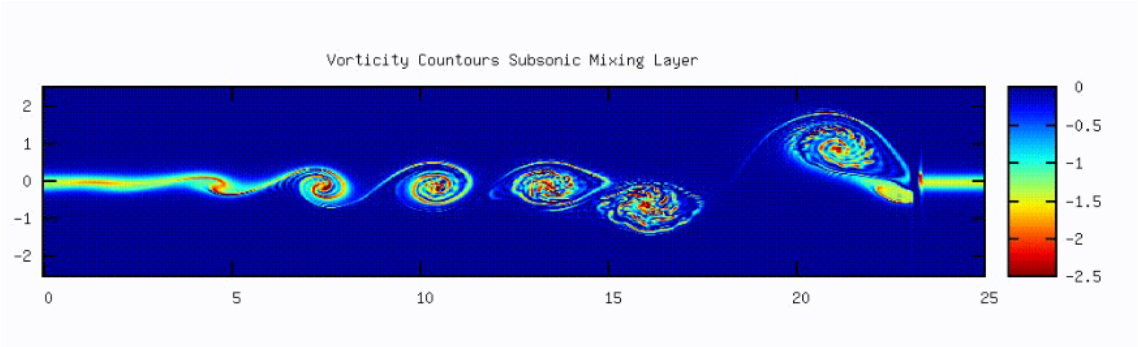


Figura 1. Simulação numérica direta - vorticidade.

O tempo sequencial de execução da aplicação para 100 imagens demora cerca de 1 hora e 20 minutos, aumentando proporcionalmente ao número de imagens. Este tempo é elevado, o que onera a execução com diferentes parâmetros de entrada ou a inserção de novas equações.

3. Otimização e Paralelização

Nesta seção são apresentadas as alternativas adotadas para a redução do tempo de execução da aplicação de simulação de camada de mistura. Em trabalhos prévios, a técnica de *loop interchange* [da Silva 2018] apresentou melhorias de desempenho em operações matriciais. Já OpenMP é uma alternativa de paralelização que pode ser utilizado com poucas modificações de código.

3.1. Loop Interchange

A técnica de *loop interchange* trabalha nos laços de repetição das aplicações visando reordená-los para melhor aproveitar os dados em *cache* [Kowarschik and Weiß 2003]. Assim, percorre-se os elementos na ordem em que estes estão armazenados na memória.

Ao alterar a ordem dos laços de repetição, o programador faz com que os índices controlados por este laço sejam percorridos de forma diferente do código original. O exemplo de algoritmo sem *loop interchange* é apresentado no Algoritmo 1 à esquerda, e com *loop interchange* à direita. Com isso, a Figura 2 apresenta o fluxo de execução sequencial e o impacto de *loop interchange* para um exemplo de matriz U de ordem U_{4x4}.

```
1 do i=0,N_Linhas-1                do j=0,N_Colunas-1
2    do j=0,N_Colunas-1               do i=0,N_Linhas-1
3       ...                              ...
4    end do                           end do
5 end do                           end do
```

Algoritmo 1: Versão Original e Versão *Loop Interchange*

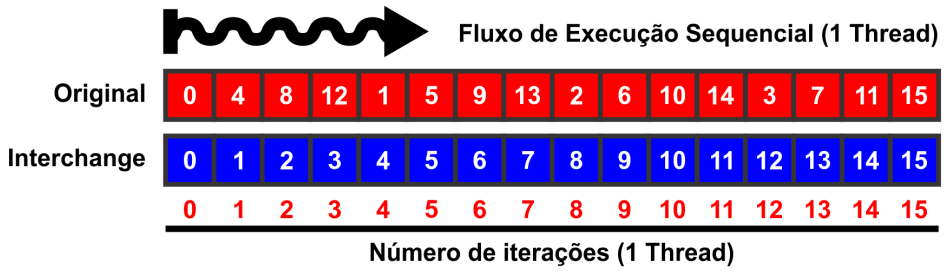


Figura 2. Representação de uma execução sequencial.

Como se pode ver no Algoritmo 1, a técnica inverte a ordem dos laços de repetição das linhas 1 e 2. Desta forma, a iteração dos elementos da matriz U ocorre na ordem em que estas estão armazenadas em memória. Isto acarreta no melhor aproveitamento dos dados quando carregados em *cache*. Em Fortran, isso significa que o ideal é percorrer primeiramente as linhas dadas pela variável **i** e posteriormente percorrer as colunas dadas por **j**. Para tanto, foram modificados no código original 15 laços de repetição aninhados utilizando *loop interchange*.

3.2. OpenMP

OpenMP é uma interface de programação paralela de memória compartilhada que possibilita transformar trechos de código sequenciais em paralelos [Chapman et al. 2008, Chandra et al. 2001]. OpenMP pode ser incorporado em linguagens como C, C++ e Fortran, viabilizando a divisão da execução em fluxos paralelos (*threads*).

A utilização de OpenMP provê redução do tempo de execução da simulação uma vez que trechos de execução concorrentes são criados. Com a execução paralela, por exemplo, é possível distribuir as iterações do fluxo de execução que executarão ao mesmo tempo em *cores* distintos, uma vez que os cálculos sobre os dados são independentes. A Figura 3 ilustra a execução paralela usando 4 *threads*.

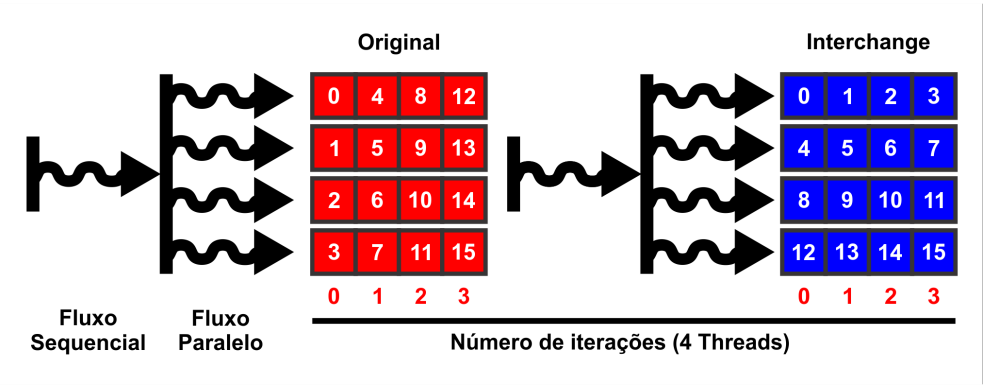


Figura 3. Representação de uma execução paralela com 4 *threads*.

A utilização de OpenMP também facilita a construção de trechos de execução paralela em poucas linhas de código. Neste trabalho foram paralelizados 25 laços de repetição utilizando *PARALLEL DO*, como mostra o Algoritmo 2, onde na linha 1 é definido o número de *threads*, na linha 2 é construída a região paralela para laços de repetição definindo as respectivas variáveis privadas, e por fim na linha 8 a região paralela é finalizada e as *threads* são sincronizadas.

```
1 call OMP_SET_NUM_THREADS(n_threads)
2 !SOMP PARALLEL DO PRIVATE(...)
3 do i=0,N_Linhas-1
4    do j=0,N_Colunas-1
5       ...
6    end do
7 end do
8 !SOMP END PARALLEL DO
```

Algoritmo 2: Exemplo de Paralelização - Parallel Do

4. Resultados Experimentais

Para a coleta dos resultados foi utilizada uma *workstation* do Laboratório de Estudos Avançados em Computação (LEA) da UNIPAMPA, com dois processadores Intel Xeon E5-2650. Cada processador conta com 8 *cores* físicos, permitindo, ao todo, a

execução de 32 *threads* com *Hyper-Threading*. Cada núcleo possui uma memória *cache* L1 privada de 32 KB, L2 privada de 256 KB, e L3 com 20 MB compartilhados.

Vale ressaltar que além das otimizações realizadas no trabalho, foram aplicadas otimizações automáticas oferecidas por diretivas de compilação. Para tanto, foi utilizada como padrão para todos os testes a diretiva -O3 do compilador GFortran.

Para obtenção dos resultados, o algoritmo foi executado de forma sequencial e paralela 100 vezes, utilizando 2, 4, 8 e 16 *threads*, com o intuito de mensurar a escalabilidade da paralelização dentro do número de núcleos de processamento disponíveis no ambiente utilizado. Foram comparadas duas versões do algoritmo, sendo a primeira a original obtida do Grupo de Fenômenos de Transporte Avançado da UNIPAMPA, e a segunda com trechos de otimização em laços de repetição com a técnica de *loop interchange*.

A Figura 4 apresenta os tempos de execução em segundos obtidos para a execução da aplicação sequencial, paralela e usando *loop interchange*. Além disso, a Tabela 1 apresenta o ganho de desempenho da versão paralela sobre a sequencial (*speedup*) e o da versão *Interchange* sobre a original (Ganho). É importante destacar que o número de elementos discretos considerados foram definidos por uma matriz U de ordem $U_{421x381}$.

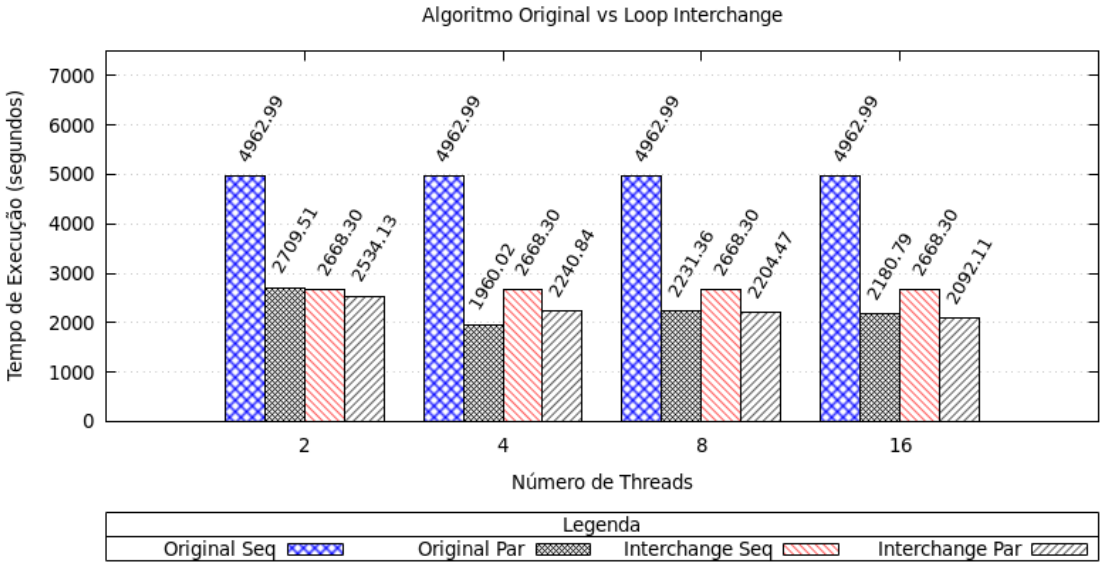


Figura 4. Tempo de Execução - Sequencial X Paralelo

Na Figura 4 é possível observar ganhos de desempenho tanto na versão paralela quanto na versão com *loop interchange*. No melhor dos casos o tempo de execução é reduzido em 2.5× em relação ao tempo sequencial original, o que diminuiu o tempo total de simulação de aproximadamente 1 hora e 22 minutos, para cerca de 32 minutos. No entanto, com o aumento do número de *threads* não houve um ganho proporcional. Isto é devido à escalabilidade estar associada à ordem da matriz, uma vez que ao aumentar o número de *threads* a carga de trabalho para cada *thread* se manteve baixa.

5. Conclusão e Trabalhos Futuros

Neste artigo foram adotadas técnicas de otimização e paralelização para reduzir o tempo de execução de uma aplicação de simulação de uma camada de mistura na área

Tabela 1. Tempo de execução (segundos), *Speedup* e Ganho

Versão	**Execução (2 *Threads*)**			**Execução (4 *Threads*)**		
	Sequencial	Paralelo	*Speedup*	Sequencial	Paralelo	*Speedup*
Original	4962.99	2709.51	1.83	4962.99	1960.02	2.53
Interchange	2668.30	2534.13	1.05	2668.30	2240.84	1.19
Ganho	1.85	1.06	-	1.85	0.87	-
Versão	**Execução (8 *Threads*)**			**Execução (16 *Threads*)**		
	Sequencial	Paralelo	*Speedup*	Sequencial	Paralelo	*Speedup*
Original	4962.99	2231.36	2.22	4962.99	2180.79	2.27
Interchange	2668.30	2204.47	1.21	2668.30	2092.11	1.27
Ganho	1.85	1.01	-	1.85	1.04	-

de dinâmica dos fluidos. Através das soluções implementadas foi possível aumentar a eficiência em 2.5×. Como trabalhos futuros outras técnicas de otimização serão analisadas na aplicação atual, como *loop unrolling* e *loop tiling* para memória *cache*. Também serão avaliadas outras arquiteturas, como co-processadores Xeon Phi e GPU's.

Agradecimentos

Este trabalho foi realizado com recursos do PROBIC/FAPERGS 2017 e pela agência de fomento CNPq via Edital Universal Processo N. 457684/2014-3.

Referências

Chandra, R., Dagum, L., Kohr, D., Maydan, D., Menon, R., and McDonald, J. (2001). *Parallel programming in OpenMP*. Morgan kaufmann.

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

da Silva, Sherlon Almeida Schepke, C. (2018). Avaliando o Desempenho de Produtos Matriciais em Arquiteturas Multi-Core e Many-Core com OpenMP e CUDA. *Anais do Salão Internacional de Ensino, Pesquisa e Extensão*, 9(3).

Jamshed, S. (2015). *Using HPC for Computational Fluid Dynamics: A Guide to High Performance Computing for CFD Engineers*. Academic Press.

Kowarschik, M. and Weiß, C. (2003). An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, pages 213–232. Springer.

Manco, J. A. A. (2014). Condições de contorno não reflexivas para simulação numérica de alta ordem de instabilidade de kelvin-helmholtz em escoamento compressível. Master's thesis, Instituto Nacional de Pesquisas Espaciais (INPE), São José dos Campos.

Moin, P. and Mahesh, K. (1998). Direct numerical simulation: a tool in turbulence research. *Annual review of fluid mechanics*, 30(1):539–578.

Quirino, S. F. and Mendonça, M. T. (2007). Direct numerical simulation of compressible shear layer with heat source. In *19th International Congress of Mechanical Engineering*, Brasília.

Paralelização em um Ambiente de Memória Distribuída de um Simulador da Formação de Edemas no Coração

Lara Turetta Pompei[1], **Ruy Freitas Reis**[1] , **Marcelo Lobosco**[1]

[1]FISIOCOMP
Laboratório de Fisiologia Computacional e Computação de Alto Desempenho,
Universidade Federal de Juiz de Fora (UFJF)

pompei.lara@engenharia.ufjf.br,{ruyfreitas,marcelo.lobosco}@ice.ufjf.br

Abstract. *The purposes of this work are the paralelization of an algorithm that simulates an edema formation in the heart, as well as to analyse its performance. The results show that the parallel version led to a reduction in the execution time up to 14 times compared to its sequential version.*

Resumo. *O objetivo deste trabalho é paralelizar um algoritmo que simula a formação de um edema no coração, usando para isso MPI, bem como analisar o seu desempenho em um cluster de computadores. Os resultados mostram que a versão paralela conseguiu reduzir o tempo de execução sequencial em até 14 vezes.*

1. Introdução

Segundo a OMS, cerca de 17,5 milhões de pessoas morrem vítimas de doenças cardiovasculares a cada ano no mundo, o que corresponde a aproximadamente 0,55 mortes por segundo, número duas vezes maior que as mortes decorrentes de câncer no mesmo período [OMS 2014]. Apesar da grande quantidade de estudos sobre doenças cardiovasculares, alguns de seus aspectos permanecem desconhecidos. Uma ferramenta que pode ajudar os cientistas a melhor compreender as causas das doenças e possíveis estratégias para seus tratamentos é a modelagem computacional. Os modelos matemático-computacionais podem ser criados para representar aspectos chave do funcionamento do coração, do surgimento de uma determinada doença ou mesmo da ação de uma droga para o seu tratamento. A partir destes modelos, simulações podem ser feitas para melhor compreender os fenômenos modelados.

No entanto, simulações podem demandar um grande poder computacional para sua execução, levando horas, dias ou mesmo meses para gerar resultados, o que pode inviabilizar seu uso. Neste cenário, o desenvolvimento de aplicações paralelas é essencial para atender a demanda computacional destas aplicações, reduzindo seu tempo de execução e tornando seu uso viável. Tais aplicações paralelas fazem uso simultâneo de várias unidades de processamento para reduzir seu tempo de computação. Uma das estratégias empregadas para o desenvolvimento de uma aplicação paralela é o chamado paralelismo de dados, em que um mesmo conjunto de instruções deve ser executado para um grande conjunto de dados [Pacheco 2011]. Os dados podem então ser divididos em conjuntos menores, que por sua vez são operados simultaneamente por distintas unidades de processamento.

Neste trabalho o paralelismo de dados é empregado para reduzir o tempo de execução de um simulador da formação de edemas em um tecido cardíaco [Reis 2018]. A versão paralela do simulador é implementada com o uso da biblioteca MPI [Pacheco 1996], sendo executada em um ambiente de memória distribuída. Os tempos de execução são então coletados e analisados.

Este artigo está organizado da seguinte forma. A Seção 2 descreve a versão serial do simulador; o método aplicado para a paralelização do simulador é apresentado na Seção 3. Os resultados são apresentados e discutidos na Seção 4. Por fim, a última seção apresenta as conclusões e perspectivas de trabalhos futuros.

2. Simulador da Formação de Edemas no Coração

Quando o corpo sofre uma infecção por um patógeno, o sistema imune desenvolve uma resposta inflamatória. Um dos mecanismos usados pelo sistema imune para controlar uma infecção é recrutar células de defesa da corrente sanguínea para o local da infecção. Isto é feito através da vasodilatação, que aumenta a permeabilidade dos capilares sanguíneos para facilitar a entrada dos neutrófilos, células de defesa, no tecido [Sompayrac 2008]. Contudo, além dos neutrófilos, também entram no tecido fluidos sanguíneos. Um edema ocorre quando um volume excessivo de fluido acumula no tecido, levando ao inchaço na região [Reis 2018, Reis *et al.* 2016b]. Edemas podem ocorrer em vários tipos de tecidos, como o cardíaco. A formação de um edema no coração pode levar à morte.

Para simular a formação de edemas no tecido cardíaco utiliza-se um conjunto de 3 Equações Diferenciais Parciais (EDPs) que descrevem tando a dinâmica de uma agente patogênico (bactéria) invasor, quanto do sistema de defesa do organismo (neutrófilos) [Reis 2018, Reis *et al.* 2016a]. O aumento de permeabilidade dos capilares leva ao aumento do fluxo para o tecido, o que aumenta o volume e consequentemente a pressão do fluido dentro do tecido. Parte deste fluido é coletado por vasos linfáticos. Assim, ocorre o aumento do fluxo linfático, que por sua vez leva a um $feedback$ negativo do fluxo para o tecido, em uma tentativa de reequilibrar o organismo [Scallan *et al.* 2010].

Os pontos discretizados do tecido são representados computacionalmente por uma malha bidimensional de valores em ponto-flutuante de precisão dupla. O método numérico usado para a implementação computacional do modelo matemático é o Método dos Volumes Finitos. O método de Jacobi foi utilizado para resolver os sistemas lineares resultantes, e o método de Euler explícito é utilizado para a discretização do tempo. A computação do termo convectivo é uma parte complexa na resolução das EDPs. A implementação utiliza o esquema conhecido como *First-Order Upwind* para garantir a solução estável do termo convectivo [Reis 2018].

3. Método

A paralelização do código do simulador foi feita para uma arquitetura de memória distribuída, sendo por esse motivo escolhida a biblioteca MPI para sua paralelização. Após identificada a parte do código que possui o maior tempo de execução, responsável pelo cálculo da dinâmica dos neutrófilos e das bactérias, verificou-se a viabilidade de empregar paralelismo de dados para reduzir o seu tempo de computação, uma vez que, a cada passo de tempo, uma mesma sequência de operações é calculada para cada ponto do domínio.

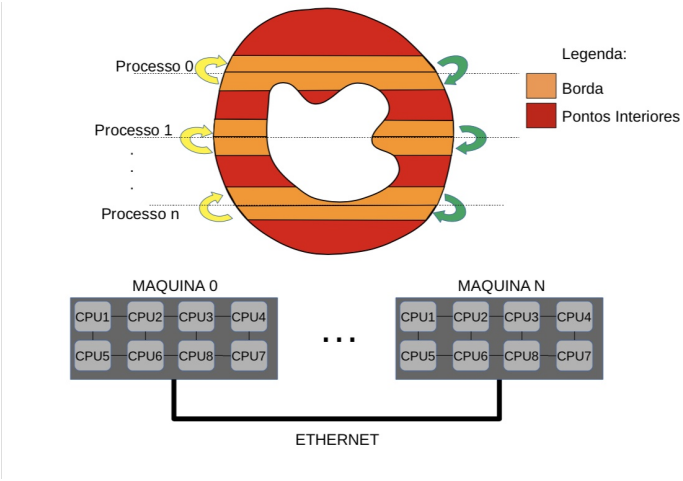


Figura 1. Ilustração da divisão de trabalho entre os processos

Dessa forma, deve-se dividir a malha que representa todo o tecido entre todos os processadores disponíveis, conforme ilustrado na Fig. 1. Por ser uma aplicação com características *CPU-bound*, foi criado um processo MPI para cada processador disponível. Ao se dividir a malha entre os processadores introduz-se, contudo, um problema que não existia na versão sequencial: para calcular as novas populações de neutrófilos e bactérias para um ponto do domínio em um dado intervalo de tempo t, faz-se necessário o acesso aos valores das populações contidas em seus pontos vizinhos no intervalo $t - 1$. Com a divisão dos dados entre os processadores, os pontos vizinhos podem agora estar localizados em outras máquinas. O conjunto de pontos vizinhos, chamados de bordas, devem ser trocados entre máquinas a cada passo de tempo para que a computação possa ser realizada corretamente. Introduz-se, assim, um custo de comunicação a cada passo de tempo, conforme ilustrado pelo Algoritmo 1 e indicado pela Fig. 1 através das setas verdes e amarelas.

Algoritmo 1: Pseudocódigo da versão paralela do código.

```
1  MPI_Init (NULL, NULL);
2  MPI_Comm_rank (MPI_COMM_WORLD, &r);
3  MPI_Comm_size(MPI_COMM_WORLD,&n);
4  ...
5  k = nx/n; // k = total de linhas a ser computada por cada processo
6  for t = 0; t < nt; t++ do
7      for x = k*r; x ≤ (k*r)+(k-1); x++ do
8          for y = 0; y < ny; y++ do
9              Computa bactérias e neutrófilos para o ponto x,y
10         end for
11     end for
12     MPI_Sendrecv para trocar minhas bordas com meus vizinhos
13 end for
```

Pode-se fazer a divisão da malha entre os processos de três formas distintas: a) dividir as colunas entre os processos, b) dividir as linhas entre os processos ou c) dividir a matriz em submatrizes a serem computadas simultaneamente. Inicialmente optou-se por

dividir a malha em linhas (Fig. 1) a fim de se reduzir o número de mensagem que precisam ser trocada entre os processos, se comparado com a divisão em submatrizes. Quando a divisão por linhas é comparada com a divisão por colunas, o número de mensagens trocadas é igual pelo fato da matriz utilizada nos testes ser quadrada. Porém pode existir uma pequena diferença no tempo de comunicação devido a forma como os dados são armazenados na memória. Tendo em vista que o código sequencial foi originalmente implementado em C, e que nesta linguagem matrizes são armazenadas na memória por linhas, os acessos aos dados para montagem de um pacote por parte do MPI tendem a ser mais rápidos pelo fato dos dados estarem localizados em endereços contíguos de memória.

Os experimentos foram executados em *cluster* composto por oito computadores, cujo acesso exclusivo é feito através de uma fila de submissão de *jobs* (OGE). Cada computador possui dois processadores Intel Xeon E5620 de 2.40 GHz, cada um com quatro núcleos, totalizando assim 8 núcleos de processamento por computador. Cada núcleo possui 32KB de *cache* L1 de dados, 32 KB de *cache* L1 de instruções, e 256KB de *cache* L2, esta compartilhada por instruções e dados. Cada processador possui ainda uma *cache* L3 de 12MB, também usada para armazenar dados e instruções, e compartilhada entre os quatro núcleos do processador. Apesar deste processador possuir suporte para a tecnologia *hyper-threading*, esta foi desabilitada na BIOS. Os computadores executam o SO Linux com *kernel* na versão $3.10.0$. O compilador gcc na sua versão $4.8.5$ foi usado para compilar os programas e a biblioteca *mpich* versão 3.2 foi usada para estabelecer a comunicação entre os processos. O tempo foi computado através do aplicativo *time*, disponível no SO. Para fins de cálculo do tempo de execução, foi considerada a média aritmética simples de 8 execuções, sendo o maior desvio-padrão observado igual a $0,8\%$. A simulação foi feita para representar um tecido de dimensões 1cm $\times$ 1cm, sendo discretizado com 800×800 pontos. São usados 100.000 passos de tempo na simulação. O tempo de execução sequencial da aplicação é de 5.201s (cerca de 1h27), o que torna proibitiva a simulação de um coração completo em tempo real.

4. Resultados

Nesta seção discute-se os resultados obtidos para execução paralela usando diferentes quantidades de computadores e de núcleos.

A Figura 2 apresenta o *speedup* da aplicação quando executada com 2, 4, 8, 16, 32 e 64 núcleos. Considera-se sempre a execução em uma configuração com o menor número de máquinas possível. Como estão disponíveis apenas 8 núcleos por máquina, as configuração com 2, 4 e 8 núcleos foram executadas em uma mesma máquina, 16 núcleos foram executadas com 2 máquinas, 32 núcleos com 4 máquinas e 64 núcleos com 8 máquinas. Pode-se observar que a eficiência foi baixa: iniciou-se com 92% para dois núcleos, e chegou a apenas 17% para 64 núcleos.

O melhor tempo de execução paralelo foi obtido em uma configuração que utiliza 43 núcleos: 366s, o que representa um *speedup* de 14 vezes, ou seja, uma melhoria de 93% do tempo de execução sequencial.

Um experimento final foi realizado: forçou-se a alocação dos processos MPI em um número maior de máquinas, através de comandos específicos para a fila de execução, para que se pudesse investigar se haveria impacto no tempo total de execução. A princípio

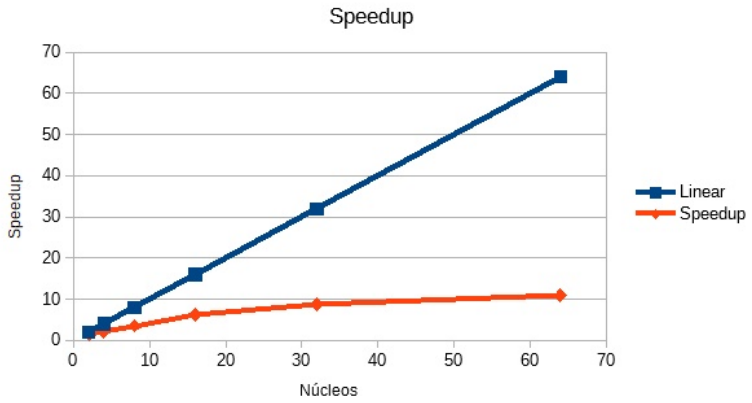


Figura 2. *Speedup*, considerando sempre a execução em uma configuração com o menor número de máquinas.

Tabela 1. Tempo de execução (em s) para 2, 4, 8, 16, 32 e 64 processos em 1, 2, 4, 8 e 16 máquinas. Os menores tempos para cada configuração estão destacados em negrito.

Número de Máquinas	2	4	8	16	32	64
1	**2.828**	2.434	1.493			
2	3.859	2.432	**1.412**	835		
4		**2.431**	1.416	809	593	
8			1.573	**801**	**509**	**473**
16				931	555	561

se esperaria um aumento no tempo de execução em virtude de um possível aumento no tempo de comunicação. Isso ocorreria em virtude da comunicação se dar por padrão, quando os processos estão localizados na mesma máquina, por memória compartilhada (usa-se o padrão *ch3:nemesis*). Por exemplo, no caso da Fig. 1, os processos localizados em uma mesma máquina se comunicariam por memória compartilhada. Contudo, de modo surpreendente, os resultados mostram que o emprego de um número maior de máquinas pode, eventualmente, reduzir o tempo de execução, ao invés de aumentá-lo. Essa redução do tempo pode ocorrer devido à disponibilidade de maior quantidade de memória cache em cada uma das configurações. No entanto seria necessário o uso de ferramentas de *profile* para verificarmos a veracidade dessa hipótese. A Tabela 1 apresenta os tempos de execução de 2, 4, 8, 16, 32 e 64 processos em distintas configurações.

5. Conclusões e trabalhos futuros

Este trabalho apresentou a paralelização, usando MPI, de um simulador da formação de edemas em um tecido cardíaco. Na sequência, a versão paralela do código foi executada em um *cluster* de 8 máquinas, sendo cada máquina composta por 8 núcleos, totalizando assim 64 núcleos. A versão paralela obteve ganhos de desempenho de até 14 vezes em relação a sua versão sequencial, reduzindo o tempo de computação de cerca de 5.201s para 366s. Também foram feitos experimentos forçando que os processos MPI fossem criados em diferentes máquinas, de modo que se pudesse avaliar seu impacto no tempo de

comunicação. Surpreendentemente, apesar do suporte para memória compartilhada oferecido pelo *mpich*, que deveria reduzir os tempos de comunicação quando uma mensagem fosse enviada para um destinatário localizado na mesma máquina, os experimentos mostraram que para certas configurações reduz-se o tempo de execução ao alocar os processos MPI em um número maior de máquinas. Suas causas, no entanto, devem ser melhor investigadas em trabalhos futuros.

Adicionalmente, como trabalhos futuros, pretende-se melhorar a implementação atual, dividindo a computação de cada núcleo em dois pedaços: bordas e pontos interiores. A ideia é computar inicialmente as bordas e, enquanto os pontos interiores são computados, realizar em paralelo a troca das bordas. Também deseja-se a) utilizar aceleradores, como GPUs, na computação, b) investigar se de fato a divisão das malhas atual é a que leva ao melhor desempenho, c) utilizar diferentes tamanhos de malhas para estudar a escalabilidade do programa, e d) coletar eventos do processador para averiguar os impactos da *cache* nas variações de tempo de execução de cada configuração.

Observações Finais e Agradecimentos

Lara Turetta Pompei é bolsista PIBIC-CNPq, sendo responsável por implementar a versão paralela do código, realizar os experimentos e escrever o texto. O prof. Ruy Freitas Reis desenvolveu o modelo matemático e a versão sequencial do código. O prof. Marcelo Lobosco orientou a aluna de IC e revisou a versão final do texto.

Os autores agradecem o apoio do CNPq, FAPEMIG e UFJF para a elaboração desse trabalho.

Referências

OMS (2014). Organização Mundial da Saúde. Publicação Eletroñica - http://www.who.int/. Uĺtimo acesso em 21 de setembro de 2016.

Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Pacheco, P. S. (1996). *Parallel Programing with MPI*. Morgan Kaufmann Publishers.

Reis, R. F. (2018). *Modelagem Mecânica da Formação de Edemas*. PhD thesis, Universidade Federal de Juiz de Fora.

Reis, R. F., dos Santos, R. W., de Oliveira Campos, J., and Lobosco, M. (2016a). Interstitial pressure dynamics due to bacterial infection. *Mecánica Computacional. Bioengineering And Biomechanics (B)*, 34(1):1181–1194.

Reis, R. F., dos Santos, R. W., and Lobosco, M. (2016b). A plasma flow model in the interstitial tissue due to bacterial infection. *Lecture Notes in Computer Science*, pages 335–345.

Scallan, J., Huxley, V. H., and Korthuis, R. J. (2010). *Capillary fluid exchange: regulation, functions, and pathology*, volume 2. Morgan & Claypool Life Sciences.

Sompayrac, L. M. (2008). *How the Immune System Works*. Wiley, John & Sons, Incorporated.

Aplicação de Autômatos Celulares Paralelos para Simulação do Impacto Causado pela Elevação do Nível do Mar

Edenilton Michael de Jesus [1], André L. Silva Santos[1], Omar A. Carmona Cortes[1], Hélder Pereira Borges[1]

[1]Instituto Federal do Maranhão (IFMA)
Campus Monte Castelo – São Luís – MA – Brasil

edeniltonmichael.dj@gmail.com, {andresantos,omar,helder}@ifma.edu.br

Abstract. *Geographical simulations are computational intense because they have to deal with a considerable amount o data. The more the required precision, the bigger the necessary processing. In this context, parallel computing can be used as an allied to process such amount of data. For instance, an interesting application in this scenario is to discover environmental problems in large-scale geographic areas. Thus, this work proposes the using of parallel cellular automata to detect the impact caused by the tide rising. This work was implemented in C++ using MPI. Our simulations showed that it is possible to achieve a superlinear speedup using 1,716,064 cells with two process.*

Resumo. *A simulação de processos geográficos demandam a utilização de uma grande quantidade de dados, pois quanto maior a precisão necessária em sua representação, mais dados devem ser processados. Nesse contexto, a programação paralela surge como um aliado para diminuir o tempo de processamento necessário em simulações que podem, por exemplo, encontrar problemas ambientais em grandes áreas geográficas. Dessa forma, este artigo propõe a utilização de autômatos celulares paralelos para determinar o impacto causado pela elevação no nível do mar. O trabalho foi implementado utilizando-se C++ e MPI, sendo que a simulação pode alcançar um speedup superlinear a partir da utilização de 1,716,064 células com dois processos.*

1. Introdução

A representação computacional de dados geográficos ganhou gradativa notoriedade em razão de paradigmas propostos, como por exemplo, o de quatro universos. De acordo com [Casanova et al. 2005], este paradigma distingue quatro passos entre o mundo real e sua realização computacional: universo ontológico, formal, estrutural e de implementação. Considerando que os dois primeiros universos consistem no mapeamento do espaço geográfico real para modelos de dados abstratos, este trabalho é centrado nos universos estrutural e de implementação, que permite o mensuramento das mudanças ocorridas no espaço geográfico ambiental como, por exemplo, as mudanças do fenômeno de elevação média global do mar [Bezerra 2014]. Como consequência dessas altas, o mar avança gradativamente e pode provocar mudanças nas características das áreas costeiras ou mesmo as inundar. Dessa forma, um grande desafio é a representatividade computacional dessas mudanças e estratégias de previsão que possam ajudar a mapeá-las com a maior precisão possível, determinando assim os impactos ou prever o que pode acontecer ao ecossistema costeiro. Neste trabalho, a estratégia utilizada é baseada em autômato celulares

(AC) [Batty and Xie 1994, Liu et al. 2014], que são definidos por seus espaços celulares e por suas regras de transição [Martins et al. 2011].

Para representação computacional, toma-se como base a estrutura de mapas bidimensionais matriciais, que consiste no uso de uma malha quadriculada regular sobre a qual se constrói, célula a célula, o elemento que está sendo representado [Câmara et al. 2001]. Nesta representação, o espaço é uma matriz *P(m, n)* composto de *m* colunas e *n* linhas, onde cada célula possui um número de linha, um número de coluna e um valor correspondente ao atributo estudado, sendo cada célula individualmente acessada pelas suas coordenadas [Casanova et al. 2005].

Em muitas ocasiões, dependendo da área de mapa estudada e da quantidade de dados a se processar, há algumas limitações para utilização de escalas mínimas para representação computacional por conta do grande volume de dados gerados. Isso influencia negativamente na precisão de uma simulação pois quanto maior a escala, maior a área real que uma única célula representará, podendo assim abranger uma área real altamente heterogênea. Portanto, para se obter melhores precisões nos resultados, uma escala menor deve ser utilizada gerando assim um maior volume de dados para se processar exigindo mais processamento computacional. Assim, para a simulação de muitos aspectos dessa natureza é interessante o uso programação paralela, pois arquiteturas paralelas fornecem uma excelente oportunidade para aplicativos com grandes requisitos computacionais [Matloff 2011].

Nesse contexto, a abordagem deste trabalho é a utilização do *Message Passing Inteface* (MPI) para a simulação do impacto causado pela elevação do nível do mar às zonas costeiras do Maranhão, utilizando-se da metodologia de autômatos celulares, permitindo assim simulações experimentais de regiões com o máximo de dados possíveis.

2. Arquitetura de Simulação Paralela

A arquitetura da simulação é formada basicamente por quatro módulos. O módulo *Gerenciador de Dados* é responsável pela estruturação dos dados e interface de gerenciamento dos dados pelo autômato. O módulo *Autômato* divide o autômato celular em *P* processos numerados de 0 a P-1. O módulo *Regras* aplica as regras de simulação em cada P processo. Finalmente, o módulo *Saída* fornece as saídas do sistema que são os valores de inundação para cada tipo de solo inicial.

Em termos estruturais, para a simulação desse estudo foram considerados os dados de altimetria e tipos de solos. O primeiro corresponde a altitude do relevo, já o segundo a classificação do solo de acordo com suas características físicas reais. Os tipos de solos utilizados na simulação foram: manguezal, vegetação de terra firme, praia, água e uso antrópico, representados por valores inteiros. Esses dados foram atribuídos às células através do algoritmo de preenchimento do espaço celular do software *TerraView* 4.2.0 [Bezerra 2014]. Além desses, ao longo da simulação criam-se novos tipos de solo de acordo com a inundação. Além disso, a estruturação do armazenamento segue o modelo matricial, cuja representação supõe que o espaço pode ser tratado como uma superfície plana, onde cada célula está associada a uma porção do terreno [Câmara et al. 2001]. Sendo assim, isso configura uma representação cartesiana bidimensional, onde cada célula possui uma coordenada *(x,y)* associada a um valor ou conjunto de valores. Seguindo esse contexto, utilizaram-se 2 tecnologias de armazenamento: Arquivos de texto

(padrão CSV) e banco de dados MySQL, sendo a estruturação dos dados armazenados em quatro campos: linha, que corresponde à coordenada *x*; coluna, que corresponde à coordenada *y*; altimetria, representando a altitude da célula; e, uso, que armazena o tipo de solo associado à célula.

Em termos de implementação, aqui são tomadas as decisões levando em conta as aplicações às quais o sistema é voltado, a disponibilidade de algoritmos para tratamento de dados geográficos e o desempenho do hardware [Casanova et al. 2005]. Para codificação, compilação e depuração de todo o projeto utilizou-se o *Visual Studio 2015 Community Edition* na linguagem C++. Para paralelizar a aplicação, utiliza-se a biblioteca MPI 3.1 da Microsoft. Além disso, os mapas são construídos utilizado-se um algoritmo de mapeamento em *bitmap*, onde são aplicadas as coordenadas das células mapeadas a uma legenda em cores RGB, associadas a cada tipo de solo. A próxima seção descreve os detalhes da paralelização.

3. O Automato Celular Paralelo

Devido à grande quantidade de dados gerados pelas representações geográficas, e, devido ao seu tempo de processamento, necessita-se implementar soluções capazes de aumentar o desempenho e abrangência da área geográfica desse processamento. Uma das principais fontes de aceleração é a paralelização das operações [Matloff 2011]. Através dessa metodologia podem-se processar dados de maneira concomitante, produzindo resultados mais rapidamente. Nesse âmbito, a arquitetura paralela desse trabalho segue o padrão *Single Program Multiple Data* (SPMD), pois unidades de processamento trabalham de forma independente, acessando locais de memória diferentes. A Figura 1, mostra como o espaço celular é dividido entre diversas unidades de processamento, sendo que a distribuição celular é feita através de uma instrução *MPI_Scatter*. Em resumo, considerando uma matriz $(M \times N)$, cada unidade de processamento irá receber M/P linhas, ou seja, submatrizes de $(M/P \times N)$. No caso de M ser ímpar o processo número P recebe o menor espaço celular completando-se o espaço restante com células nulas.

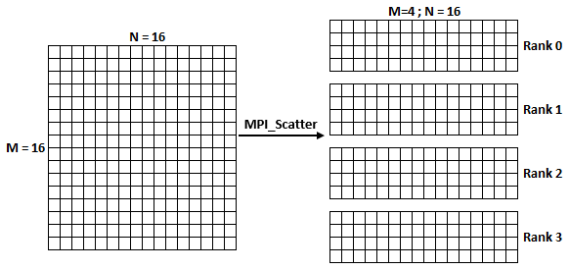


Figura 1. Divisão do espaço celular para cada processo

3.1. Aplicação do Autômato Celular

Os ACs são estruturas computacionais de implementação simples que permitem a manipulação direta de seus parâmetros para o estudo de sua dinâmica [Wolfram 2002]. Eles são definidos por seu espaço celular e suas regras de transição, que proporcionam mudanças nos estados das células a cada intervalo de tempo bem definido. Isso é possível através de um sistema de vizinhança, a qual foi utilizada adotando-se a estratégia de *Moore*, que, segundo [Wolfram 2002], inclui vizinhos diagonais e, portanto, envolve um total de 8 vizinhos para cada célula como mostrado na Figura 2.

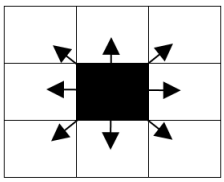


Figura 2. Vizinhança de *Moore*

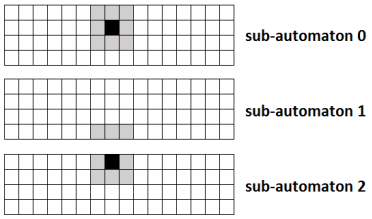


Figura 3. Casos de processamento

As regras de transição influenciam diretamente no estado da vizinhança em si. Através da Figura 2, observam-se as setas direcionadas aos vizinhos. Isso torna intuitiva a ideia de inundação, visto que a água em seu estado líquido avança em qualquer direção caso não tenha barreiras mais altas. Nesse contexto, a altura de cada célula é determinante para a regra de transição aqui utilizada. Generalizando-a, tem-se os simples passos: (i) Verifica-se se a célula atual tem o tipo de solo MAR ou tem algum solo inundado; (ii) para cada célula vizinha, verifica-se se ela tem o tipo de solo diferente de algum solo inundado; (iii) se sua altimetria for menor que a da célula principal então ela é inundável e entra no cálculo de fluxo da água; (iv) calcula-se o fluxo baseando-se quantidade de células inundáveis; (v) inundam-se as vizinhas; e (vi) a partir da inundação, criam-se novos tipos de solos, por exemplo, se o solo vizinho anterior era vegetação, então após a inundação passa a ser vegetação inundada.

O cálculo de fluxo da água entre as células é feito por $fluxo = am/ci$, no qual *am* é o valor do aumento do nível do mar a cada iteração conforme [Bezerra 2014] e *ci* é a quantidade de células inundáveis calculadas no passo anterior. Diante disso, e sabendo-se que *viz.alt* corresponde à altimetria da célula vizinha candidata à inundação, utiliza-se a seguinte condição para se aplicar o processo de inundação: ***Se*** *am* $>=$ *viz.alt* + *fluxo* ***Então*** *inunda-se vizinho*.

Diferentemente da versão sequencial, na qual todos os vizinhos são acessíveis por uma célula, na versão paralela deve-se observar se a vizinhança de uma célula não está contida na submatriz que foi enviada a outra unidade de processamento. Nesse contexto, os subautômatos envolvidos devem se comunicar entre si através de comunicação ponto-a-ponto. A Figura 3 mostra um exemplo na qual os vizinhos do subautômato 0 se encontram na mesma unidade de processamento, enquanto que parte da vizinhança do subautômato 2 encontra-se no subatômato 1. Assim, levando-se em conta o segundo caso mencionando, foram implementadas duas funções de comunicação: *get_cell* e *update_cell*, que são utilizadas para requisitar e atualizar uma célula de um subautômato adjacente, respectivamente. Para essa tarefa foi criado um tipo de dados específico para se enviar nas mensagens entre os processos, utilizando-se a função *MPI_Type_create_struct*, que contém as informações $intlinha$, $intcoluna$, $doublealtimetria$ e $intuso$, representando respectivamente a coordenada x, a coordenada y, a altimetria e o tipo de uso do solos.

4. Simulação e Resultados

As simulações foram executadas em um computador com processador *Intel core* I5, que contém 4 núcleos de processamento e 4 *gigabytes* de memória RAM. Para aplicação dos valores de elevação do nível do mar foram realizados 88 eventos com elevação total de

0,011 m a 0,97 m entre 2012 e 2100, distribuídos como uma progressão aritmética de razão 0,011m como apresentado em [Bezerra 2014]. Para aplicação do caso de teste e geração dos mapas utilizou-se o banco de dados da ilha do Maranhão, com escala de 1 HA, onde dados foram extraídos da EMBRAPA e do catálogo de imagens do INPE, conforme [Bezerra 2014]. A métrica de desempenho em relação ao paralelismo é o speedup que é calculado pela razão entre tempo sequencial e tempo paralelo ($sp = Ts/Tp$). A Tabela 1 exibe o resultado geral do processo de inundação na simulação paralela, na qual cada coluna representa a quantidade inundada em hectares (HA) para diferentes tipos de solo. De acordo com a tabela, nota-se maior perda de vegetação em relação aos outros solos simulados. A visualização do mapa inicial da ilha do Maranhão e após a simulação pode ser vista na Figura 4.

Tabela 1. Áreas inundadas (HA)

Período (anos)	Mangue	Uso antrópico	Praia	Vegetação
2013 a 2035	3313	1888	4	5586
2036 a 2058	4245	2078	48	6683
2059 a 2081	4812	3120	81	7273
2082 a 2100	5255	6462	104	7668

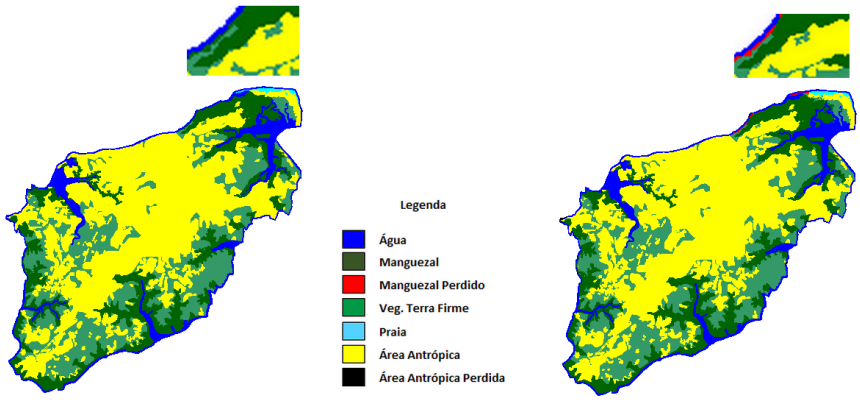


Figura 4. Mapa inicial e final, respectivamente, com destaque a perda de mangue.

A Tabela 2 apresenta o tempo de processamento em segundos para a quantidade de células, para os dados armazenados em arquivos e para os dados armazenados no SGBD, sendo que o processamento paralelo foi feito com 2 processos e 10 execuções. Na tabela pode-se observar que a medida que a quantidade de células aumenta, o speedup também aumenta, chegando a uma speedup superlinear a partir de 1,716,064 células para acesso via MySQL e 2,592,064 para arquivo, possivelmente porque a maioria das células vizinhas inundadas estavam no mesmo subautômato. Outra possibilidade de alcançar a superlinearidade é a regra de transição que deve ter sido calculada, em sua maioria, localmente. Observa-se também que acessar os dados via MySQL apresentou resultados ainda melhores que o acesso via arquivos.

5. Conclusão e trabalhos futuros

Este trabalho apresentou uma implementação paralela em MPI para simulação usando autômatos celulares. Os resultados mostraram que a versão usando 2 unidades de pro-

Tabela 2. Tempos de execuções para os conjuntos distintos de células

	Execução Serial		Execução 2 Processos			
Células	Arquivo	MySQL	Arquivo	MySQL	Sp Arquivo	Sp MySQL
97.401	28,150	20,190	28,052	20,068	1,00	1,00
Desvio	0,29	1,15	1,06	0,50		
504.064	44,472	37,508	41,254	32,356	1,08	1,16
Desvio	0,78	0,45	0,60	0,73		
1.020.064	72,152	62,656	47,391	38,538	1,52	1,62
Desvio	0,65	0,64	0,75	0,73		
1.716.064	105,833	100,280	55,925	48,294	1,89	2,08
Desvio	0,59	0,72	0,76	0,72		
2.592.064	151,987	146,274	66,222	58,272	2,29	2,51
Desvio	0,97	0,60	0,65	0,64		
3.648.064	206,395	197,963	78,143	67,270	2,64	2,94
Desvio	0,73	0,74	0,69	0,61		

cessamento alcançou *speedup* superlinear para algumas configurações. Por outro lado, algumas limitações foram encontradas, especialmente quando foram utilizados 4 processos na execução da simulação, pois como os subautomatos são menores a quantidade de comunicação exigida é muito grande, o que levou a tempos piores que a execução serial, ou seja, obteve-se desempenho sublinear. Como trabalhos futuros espera-se poder realizar a simulação em máquinas realmente multiprocessadas e/ou com GPUs.

Referências

Batty, M. and Xie, Y. (1994). From cells to cities. *Environment and Planning B: Planning and Design*, 21(7):S31–S48.

Bezerra, D. D. S. (2014). *MODELAGEM DA DINÂMICA DO MANGUEZAL FRENTE À ELEVAÇÃO DO NÍVEL DO MAR*. PhD thesis, INPE.

Câmara, G., Davis, C., and Monteiro, A. M. V. (2001). Introdução à ciência da geoinformação.

Casanova, M. A., Câmara, G., Davis, C., Vinhas, L., and Queiroz, G. R. (2005). *Banco de dados geográficos*. MundoGEO Curitiba.

Liu, X., Ma, L., Li, X., Ai, B., Li, S., and He, Z. (2014). Simulating urban growth by integrating landscape expansion index (lei) and cellular automata. *International Journal of Geographical Information Science*, 28(1):148–163.

Martins, L. G., Fynn, E., and Oliveira, G. (2011). Algoritmos genéticos usados na obtenção de autômatos celulares para resolução da tarefa de classificação de densidade no espaço bidimensional. *Horizonte Científico*, 5(2).

Matloff, N. (2011). Programming on parallel machines. *University of California, Davis*.

Wolfram, S. (2002). *A new kind of science*, volume 5. Wolfram media Champaign, IL.

Uma análise de custo e desempenho do modelo meteorológico BRAMS na nuvem computacional Azure

Willian M. Hayashida[1], Charles B. Rodamilans[1], Martin Tygel[1] e Edson Borin[1]

[1]Centro de Estudos de Petróleo (CEPETRO)
Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6052 – 13.083-970 – Campinas – SP – Brasil

w188705@g.unicamp.br, charles@ggaunicamp.com,

tygel@ime.unicamp.br, edson@ic.unicamp.br

Resumo. *Tecnologias de alta capacidade computacional estão sendo entregues como serviço com a computação em nuvem, e seus benefícios, tais como, flexibilidade e modelo de receita baseado no uso, apresentam uma oportunidade interessante para aplicações de HPC. Este trabalho apresenta uma análise de custo versus benefício da execução de uma aplicação meteorológica na nuvem computacional, comparando seu processamento em diversas configurações de máquinas virtuais otimizadas para computação.*

1. Introdução

A nuvem computacional permite o uso de largos sistemas computacionais customizáveis e seu modelo de receita baseia-se no uso. O usuário não se preocupa com o gerenciamento dos sistemas pois a instalação e manutenção dos equipamentos são de responsabilidade do provedor, dentre os principais, *Amazon Web Services* [Amazon 2018] e *Microsoft Azure* [Azure 2018]. A computação de alto desempenho, ou HPC[1], requer uma infraestrutura com alta capacidade computacional [Netto *et al*. 2018]. A nuvem pode ser uma solução para essas aplicações, ao oferecer a flexibilidade de aumentar ou diminuir a capacidade de computação conforme a demanda.

O BRAMS[2] (Sistema de Modelagem Atmosférica Regional Brasileiro) é um modelo numérico, projetado para pesquisa e previsão atmosférica em escala regional, com foco em química atmosférica, qualidade do ar e ciclos biogeoquímicos [Freitas *et al*. 2017]. Ele é baseado no Sistema de Modelagem Atmosférico Regional (RAMS[3]), o qual começou a ser desenvolvido em 1970 por Roger Pielke e outros [Pielke *et al*. 1992], e foi desenvolvido, no Brasil, para o clima tropical e sub-tropical pelo CPTEC [CPTEC 2018].

Neste trabalho investigamos o uso das máquinas virtuais do Microsoft Azure para uma simulação atmosférica com o modelo meteorológico BRAMS. Focamos a pesquisa na avaliação do comportamento da aplicação com diferentes configurações de máquinas virtuais, para, assim, encontrar o equilíbrio da capacidade de computação em uma única máquina, com custo elevado, e distribuído, em várias máquinas com o custo reduzido. Os experimentos realizados sugerem que o BRAMS tem seu desempenho limitado pelo

[1]do inglês: *High Performance Computing*
[2]do inglês: *Brazilian Regional Atmospheric Modeling System*
[3]do inglês: *Regional Atmospheric Modeling System*

acesso à memória do sistema e que seja mais vantajoso utilizar mais máquinas com menos núcleos do que menos máquinas com mais núcleos. Entretanto, em função do grande volume de comunicação entre os processos, o aumento do número de máquinas não garante necessariamente um aumento de desempenho.

Este texto está organizado da seguinte forma: a Seção 2 apresenta os trabalhos relacionados; a Seção 3 apresenta os materiais, o ambiente e a metodologia utilizada; a Seção 4 apresenta os resultados junto com a análise dos dados; por fim, a Seção 5 apresenta as conclusões do trabalho.

2. Trabalhos relacionados

Carreño e outros [Carreño *et al*. 2015] propuseram um estudo comparando a execução do modelo BRAMS em clusters locais com a nuvem computacional, focando em descrever os principais desafios e soluções. Nele é abordado a latência da rede, além do tempo de controle e operação do Azure. Os resultados do trabalho colaboram com a nossa análise da escalabilidade do BRAMS, estudada na seção 4.2, porém o artigo não aborda uma análise de custo detalhada como foi realizado neste trabalho.

3. Materiais e Métodos

A aplicação implementa o modelo meteorológico BRAMS e utiliza o padrão *Message Passing Interface* (MPI) para distribuir seus processos. Foi utilizado a implementação *mpich* versão *3.2.1* e compilou-se o BRAMS com os compiladores *mpicc* e *mpif90*, advindos da instalação do *mpich*. Na simulação meteorológica foi utilizado o dado de entrada que contém informações atmosféricas coletadas no dia 27/01/2014 pelo Centro de Previsão de Tempo e Estudos Climáticos [CPTEC 2018], disponibilizado na documentação do BRAMS [CPTEC-INPE 2018], e possui um tamanho de 722 MB.

A interface do MPI procurará o meio de transmissão mais rápido entre dois processos, portanto, para processos em um sistema de memória compartilhada as mensagens serão enviadas e recebidas pela memória do usuário, enquanto que, para dois processos em máquinas diferentes a interface procurará a melhor transmissão entre elas [Chai 2009].

Utilizamos as máquinas virtuais do Azure para a computação dos processos do BRAMS. Para a realização dos experimentos, escolheu-se a categoria computação otimizada da Azure e, dentro desta, as séries *F* e *Fv2*, como indicado na Tabela 1. A série *F* é equipada com o processador Intel Xeon E5-2673 v3 Haswell com frequência de operação de 2.4 Ghz e a série *Fv2* possui o processador Intel Xeon Platinum 8168 Skylake com frequência de 2.7 Ghz.

Cada máquina virtual foi instanciada a partir de uma imagem criada para estes experimentos, com Ubuntu 16.04 LTS, BRAMS, *mpich* e outros pacotes já instalados (gcc, g++, gfortran, build-essential, libtool, m4 e automake). O dado de entrada é copiado na inicialização das máquinas virtuais e o tempo dessa transferência, bem como as operações de disco antes de iniciar a execução, foram desprezados, sendo considerado somente o tempo total de execução do BRAMS.

Nos experimentos utilizou-se o mesmo número de processos MPI que o número total de núcleos. Para facilitar a descrição, denotaremos por configuração a escolha da instância e a quantidade de máquinas daquela instância, ou seja, a configuração 4_F32s_v2 indica que foram utilizadas 4 máquinas da instância F32s_v2s.

Tabela 1. Instâncias Azure.

Série	Tipo de Instância	vCPUs (núcleos virtuais)	RAM (GiB)	Custo (USD/h)
F	F1	1	2	0,05
F	F2	2	4	0,10
F	F4	4	8	0,21
F	F8	8	16	0,43
F	F16	16	32	0,87
Fv2	F2s_v2	2	4	0,10
Fv2	F4s_v2	4	8	0,20
Fv2	F8s_v2	8	16	0,40
Fv2	F16s_v2	16	32	0,81
Fv2	F32s_v2	32	64	1,63
Fv2	F64s_v2	64	128	3,26

4. Resultados Experimentais

A Figura 1 e a Figura 2 representam, respectivamente, os tempos de execução do BRAMS para as séries *F* e *Fs_v2* em relação ao número de núcleos por execução. Os gráficos foram linearizados colocando o eixo x em potências de 2.

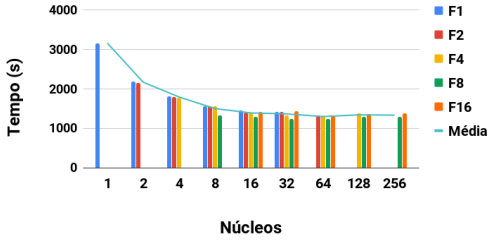


Figura 1. Tempo de execução da série F.

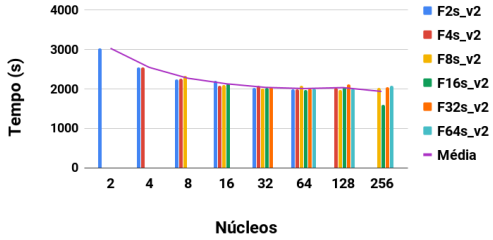


Figura 2. Tempo de execução da série Fv2.

Apesar do número de núcleos computacionais ser idêntico, o tempo de processamento com uma instância F64s_v2, que possui 64 núcleos, foi maior que o tempo de processamento com 8 máquinas da instância F8, que possui apenas 8 núcleos computacionais cada. Como múltiplos processos concorrem entre si pela mesma memória na instância F64s_v2, conjecturamos que o barramento de memória poderia ter sido o responsável por prejudicar o desempenho da aplicação. Esta hipótese é investigada na próxima seção.

4.1. Concorrência

Para investigar a hipótese de que a competição pela memória estava prejudicando o desempenho na instância F64s_v2, executamos 64 processos em 3 configurações diferentes desta instância: 1 máquina (executando os 64 processos), 2 máquinas (32 processos em cada) e 4 máquinas (16 processos em cada).

Como podemos observar na Tabela 2, o ganho de desempenho aumentou conforme se distribuiu a computação em múltiplas máquinas, e a utilização de *CPU* diminuiu proporcionalmente. Este resultado corrobora nossa hipótese de que o barramento de memória pode ser o responsável por prejudicar o desempenho da aplicação.

Tabela 2. 64 processos *MPI* em configurações diferentes da instância F64s_v2.

Número de máquinas	Tempo (s)	Ganho de desempenho [4]	Processos por máquina	Utilização da CPU [5](por máquina)
1	1987,3	1,00	64	100%
2	1504,2	1,32	32	50%
4	1461,5	1,36	16	25%

4.2. Escalabilidade

Nesta seção realizamos a análise de escalabilidade do BRAMS. Observa-se, nas Figuras 1 e 2, que o tempo de processamento adquire um comportamento constante a partir de 32 núcleos, indicando que o desempenho da aplicação deixou de escalar e, possivelmente, atingiu o limite da região de paralelismo, portanto, relacionado à lei de Amdahl [Hill e Marty 2008].

Esse resultado é importante para avaliar o custo *versus* benefício que será discutido posteriormente, pois instâncias menores, consequentemente menos custosas, apresentam um desempenho similar às instâncias maiores e com custo maior.

Para estudar a escalabilidade da aplicação escolheu-se a instância F8, pelo bom desempenho nos experimentos realizados (Figura 1 e Figura 2), e a instância F64s_v2, por ser a instância que foi utilizada nos experimentos realizados anteriormente, na Seção 4.1.

A Figura 3 e Figura 4 mostram os tempos de processamento do modelo BRAMS para diferentes configurações com as instâncias F8 e F64s_v2 respectivamente. Os resultados dão substrato à hipótese do limite de paralelismo, pois, novamente, os resultados mostram um bom ganho de desempenho para até 32 núcleos, mas, observa-se que para a instância F64s_v2, diferentemente da instância F8, se tem um aumento repentino do tempo de processamento para valores maiores. Essa característica foi analisada anteriormente na subseção 4.1.

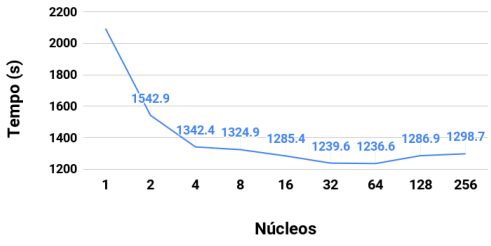


Figura 3. Tempo de execução da instância F8.

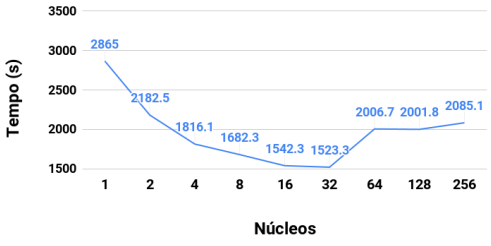


Figura 4. Tempo de execução da instância F64s_v2.

Idealmente espera-se que o ganho de desempenho seja proporcional ao aumento de núcleos, contudo esse comportamento não se verificou. De fato, as curvas de ganho indicam que se maximiza a região de paralelismo com 32 núcleos. Com base nos resultados da Seção 4.1, a hipótese mais forte para explicar a queda de desempenho é a competição pela memória, que pode causar perda significativa de desempenho [Borin *et al.* 2015].

[4]O ganho de desempenho é definido como a razão do tempo de referência (1987,3s) pelo tempo obtido
[5]do inglês: Central Processing Unit

4.3. Custo *versus* Desempenho

A relação custo *versus* desempenho pode ser avaliada multiplicando-se o tempo de uso de uma instância pelo custo do uso por hora. A Figura 5 apresenta graficamente o custo *versus* o desempenho de cada experimento. O eixo horizontal representa o custo total do experimento, em dólares (U$), assim, à medida em que se distancia do eixo vertical, tem-se um aumento do custo. O eixo vertical representa o tempo de execução, assim temos que quanto mais distante do eixo horizontal, maior o tempo de processamento.

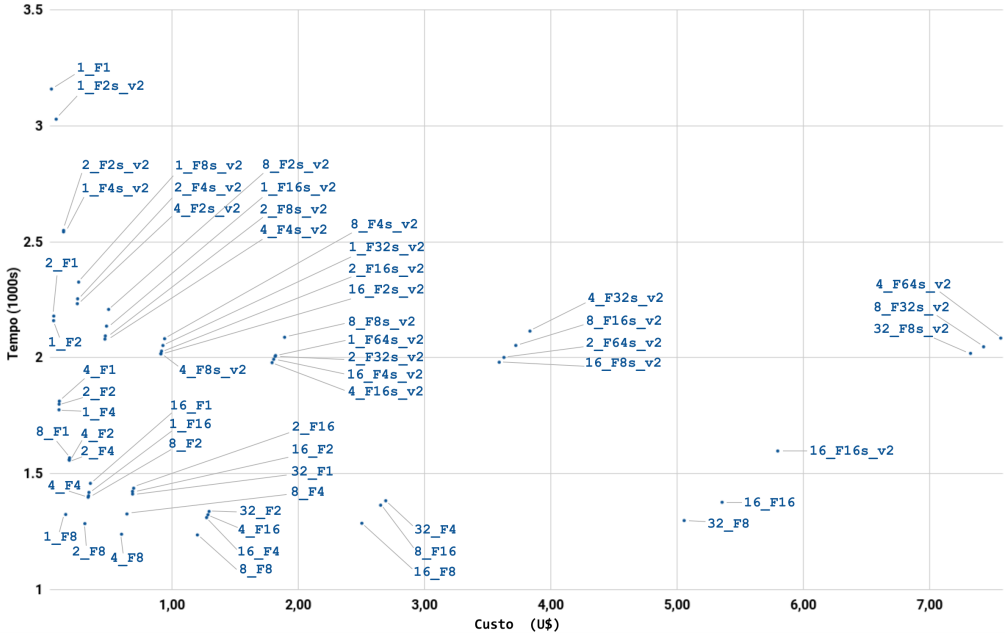


Figura 5. Custo por tempo de execução para as instâncias da Tabela 1.

A configuração de menor custo contém 1 máquina da instância F1 (1_F1), com custo de U$0,048 e tempo de 3158,8 segundos, e a configuração de melhor desempenho contém 8 máquinas da instância F8 (8_F8), com custo de U$ 1,204 e tempo de 1239,6 segundos. O custo da segunda é 25 vezes maior que o custo da primeira enquanto o ganho de desempenho é de aproximadamente 2,55 vezes. Existem outras configurações que se encontram mais próximas à origem, indicando um bom equilíbrio entre desempenho e custo. Por exemplo, a configuração 1_F8, com 1 máquina da instância F8, atinge um tempo de execução (1324,9 s) quase tão bom quanto o tempo de execução da configuração mais rápida (1239,6 s) enquanto tem um custo quase tão baixo (U$ 0,161) quanto o custo da configuração mais econômica (U$ 0,048).

5. Conclusões

Neste trabalho realizamos um estudo de custo versus benefício da execução de uma aplicação meteorológica na nuvem computacional Microsoft Azure.

O trabalho contemplou a execução de experimentos com diversas configurações distintas de máquinas virtuais na nuvem e mostrou que a configuração que oferece o melhor desempenho (8_F8) pode ter um custo muito maior do que a configuração que oferece o menor custo (1_F1). No entanto, existem configurações que oferecem um bom custo benefício (*e.g.* 1_F8), com um desempenho quase tão bom quanto a melhor configuração e um custo quase tão baixo quanto a configuração mais econômica.

Os experimentos também mostraram que o uso de uma instância maior, como a F64s_v2 com 64 núcleos, não resulta necessariamente no melhor desempenho. De fato, os resultados indicam que a aplicação utilizada faz uso intensivo da memória e que configurações de máquinas com muitos núcleos computacionais que compartilham a mesma memória podem sofrer perda de desempenho.

Como trabalhos futuros, pretendemos reproduzir os experimentos em outros provedores de nuvem computacional (*e.g.* Amazon AWS e Google Cloud), a fim de generalizar nossas conclusões. Além disso, utilizaremos ferramentas de perfilamento e análise que fazem uso dos contadores de desempenho em *hardware* para levantar mais evidências que corroborem as hipóteses levantadas para os problemas de desempenho encontrados.

Agradecimentos

Os autores agradecem à Petrobras, à Fapesp, ao CNPq e à CAPES pela ajuda financeira e à Microsoft pelo acesso ao Azure. Os autores também agradecem às equipes do *High Performance Geophysics* (HPG) e LMCAD pelo suporte computacional. E por fim, agradecemos ao Dr. Demerval Soares Moreira pela ajuda na execução do modelo BRAMS.

Referências

Amazon (2018). Amazon web services (aws). `https://aws.amazon.com`. Acessado em 20/07/2018.

Azure (2018). Microsoft azure. `https://azure.microsoft.com/`. Acessado em 20/07/2018.

Borin, E. *et al*. (2015). Accelerating engineering software on modern multi-core processors. *Advances in Engineering Software*, 84(C):77–84.

Carreño, E. D. *et al*. (2015). Challenges and solutions in executing numerical weather prediction in a cloud infrastructure. *Procedia Computer Science*, 51:2832 – 2837. International Conference On Computational Science, ICCS 2015.

Chai, L. (2009). High performance and scalable mpi intra-node communication middleware for multi-core clusters. *Ohio State University*.

CPTEC (2018). Centro de previsão de tempo e estudos climáticos. `https://www.cptec.inpe.br`. Acessado em 20/07/2018.

CPTEC-INPE (2018). A brams guide for beginners (first time users) - version 5.2.5. `http://brams.cptec.inpe.br/get-started/`. Acessado em 20/07/2018.

Freitas, S. R. *et al*. (2017). The brazilian developments on the regional atmospheric modeling system (brams 5.2): an integrated environmental model tuned for tropical areas. *Geoscientific Model Development*.

Hill, M. D. e Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38.

Netto, M. A. S. *et al*. (2018). Hpc cloud for scientific and business applications: Taxonomy, vision, and research challenges. *ACM Computing Surveys*, 51(1).

Pielke, R. A. *et al*. (1992). A comprehensive meteorological modeling system—rams. *Meteorology and atmospheric Physics*, 49(1-4):69–91.

Análise de Desempenho e Paralelização da Biblioteca Astronômica em Python CCDPROC

Luis R. Manrique, Luiz B. Galdino, Daniel Cordeiro

[1]Escola de Artes, Ciências e Humanidades
Universidade de São Paulo

{lmanrique, luiz.galdino, daniel.cordeiro}@usp.br

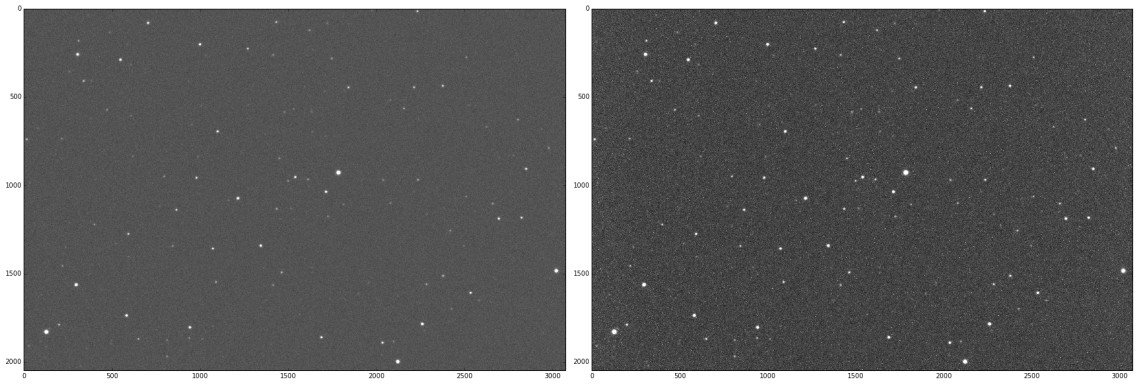
Abstract. *This paper analyzes the performance difference between the original serial code and our parallel version of a library written in Python called CCDPROC. Our code was parallelized using a SMP machine with two Intel® Xeon processors with 8 cores each (4 physical and 4 virtual) and had a speedup of 5.6x in some cases. The motivation for this study was to promote the discussion about the impact that a few changes in the code of serial libraries can cause in multi-core machines.*

Resumo. *Este artigo analisa a diferença no desempenho entre o código serial original e nossa versão paralela da biblioteca escrita em Python chamada CCDPROC. O algoritmo foi implementado de forma paralelizada em uma máquina SMP com dois processadores Intel® Xeon de 8 cores cada (4 físicos e 4 virtuais) e obteve um speedup de até 5,6x em alguns casos. O intuito desse estudo é promover a discussão sobre o impacto que alterações relativamente simples podem causar no desempenho de bibliotecas distribuídas originalmente sem otimizações para processadores multi-core.*

1. Introdução

A astronomia é uma ciência que gera muitos dados, obtidos através de simulações e observações com telescópios, trazendo desafios de como se armazenar, organizar, analisar e disponibilizar essas informações. Esses dados são pré-processados antes de serem utilizados nas pesquisas. Essa etapa é chamada de *redução dos dados* e, nela, são removidos dados que não são relevantes, o que ajuda na otimização do uso do espaço para armazenamento, além da correção ou remoção de distorções causadas por influências externas como poluição atmosférica, ruído nos sensores ou a proximidade dos objetos observados.

Uma ferramenta popular para reduzir dados astronômicos é o CCDPROC, cuja técnica utilizada é a de empilhar várias imagens de um mesmo ponto de observação, formando assim um cubo de imagens [Craig et al. 2015]. Com o cubo, pode-se calcular os valores médios, medianos ou somatórios de cada pixel, ajudando a reforçar os valores esperados e suavizando os *outliers*, como pode se verificar na Figura 1. Isso traz um grande desafio computacional, pois são muitas operações matemáticas a serem executadas na pilha de imagens devido ao grande número de pixels e ao uso excessivo de memória RAM para carregar os dados. Um exemplo são as imagens astronômicas captadas pelo telescópio T80S da Universidade de São Paulo que está localizado em Cerro Tololo, Chile [Coelho 2015] e que têm a resolução de 9200x9200 pixels.



(a) Imagem sem tratamento. (b) Mesma imagem após o uso do CCDPROC.

Figura 1. Tratamento de imagens astronômicas via CCDPROC. Fonte: http://nbviewer.jupyter.org/gist/mwcraig/06060d789cc298bbb08e

2. Metodologia

Através do uso de uma ferramenta de perfilamento (ou *profiling*, um software que permite a análise de desempenho em códigos por meio do monitoramento do uso de recursos de hardware como CPU e memória RAM [Abdulla 2010]), foi selecionada a parte do código a ser modificada. A ferramenta escolhida foi a *Intel® Vtune Amplifier*, que oferece suporte ao Python.

2.1. Detecção de Pontos Críticos

Com a ferramenta de *profiling* da Intel® foi possível identificar um ponto crítico do código que ocupa mais de `70%` do tempo total de CPU utilizado pelo programa, como visto na Figura 2. Nesse ponto é executada a função de normalização escolhida pelo usuário (median, average ou sum: métodos da biblioteca NumPy, chamados pelo CCDPROC, que serão executados em cada pixel do cubo de imagens, gerando uma única imagem como saída), e esse foi o trecho selecionado para ser paralelizado.

▼ <module>	98.9%	0s	ccd_teste.py	<module>	ccd_teste.py	0x7ffaaaa8aca8
▼ call_function	77.7%	0s	python3.6	call_function		0x19cb70
▼ average_combine	70.7%	0s	combiner.py	Combiner.average_combine(sel...	combiner.py	0x7ffa96e27940
▼ call_function	61.6%	0s	python3.6	call_function		0x19cb70

Figura 2. Tabela gerada pelo Intel® Vtune. A linha destacada mostra a função Python usada para combinar imagens (neste caso, `average_combine`) e o seu tempo de ocupação de CPU.

Nesse trecho é executada uma função da biblioteca NumPy, que, por sua vez, faz uso de bibliotecas matemáticas instaladas no computador. Nesse estudo, foram medidos os desempenhos das implementações da biblioteca BLAS/Lapack OpenBLAS e Intel® Math Kernel Library (MKL).

2.2. Estratégia de Paralelização

A adaptação do código para sua versão paralela se deu através do uso da biblioteca Joblib [Varoquaux 2009]. Para isso, foi necessário extrair um trecho do código original, criando um novo método: `worker` [Singh et al. 2013]. O método original (`combine`)

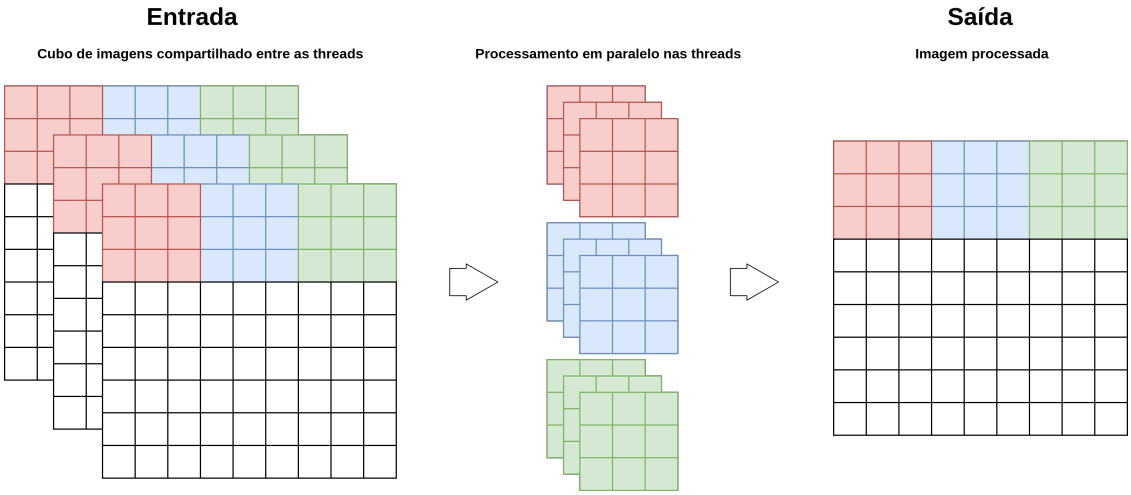


Figura 3. Estágio dos cubos de imagem na memória RAM, do início do processamento até a saída.

recebe um cubo de imagens, que, no código paralelo (`combineParallel`), é compartilhado para todas as threads. Através de parâmetros pré-definidos na chamada do `combineParallel`, que dizem respeito ao limite de uso de memória RAM e número de núcleos de processador, são definidos sub-cubos dentro do cubo original de imagens, e cada thread do método `worker` recebe os limites das coordenadas x e y que se deve processar. Os limites não foram definidos no eixo z, que representa o número de imagens empilhadas. Isso se deve ao fato de que as operações matemáticas ocorrem nesse eixo, e dividi-lo em partes exigiria processamento adicional. Após o processamento dos sub-cubos, são criadas matrizes que serão inseridas como partes da imagem resultante na saída do método `combineParallel`, conforme Figura 3. Na Figura 4 é apresentado o fluxo de processamento após as modificações.

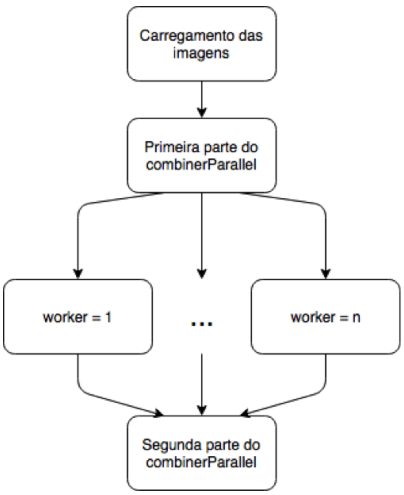


Figura 4. Fluxo de processamento do método paralelo de combinação de imagens.

Um último ponto observado foi o grande consumo de memória RAM. Sua redução pode influenciar no tempo de execução do programa, já que mais dados úteis poderiam

ficar disponíveis na memória e menos ciclos de escrita e leitura deveriam ser feitos. Nesse sentido, fizemos apenas um estudo das possibilidades, e, portanto, esse item não constará na seção de resultados.

3. Resultados

Nesta seção serão apresentados os resultados do uso de duas implementações da biblioteca matemática BLAS/Lapack: Intel® MKL, que pode ser baixada no site da Intel® e utilizada pelo pacote NumPy; e GNU LAPACK/BLAS, que, na versão do NumPy distribuída através do repositório oficial do Ubuntu 16.04, é a biblioteca padrão. Essa biblioteca é responsável pelos cálculos de média, somatório e mediana do método que combina as imagens. Também serão apresentados os resultados comparativos de desempenho do código paralelo em relação ao código original e resultado de escalabilidade do código paralelo para diversos números de núcleos de processador. Os recursos computacionais utilizados foram:

- Sistema Operacional: Ubuntu 16.04 Desktop 64bits;
- Kernel: 4.4.0-127;
- Processador: 2 x Intel Xeon E5620 4 cores (8 HT) 2,4GHz;
- Memória RAM: 16GB;
- Bibliotecas:
  - Intel®: NumPy versão 1.14.3 com MKL 2018.0.2;
  - LAPACK/BLAS: NumPy versão 1.11.0 com LAPACK/BLAS 3.6.

3.1. Bibliotecas Matemáticas

Para gerar o gráfico da Figura 5, foram executadas trinta vezes cada um dos métodos (média, somatório e mediana), e com os resultados obtidos foi calculada a sua média. Esses testes utilizaram o mesmo conjunto de imagens tanto para o NumPy com a biblioteca Intel® MKL, quanto a LAPACK/BLAS.

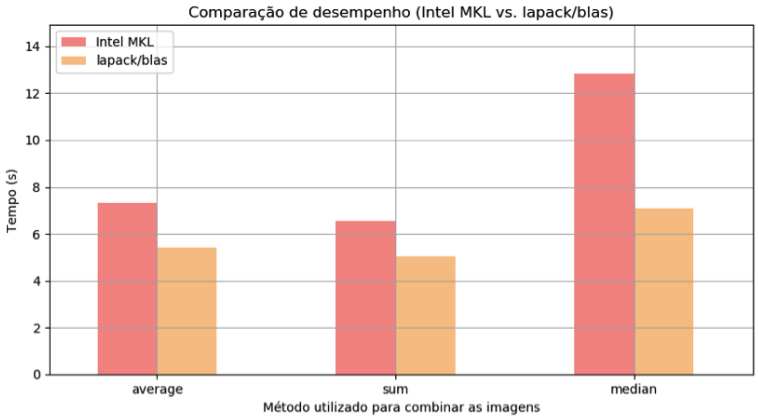


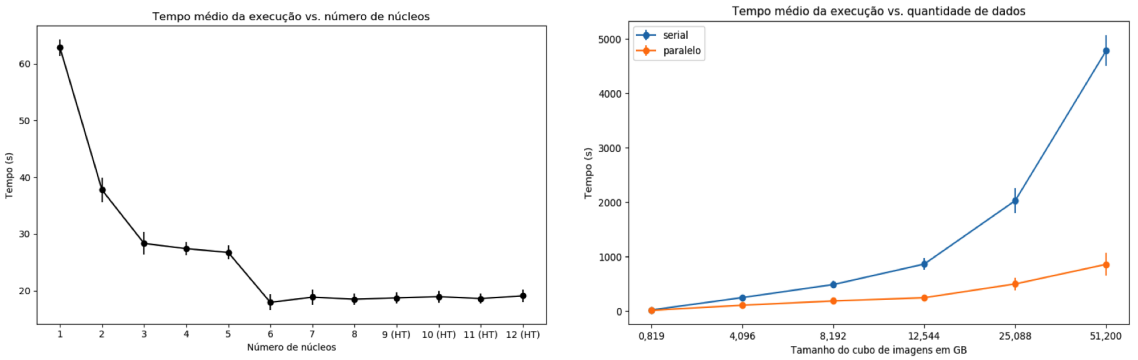
Figura 5. gráfico comparativo de desempenho Intel® MKL vs. LAPACK/BLAS.

Através dos resultados obtidos foi possível perceber que a biblioteca GNU LAPACK/BLAS obteve um melhor desempenho para cada um dos três métodos utilizados no método `combiner`, essa diferença de desempenho aumenta quando o método utilizado faz a ordenação dos dados.

3.2. Código Paralelo

O código paralelo, escrito pelos autores deste trabalho, foi testado em um computador com dois processadores Xeon de quatro núcleos cada, que têm suporte à tecnologia Intel® Hyper Threading (HT), que são núcleos virtuais de processamento. Assim, o sistema operacional Ubuntu 16.04 reconhece 16 núcleos de processador, sendo oito reais e oito virtuais.

A forma utilizada para medir o desempenho e determinar se o código está escalando, conforme a oferta de núcleos do processador aumenta, foi a execução de trinta vezes o método `combineParallel` para o cálculo da mediana (que é a função com maior demanda de processamento, como verificado no teste anterior das bibliotecas), com o uso da biblioteca GNU LAPACK/BLAS e a computação da média desses resultados. Esse mesmo teste foi executado para as configurações de um até doze núcleos, testando, assim, o desempenho na tecnologia HT.



(a) Gráfico do tempo de execução vs. número de núcleos. Cubo de 20 imagens com resolução de 3200x3200 (1,6 GB).

(b) Gráfico do tempo de execução vs. tamanho do cubo de imagens.

Figura 6. Gráficos de tempos médios de execução.

Como observado no gráfico da Figura 6(a), o desempenho tem uma melhora significativa quando se aumenta o número de núcleos disponíveis até seis. Após essa quantidade, o tempo médio fica estagnado em torno de 18,5 s. Não houve melhoria com o uso do HT, já que a estagnação ocorre antes do esgotamento dos núcleos reais. Para efeito de comparação, o desempenho do código para um núcleo é de 62,8 s contra 17,9 s para seis núcleos. Isso dá uma melhora no desempenho de aproximadamente 3,5 vezes.

Foram executados diversos testes com conjuntos de imagens diferentes para assegurar que esses resultados na melhoria de desempenho são consistentes. No gráfico da Figura 6(b), cada ponto nas curvas representa um valor médio para o resultado de trinta execuções dos métodos serial e paralelo, com seus respectivos desvios padrão. No eixo x foi calculado o tamanho do conjunto de imagens das execuções em GB. Foi possível perceber uma correlação entre o tamanho do conjunto de dados e a melhoria no desempenho. No maior conjunto de dados testado, de 51,2 GB, o resultado foi de 4785,03 s para o código serial contra 857,93 s para o código paralelo, representando uma melhora de aproximadamente 5,6 vezes.

4. Considerações finais

Este trabalho apresentou a análise e paralelização do código do CCDPROC, um arcabouço para o desenvolvimento de aplicações da área da astronomia escrito em Python. Apresentamos uma análise do desempenho com ferramentas de *profiling* e uma estratégia de paralelização que permitiu um *speedup* de até 5,6 utilizando-se 8 núcleos, se comparado à execução sequencial. Otimizações como esta tem significativa importância prática, uma vez que os telescópios, quando estão na sua fase de plena atividade, geram, todas as noites, muitas imagens que devem ser armazenadas na forma reduzida, de forma a evitar acúmulo de dados e atrasos nas pesquisas.

Mostramos que pequenas otimizações, como o uso de uma implementação de uma biblioteca matemática específica, ou paralelização de trechos do código, podem ter um grande impacto no desempenho final de uma biblioteca. Elas vão desde o uso de uma biblioteca matemática específica, até pequenas alterações no código fonte do CCDPROC, visando a paralelização de sua função de combinação de imagens. Com a paralelização realizada, o código foi capaz de melhorar a utilização dos recursos computacionais disponíveis, bem como usar múltiplos núcleos, recurso presente na grande maioria dos processadores modernos.

Os resultados obtidos ressaltam a necessidade de utilizar-se uma ferramenta de detecção de pontos críticos para auxiliar na escolha dos trechos a serem otimizados. Nossa análise mostrou que uma pequena parcela do código consumia mais de `70%` do tempo de execução e que otimizá-la causou um grande impacto em seu desempenho.

Dentre os trabalhos futuros, destaca-se a otimização do consumo de memória da ferramenta. O perfilamento da aplicação mostrou que uma grande quantidade de memória RAM é utilizada. Técnicas simples de gerenciamento de memória como o uso da chamada de sistema `mmap()` já são utilizadas pelo CCDPROC, o que significa que a redução do consumo de memória necessitará de estratégias mais sofisticadas de organização dos dados em memória.

Referências

Abdulla, M. F. (2010). Manual and fast c code optimization. *Ann. Univ. Tibiscus Comp. Sci. Series VIII*.

Coelho, P. (2015). Data management for S-PLUS. IAG/USP.

Craig, M. W., Crawford, S. M., Deil, C., Gomez, C., Günther, H. M., Heidt, N., Horton, A., Karr, J., Nelson, S., Ninan, J. P., Pattnaik, P., Rol, E., Schoenell, W., Seifert, M., Singh, S., Sipocz, B., Stotts, C., Streicher, O., Tollerud, E., Walker, N., and ccdproc contributors (2015). ccdproc: Ccd data reduction software. *Provided by the SAO/NASA Astrophysics Data System*.

Singh, N., Browne, L.-M., and Butler, R. (2013). Parallel astronomical data processing with python: Recipes for multicore machines. *Astronomy and Computing, Volume 2*.

Varoquaux, G. (2009). Joblib: running python functions as pipeline jobs.

Avaliação de Soluções para Alocação de Aplicações Distribuídas em Ambientes de Nuvem

Ana L. R. Herrmann[1], **Guilherme Galante**[1]

[1]Ciência da Computação
Universidade Estadual do Oeste do Paraná (UNIOESTE)
Caixa Postal 711 – 85.819-110 – Cascavel-PR

{ana.hermann, guilherme.galante}@unioeste.br

Abstract. *In IaaS public providers, the user does not have accurate information about the allocation of their virtual machines and how they are connected. This allocation model can lead to significant variations in mean latency between the allocated instances, which can result in degradation of performance of distributed applications. This work evaluates two solutions, Cloudia and Choreo, that aim to improve the allocation of distributed application components among the available VMs considering the latency variability. According to the experiments, both solutions were effective, reducing the execution time of the applications when using the propoded allocations.*

Resumo. *Nos provedores públicos de IaaS, o usuário não possui informação precisa sobre a alocação de suas máquinas virtuais e como estão conectadas. Esse modelo de alocação pode levar a variações significativas na latência média entre as instâncias alocadas, o que pode resultar em degradação significativa do desempenho de aplicações distribuídas, a menos que seja tomado cuidado em como os componentes da aplicação são mapeados para as instâncias. Neste trabalho avalia-se duas soluções, Cloudia e Choreo, que visam melhorar a alocação dos componentes de aplicações distribuídas entre as VMs disponíveis considerando essa questão da latência. De acordo com os experimentos ambas as soluções mostraram-se efetivas, reduzindo o tempo de execução das aplicações ao utilizar as alocações propostas.*

1. Introdução

Na última década, a computação em nuvem ganhou grande popularidade como uma plataforma promissora para a execução de aplicações distribuídas. Seguindo a ideia de Infraestrutura como Serviço (IaaS), algumas nuvens públicas, como o Amazon EC2[1] ou o Google Cloud Platform[2], permitem que seus clientes adquiram conjuntos de recursos de computação por meio de pagamento por uso de curto prazo. De modo geral, os recursos de computação adquiridos são entregues em forma de máquinas virtuais (VMs) hospedadas dentro dos data centers dos provedores.

Atualmente, os provedores de nuvem pública não expõem o escalonamento das instâncias virtuais ou a topologia de rede [Battré et al. 2011]. Os clientes sabem apenas que suas VMs estão em execução em algum lugar na nuvem, sem saber quais VMs

[1]aws.amazon.com/ec2
[2]cloud.google.com

estão hospedadas no mesmo servidor físico, quais servidores compartilham o mesmo rack ou data center, etc. Esse modelo de alocação pode levar a variações significativas na latência média entre as instâncias alocadas pelo cliente, o que pode resultar em degradação significativa do desempenho em aplicações sensíveis à latência, a menos que seja tomado cuidado em como os componentes da aplicação são mapeados para as instâncias [Zou et al. 2015].

Infelizmente, os provedores não disponibilizam serviços que permitam que o cliente mapeie sua aplicação de maneira inteligente, considerando essas questões. Nesse sentido, alguns trabalhos apresentam soluções que visam melhorar a alocação dos componentes de aplicações distribuídas entre as VMs disponíveis considerando a variação da latência.

Neste trabalho duas soluções são avaliadas, Cloudia [Zou et al. 2015] e Choreo [LaCurts et al. 2013], com o objetivo de verificar sua efetividade nesta tarefa. As soluções foram avaliadas em uma nuvem IaaS real, a Google Compute Engine, na qual se realizou um conjunto de experimentos com aplicações com diferentes características. De acordo com os experimentos, ambas as soluções mostram-se efetivas na melhoria da alocação das aplicações em ambientes de nuvem, reduzindo o tempo de execução das aplicações testadas em até 58%.

O restante do trabalho é organizado da seguinte forma. A Seção 2 apresenta os algoritmos de escalonamento avaliados nesse trabalho. Na Seção 3 realiza-se a avaliação experimental. Por fim, a Seção 4 conclui este trabalho.

2. Soluções para Alocação em Ambientes de Nuvem

Nesta seção apresenta-se as duas soluções para a alocação de aplicações distribuídas que serão avaliadas. Ambas são voltadas para em ambiente de nuvem IaaS públicas onde nenhuma informação sobre a alocação das instâncias ou sobre a rede são fornecidas pelos provedores. É importante salientar que as soluções visam oferecer alocações sub-ótimas.

As duas soluções funcionam de maneira semelhante. Como entrada, tem-se (1) o grafo de comunicação da aplicação (representado como uma matriz de adjacências), com os respectivos pesos de cada aresta e (2) a matriz contendo os dados de latência par-a-par. Como saída, obtém-se um conjunto de mapeamentos do componente c para a instância i. Nas seções a seguir, apresenta-se o algoritmo usado por cada uma das soluções. Para mais detalhes sobre os algoritmos recomenda-se a leitura dos artigos originais.

2.1. Cloudia

A alocação dos componentes no Cloudia é realizado conforme apresentado no Algoritmo 1. No algoritmo, $D(x)$ define uma função que retorna a tarefa que está alocada em uma máquina x, enquanto $D^{-1}(v)$ define uma função que retorna a máquina virtual onde está alocada uma tarefa v. Em linhas gerais, o algoritmo parte de uma alocação arbitrária e a partir dela tenta minimizar o custo (latência) das demais alocações.

2.2. Choreo

A alocação dos componentes no Choreo é realizado conforme apresentado no Algoritmo 2. Essa solução baseia-se na alocação de tuplas $< i, j, b >$ que significa que o nó i envia b bytes para j. No Choreo o objetivo é tentar alocar as máquinas que possuem maior comunicação entre si em links com menor latência.

Algoritmo 1: Cloudia()

```
1  begin
2      S → conjunto de instâncias
3      G = (V, E) → grafo de comunicação da aplicação
4      Encontrar o link u_0 → v_0 de menor custo ∈ S × S
5      Encontrar uma aresta arbitrária (x, y) ∈ G
6      D(x)=u_0
7      D(y)=v_0
8      for i=1 até ||V|| − 2 do
9          c_min=∞
10         foreach (u, v) ∈ S × S do
11             if D^{-1}(v) não foi definido e D^{-1}(u) possui vizinhos não alocados then
12                 if latencia(u, v) < c_min then
13                     c_min = latencia(u, v)
14                     u_min = u
15                     v_min = u
16                 end
17             end
18         end
19         w = um dos vizinhos não alocados de D^{-1}(u)
20         D(w) = v_min
21     end
22 end
```

Algoritmo 2: Choreo()

```
1  begin
2      transfers = conjunto de tuplas < i, j, b > em ordem decrescente de b.
3      foreach < i, j, b > em transfers do
4          if i já foi alocada em uma máquina k then
5              P = conjunto de links k → N, ∀ nós N
6          end
7          if j já foi alocada em uma máquina ℓ then
8              P = conjunto de links M → ℓ, ∀ nós M
9          end
10         if i e j não foram alocadas then
11             P = conjunto de links M → N, ∀ nós M e ∀ nós N
12         end
13         foreach link em P do
14             if se a alocação de i em m ou j em n exceder as restrições de CPU de m ou n
                 then
15                 remover m → n de P
16             end
17         end
18         foreach link em P do
19             latência(m,n) = latência do link m → n
20         end
21         Alocar i e j em m e n in latência(m,n) seja minimizada
22     end
23 end
```

3. Experimentos e Resultados

3.1. Ambiente Computacional

O Compute Engine é o serviço de IaaS do Google Cloud Platform. Esse serviço consiste no uso de VMs rodando nos data centers do Google e conectadas à sua rede de fibra ótica. Estes recursos estão distribuídos entre diferentes regiões e zonas. Região é uma localização geográfica específica onde você pode executar os seus recursos. Cada região tem uma ou mais zonas. Por exemplo, a região *us-central1* denota uma região na região central dos Estados Unidos com as zonas *us-central1-a, us-central1-b, us-central1-c e us-central1-f.* A nuvem oferece vários tipos de máquina, que definem uma coleção específica de recursos de hardware virtualizados disponíveis para uma instância de máquina virtual.

Todas as máquinas possuem a mesma configuração: um núcleo de processamento; 4 GB de memória RAM; 40GB de armazenamento não-volátil. O sistema operacional utilizado foi o Ubuntu 16.04, e as configurações gerais são as padrões da plataforma.

3.2. Analisando a Latência

Para comprovar a problemática de heterogeneidade de latências entre um conjunto de máquinas virtuais dentro de um mesmo projeto em nuvem, foi calculada a latência entre 10 VMs instanciadas no Google Cloud Engine, alocadas em três zonas distintas (*us-east1-b, us-west1-a* e *us-central1-c*). O cálculo foi feito com base nas médias dos dados apresentados pelo comando *ping*. Os resultados são apresentados na Figura 1, que apresenta a média das latências entre as zonas.

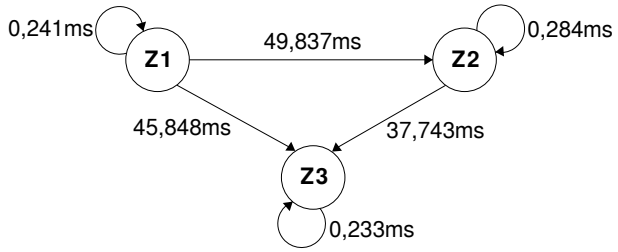


Figura 1. Latência média entre diferentes zonas.

3.3. Aplicação dos Algoritmos

Para a execução dos experimentos deste trabalho, foram utilizadas aplicações com topologias de comunicação Todos-para-Todos e Cliente-Servidor. A aplicação Todos-para-Todos tem como objetivo mimetizar o comportamento de uma aplicação *peer-to-peer*, nas quais há comunicação entre boa parte das tarefas. Por sua vez, na aplicação Cliente-Servidor ocorre o envio de dados das tarefas clientes para a tarefa Servidor, simulando uma operação de redução (todos para um).

Na aplicação Todos-para-Todos, cada uma das 10 tarefas faz o envio de um arquivo para as outras 9 tarefas. Na aplicação Cliente-Servidor as tarefas 2 a 10 enviam dados para a tarefa 1. O tamanho das mensagens enviadas pela aplicação Todos-para-Todos é apresentado na Tabela 1 e pela aplicação Cliente-Servidor é apresentada na Tabela 2. As mensagens foram geradas aleatoriamente e com tamanhos arbitrários.

Tabela 1. Matriz de comunicação para a aplicação Todos para Todos (KB).

	1	2	3	4	5	6	7	8	9	10
1	0	223920	147320	229250	299510	14630	271600	312990	202350	187660
2	59910	0	169960	240000	251200	39250	244890	46170	202890	79980
3	190210	87690	0	205920	80480	293660	44210	84180	212150	292720
4	112130	202900	284430	0	17367	286680	222610	247040	180270	49170
5	242260	78520	42330	67050	0	163700	95270	219520	141580	155080
6	257580	255690	253860	11570	162550	0	315860	254570	290700	266360
7	17960	132990	71820	127300	303880	15300	0	115130	110650	168150
8	189060	43850	52890	214820	304090	291080	15470	0	273130	12650
9	56380	230290	93580	199080	47530	124260	185400	180640	0	269390
10	78500	20440	187710	324710	239100	167280	245280	25410	35110	0

Tabela 2. Matriz de comunicação para a aplicação Cliente-Servidor (KB).

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	0
2	242300	0	0	0	0	0	0	0	0	0
3	40040	0	0	0	0	0	0	0	0	0
4	206780	0	0	0	0	0	0	0	0	0
5	116440	0	0	0	0	0	0	0	0	0
6	86330	0	0	0	0	0	0	0	0	0
7	96850	0	0	0	0	0	0	0	0	0
8	141370	0	0	0	0	0	0	0	0	0
9	105790	0	0	0	0	0	0	0	0	0
10	86790	0	0	0	0	0	0	0	0	0

Utilizando esses dois cenários, as soluções Cloudia e Choreo são comparadas por uma abordagem *First-Fit*, na qual a tarefa 1 é alocada na máquina 1, a tarefa 2 é alocada na máquina 2 e assim sucessivamente.

Para a aplicação Todos-para-Todos, após aplicar o Cloudia e Choreo, obteve-se a alocação apresentada na Tabela 3 para cada uma das tarefas. Na Tabela 4 apresenta-se o tempo de execução da aplicação utilizando as distintas alocações. Pode-se observar que a melhoria obtida pelo uso das soluções Cloudia e Choreo é superior a 40%, com uma diminuição de aproximadamente 58 segundos no tempo de execução.

Tabela 3. Tarefa da aplicação, e as VMs escolhidas por cada um dos métodos.

Tarefa	VM (First-Fit)	VM (Choreo)	VM (ClouDiA)
1	1	10	10
2	2	1	6
3	3	8	9
4	4	4	4
5	5	2	7
6	6	9	3
7	7	5	2
8	9	3	1
9	9	7	5
10	10	6	8

Tabela 4. Tempo de execução (segundos) para a aplicação Todos-para-Todos.

	ClouDiA	Choreo	First Fit
Tempo Médio	78,5	79,33	136,5
Comparação com o FF	57,51%	58,12%	100%

Para a aplicação Cliente-Servidor, obteve-se uma mesma alocação com os algoritmos do Cloudia e Choreo, conforme apresentada na Tabela 5. Na Tabela 6 apresenta-se o

tempo de execução da aplicação utilizando as distintas alocações. Pode-se observar que a melhoria obtida pelo uso das soluções Cloudia e Choreo é de aproximadamente 58%, com uma diminuição de aproximadamente 16 segundos no tempo de execução.

Tabela 5. Tarefa da aplicação, e as VMs escolhidas por ambos os métodos.

Tarefa	VM (First-Fit)	VM (Choreo e Cloudia)
1	1	2
2	2	6
3	3	3
4	4	1
5	5	4
6	6	8
7	7	7
8	8	9
9	9	10
10	10	5

Tabela 6. Tempo de execução (segundos) para a aplicação Cliente-Servidor.

	ClouDiA	Choreo	First Fit
Tempo Médio	12	12	28
Comparação com o FF	42,85%	42,85%	100%

4. Conclusão

As características da infraestrutura das nuvens públicas, podem causar uma diferença significativa nas latências em comunicações utilizando diferentes pares de máquinas virtuais alocadas para um determinado cliente. Por consequência, aplicações sensíveis à comunicação podem ter seu desempenho afetado negativamente.

Esse impacto pode ser minimizado se forem utilizados técnicas para melhorar a alocação dos componentes de aplicações distribuídas entre as VMs disponíveis considerando essa variação de latência.

Neste trabalho, duas soluções para a alocação de aplicações em nuvens foram avaliadas, Cloudia [Zou et al. 2015] e Choreo [LaCurts et al. 2013]. Ambas as soluções mostraram-se igualmente efetivas na melhoria das alocações das VMs, possibilitando a redução do tempo de execução das aplicações entre 40% e 58%.

Como trabalho futuro, pretende-se estudar outros algoritmos para a alocação de aplicações em nuvens públicas, assim como realizar testes mais completos com outros cenários e utilizando outras classes de aplicações.

Referências

Battré, D., Frejnik, N., Goel, S., Kao, O., e Warneke, D. (2011). Evaluation of network topology inference in opaque compute clouds through end-to-end measurements. In *IEEE CLOUD*, pages 17–24. IEEE Computer Society.

LaCurts, K., Deng, S., Goyal, A., e Balakrishnan, H. (2013). Choreo: Network-aware task placement for cloud applications. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 191–204, New York, NY, USA. ACM.

Zou, T., Bras, R., Salles, M. V., Demers, A., e Gehrke, J. (2015). Cloudia: A deployment advisor for public clouds. *The VLDB Journal*, 24(5):633–653.

Avaliação de Heurísticas de Mapeamento de Tarefas no MPSoC HeMPS

Ezequiel L. Vidal[1], **Aline V. de Mello**[1], **Ewerson L. S. Carvalho** [2], **Claudio Schepke**[1]

[1]Universidade Federal do Pampa (UNIPAMPA)
Av. Tiarajú, 810 – 97546-550 – Alegrete – RS – Brasil

[2]Universidade Federal do Rio Grande - (FURG)
Av. Itália, Km 8 – 96203-900 – Rio Grande – RS – Brasil

{ezequielvidal.ti, alinemello, ewersoncarvalho, schepke}@gmail.com

Abstract. *There is a tendency that MPSoCs will be composed of dozens or hundred of processing elements, allowing the execution of many tasks in parallel. Therefore, efficient resource allocation strategies need to be developed. In this context, this work investigates the performance of the heuristics for task mapping First Free, Nearest Neighbor, Path Load and Best Neighbor in HeMPS MPSoC. The Best Neighbor heuristic presented the best result in relation to the occupation of the communication channels of the MPSoC, with reduction of approximately 32% when compared to the First Free heuristic. However, Best Neighbor presented simulation time up to 24.21% higher than First Free heuristic due to the complexity of its algorithm.*

Resumo. *Há uma tendência de que MPSoCs sejam compostos por dezenas ou centenas de elementos de processamento, permitindo a execução de muitas tarefas em paralelo. Assim, estratégias de alocação de recursos eficientes precisam ser desenvolvidas. Neste sentido, este trabalho investiga o desempenho das heurísticas de mapeamento de tarefas First Free, Nearest Neighbor, Path Load e Best Neighbor no MPSoC HeMPS. A heurística Best Neighbor apresentou o melhor resultado em relação à ocupação dos canais de comunicação do MPSoC, com redução de aproximadamente 32% quando comparada a heurística First Free. No entanto, essa heurística apresentou tempo de execução até 24,21% superior a heurística First Free devido à complexidade de seu algoritmo.*

1. Introdução

Multi-Processor Systems on Chip (MPSoCs) são sistemas embarcados compostos por vários elementos de processamento (PEs), memórias e *IP-Cores* específicos, geralmente interconectados através de uma *Network on Chip* (NoC). Alguns exemplos incluem o processador de 72 núcleos Tile-Gx72 [Mellanox 2015], o processador MPPA de 256 núcleos da Kalray [De Dinechin et al. 2014] e mais recentemente o processador de 1000 núcleos desenvolvido em parceria entre a IBM e a universidade UC Davis [Bohnenstiehl et al. 2016].

O ato de escolher os melhores PEs de um MPSoC para alocar uma tarefa é denominado de mapeamento de tarefas [Mandelli 2011]. O mapeamento de tarefas está na classe de problemas *NP-Hard* [Garey and Johnson 1979], sendo um dos problemas de

otimização combinatória mais desafiadores que existem. Quando o número de PEs é pequeno, é possível obter soluções ótimas em tempo consideravelmente curto. Entretanto, MPSoCs possuem uma quantidade de PEs cada vez maior, o que torna imprescindível que estratégias de alocação de recursos eficientes sejam desenvolvidas [Ding et al. 2014].

Nesse contexto, o objetivo deste trabalho é avaliar o desempenho das heurísticas de mapeamento de tarefas *First Free, Nearest Neighbor, Path Load* e *Best Neighbor* no MPSoC HeMPS [Carara et al. 2009]. O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados sobre mapeamento de tarefas. A Seção 3 descreve a metodologia adotada para avaliação. Na Seção 4, os resultados são apresentados e uma análise é realizada. Por fim, a Seção 5 apresenta as considerações finais.

2. Trabalhos Relacionados

Conforme observado na Tabela 1, a maioria dos trabalhos relacionados utiliza mapeamento dinâmico para reduzir o consumo de energia (CE) e o tempo de execução (TE) das aplicações rodando sobre MPSoC, com exceção de [Ost et al. 2013] que foca apenas no consumo de energia. Somente os trabalhos de [Quan and Pimentel 2015] e [Carvalho et al. 2010] fazem uso de MPSoC com processadores heterogêneos, enquanto os demais utilizam MPSoCs com processadores homogêneos. Com relação ao tipo de controle de mapeamento, o centralizado é a estratégia adotada na maioria dos trabalhos, mesmo não sendo escalável. O único trabalho que apresenta um controle distribuído é proposto por [Mendis et al. 2015]. A maioria dos trabalhos utilizam a abordagem *Evitar Congestionamento*, cujo objetivo é diminuir o fluxo de dados nos canais de comunicação.

Tabela 1. Trabalhos Investigados sobre Mapeamento de Tarefas.

Trabalho	Otimização	Arquitetura	Controle	Abordagem
[Carvalho et al. 2010]	TE, CE	Heterogênea	Centralizado	Evitar Congest.
[Ost et al. 2013]	CE	Homogênea	Centralizado	Evitar Congest.
[Fattah et al. 2014]	TE, CE	Homogênea	Centralizado	Evitar Congest.
[Mendis et al. 2015]	TE, CE	Homogênea	Distribuído	Remapeamento
[Quan and Pimentel 2015]	TE, CE	Heterogênea	Centralizado	Híbrida
[Ng et al. 2015]	TE, CE	Homogênea	Centralizado	Desfragmentação
[Huang et al. 2015]	TE, CE	Homogênea	Centralizado	Evitar Congest.
[Singh et al. 2016]	TE, CE	Homogênea	Centralizado	Híbrida
[Seidipiri et al. 2016]	TE, CE	Homogênea	Centralizado	Evitar Congest.

TE = Tempo de Execução; CE = Consumo de Energia.

O presente trabalho implementa e avalia as heurísticas de mapeamento de tarefas *First Free* (FF), *Nearest Neighbor* (NN), *Path Load* (PL) e *Best Neighbor* (BN) [Carvalho et al. 2010] no MPSoC HeMPS. Esse MPSoC possui arquitetura homogênea com controle centralizado. A avaliação considera o tempo de execução e a ocupação dos canais de comunicação, a fim de evitar congestionamentos.

3. Metodologia

A investigação conduzida neste trabalho é realizada sobre um MPSoC HeMPS com dimensão 5x5, totalizando 25 PEs. O PE 0x0 não recebe nenhuma tarefa pois tem como

função gerenciar toda a comunicação. No MPSoC HeMPS, é papel do programador particionar uma dada aplicação paralela em diferentes tarefas.

A Figura 1 apresenta as três aplicações utilizadas neste trabalho, representadas por grafos dirigidos, onde os vértices representam as tarefas e as arestas representam o fluxo de dados entre elas. A aplicação *Dynamic Time Warping* (DTW) é utilizada para encontrar o alinhamento não-linear ótimo entre duas sequências de valores numéricos e representa o conjunto de aplicações que possuem alta taxa de comunicação entre as tarefas. *Moving Picture Experts Group* (MPEG) é uma aplicação de áudio e vídeo que representa as aplicações com carga elevada de processamento. Já *SYNTHETIC* é uma aplicação sintética que simula o envio e recebimento de um conjunto de dados, permitindo manipular a taxa de comunicação entre as tarefas.

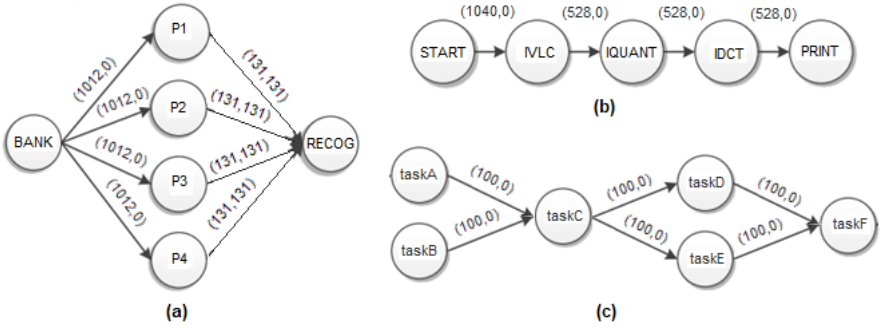


Figura 1. Aplicações utilizadas (a) DTW, (b) MPEG e (c) SYNTHETIC.

Cinco casos de testes foram elaborados. Nos casos de teste 1, 2 e 3, apenas uma aplicação (DTW, MPEG e SYNTHETIC, respectivamente) é executada no MPSoC. Já nos casos 4 e 5, respectivamente duas (DTW e MPEG) e três (DTW, MPEG e SYNTHETIC) aplicações são executadas concorrentemente no MPSoC. Cada caso de teste foi simulado para cada uma das heurísticas (FF, NN, PL e BN), totalizando 20 simulações.

As heurísticas de mapeamento de tarefas foram avaliadas utilizando as métricas: tempo de mapeamento (TM), tempo de execução (TE) e ocupação dos canais de comunicação (OC). TM é o tempo dedicado à execução do algoritmo de mapeamento. TE é o tempo total de um caso de teste, englobando TM. TM e TE são apresentados em milissegundos (ms). OC corresponde ao número total de *flits*[1] transmitidos em todos os canais de comunicação do MPSoC.

4. Resultados

A Figura 2 apresenta TM e TE para os casos de teste 1, 2 e 3. A aplicação MPEG possui a maior média de TE (9,36 ms) devido a sua carga de processamento, seguido de DTW (7,62 ms) e de SYNTHETIC (1,41 ms). Em relação ao TM, a heurística PL teve o pior desempenho em razão da complexidade de seu algoritmo ($O(x^3)$), alcançando um TM médio de 0,89 ms. A heurística FF teve um desempenho 29,92% melhor que PL (0,623 ms), seguido de NN com 29,83% (0,624 ms) e BN com 24,21% (0,674 ms).

[1] menor unidade de transferência de informação.

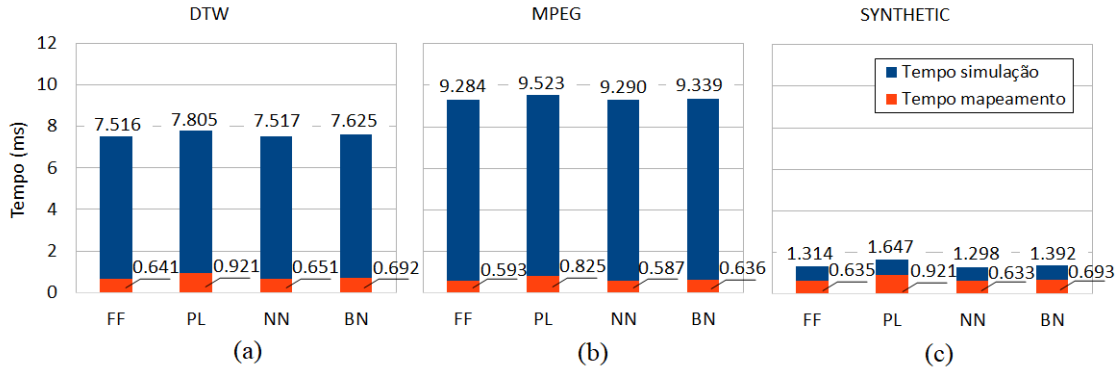


Figura 2. Tempo de mapeamento e execução para os casos de teste 1, 2 e 3.

A Figura 3 apresenta o total de *flits* transmitidos nos canais de comunicação do MPSoC para os casos de testes 1, 2 e 3. A aplicação DTW possui a maior média de *flits* transmitidos entre as tarefas (9.961 *flits*), seguido de MPEG (3.680 *flits*) e de SYNTHETIC (1.175 *flits*). Esses valores estão coerentes com o fluxo de dados apresentado no grafo das aplicações (Figura 1). A heurística BN apresentou as melhores médias de resultados em relação a OC (4.152 *flits*), seguido de NN (4.501 *flits*), PL (4.923 *flits*) e FF (6.179 *flits*). Analisando o ganho em porcentagem, a heurística BN foi 32,80% melhor que a heurística FF, seguido de NN com 27,14% e PL com 20,32%.

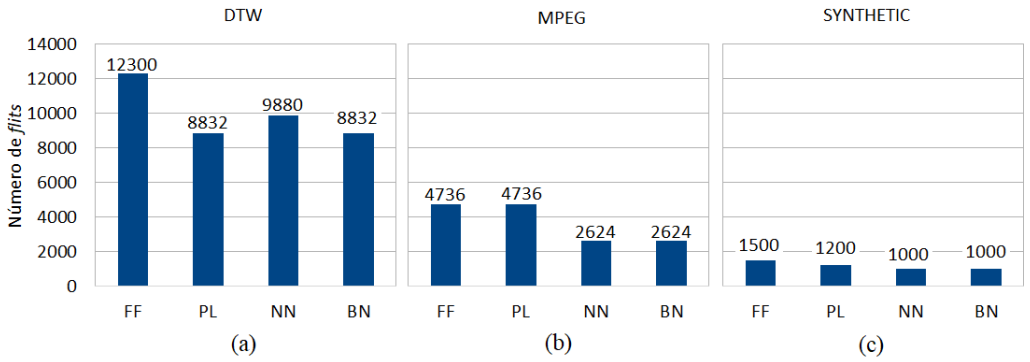


Figura 3. Total de *flits* transmitidos nos casos de teste 1, 2 e 3.

A Figura 4 apresenta o TM e o TE para os casos de teste 4 e 5, que possuem mais de uma aplicação mapeada no MPSoC. O teste 4 (DTW + MPEG) obteve TE médio de 10,12 ms e o teste 5 (DTW + MPEG + SYNTHETIC) obteve TE médio de 10,24 ms. Considerando que: (*i*) após mapeadas, as tarefas executam em paralelo no MPSoC; (*ii*) a aplicação MPEG é a mais lenta das aplicações; e (*iii*) faz parte dos casos de teste 4 e 5; é natural que ambos tenham aproximadamente o mesmo TE. Em relação ao TM, é possível observar uma diferença entre os casos de teste 4 (TM médio de 1,41 ms) e 5 (TM médio de 2,20 ms) devido ao fato de que mais aplicações foram mapeadas no caso de teste 5.

A Figura 5 apresenta o total de *flits* transmitidos nos canais de comunicação do MPSoC para os casos de teste 4 e 5. O teste 5 obteve um fluxo de dados maior pelo fato de ter uma aplicação a mais que o teste 4. É possível notar também que, para os casos de teste com múltiplas aplicações, os resultados evidenciam como cada heurística

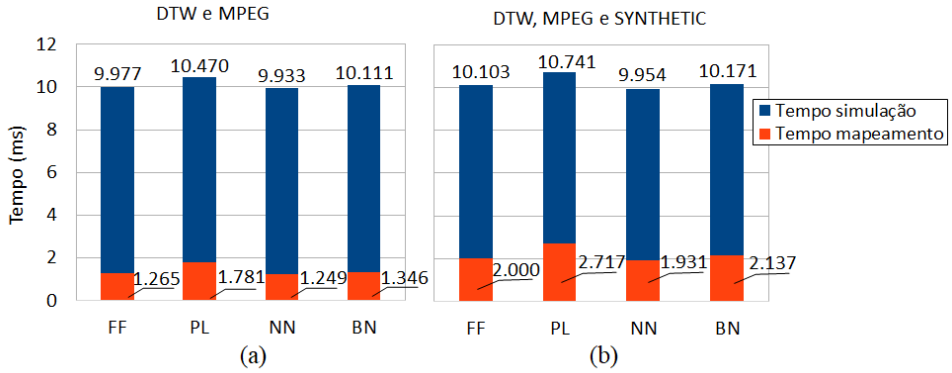


Figura 4. Tempo de mapeamento e execução para os casos de teste 4 e 5.

de mapeamento influência na OC. A heurística BN obteve a melhor média OC para os dois casos de teste, com 11.856 *flits*, seguido de NN com 13.054 *flits*, PL com 15.258 *flits* e FF com 17.536 *flits*. Analisando a diferença em porcentagem, a heurística BN obteve OC até 32,39% menor do que FF, seguida de NN (-25,55%) e PL (-19,99%).

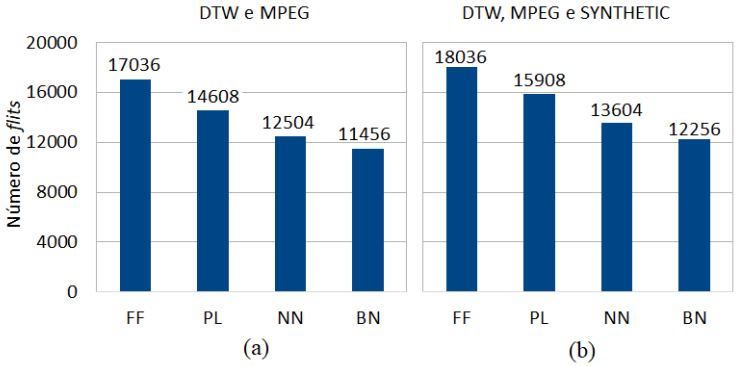


Figura 5. Total de *flits* transmitidos nos casos de teste 4 e 5.

5. Considerações Finais

O desempenho e o consumo de energia de MPSoCs dependem das características de hardware e do software, o que inclui o mapeamento de tarefas adotado. Os resultados apresentados neste artigo mostraram que a heurística BN ocupa 32% menos os canais de comunicação do que as demais heurísticas investigadas, já que possui um algoritmo mais otimizado. Em relação ao desempenho das heurísticas, foi possível identificar que as heurísticas FF e NN possuem resultados de tempo de mapeamento e de execução muito semelhantes e menores do que as heurísticas mais elaboradas (PL e BN). Como trabalhos futuros pretende-se investigar outras heurísticas além de buscar propor uma nova alternativa baseada nas características das heurísticas avaliadas.

Referências

Bohnenstiehl, B. et al. (2016). A 5.8 pJ/Op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*, pages 1–2, Honolulu, HI, USA. IEEE.

Carara, E. et al. (2009). HeMPS-a framework for NoC-based MPSoC generation. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 1345–1348, Taipei, Taiwan. IEEE.

Carvalho, E. et al. (2010). Dynamic task mapping for MPSoCs. *IEEE Design & Test of Computers*, 27(5):26–35.

De Dinechin, B. D. et al. (2014). Time-critical computing on a single-chip massively parallel processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, Dresden, Germany. IEEE.

Ding, J.-H. et al. (2014). An efficient and comprehensive scheduler on Asymmetric Multicore Architecture systems. *Journal of Systems Architecture*, 60(3):305–314.

Fattah, M. et al. (2014). Adjustable contiguity of run-time task allocation in networked many-core systems. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 349–354, Singapore. IEEE.

Garey, M. R. and Johnson, D. S. (1979). A Guide to the Theory of NP-Completeness. *WH Freemann, New York*, 70.

Huang, L.-T. et al. (2015). Wena: Deterministic run-time task mapping for performance improvement in many-core embedded systems. *IEEE Embedded Systems Letters*, 7(4):93–96.

Mandelli, M. G. (2011). Mapeamento dinâmico de aplicações para MPSOCS homogêneos. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul.

Mellanox, T. (2015). TILE-Gx72 Processor. Product Brief Description.

Mendis, H., Indrusiak, L., and Audsley, N. (2015). Bio-inspired distributed task remapping for multiple video stream decoding on homogeneous NoCs. In *Embedded Systems For Real-time Multimedia (ESTIMedia), 2015 13th IEEE Symposium on*, pages 1–10, Amsterdam, Netherlands. IEEE.

Ng, J. et al. (2015). Defrag: Defragmentation for efficient runtime resource allocation in noc-based many-core systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 345–352, Hong Kong, China. IEEE.

Ost, L. et al. (2013). Power-aware dynamic mapping heuristics for NoC-based MPSoCs using a unified model-based approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(3):75.

Quan, W. and Pimentel, A. (2015). A hybrid task mapping algorithm for heterogeneous mpsocs. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(1):14.

Seidipiri, R. et al. (2016). RASMAP: An efficient heuristic application mapping algorithm for network-on-chips. In *Information and Knowledge Technology (IKT), 2016 Eighth International Conference on*, pages 149–155, Hamedan, Iran. IEEE.

Singh, A. K. et al. (2016). Resource and throughput aware execution trace analysis for efficient run-time mapping on MPSoCs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(1):72–85.

Lista de Autores