A Software Based *Many-Core* Architecture Simulator

E. C. Pedrino
UFSCar, S. Carlos, SP
emerson@ufscar.br

T. B. Almeida
UFSCar, S. Carlos, SP
tiago_almeida0297@hotmail.com

M. C. Nicoletti
UFSCar & UNIFACCAMP-SP
carmo@ufscar.br

***Abstract* – As technology continuously advances, engineers are constantly faced with challenges that require numerous computational designs and implementations that, usually, go beyond practical feasibility, considering the available resources at hand. An area that might be considered for dealing with these problems relates to the use of many-core architectures for parallel processing. This type of architecture can be extremely efficient for intensive computational tasks and has the power to operate with low energy and low clock frequencies; however scalability issues attached to the process can significantly affect its design. This paper presents the technicalities involved in developing a scalable many-core software-based simulator named *SImulator for Many-Cores* (SIMC) that includes features such as package routing and efficient inter-process communication. It is intended as a project goal that SIMC becomes a useful software package that allows students with interests in simulating many-core based hardware projects as software systems. It is also intended that by practicing with SIMC on a diverse set of problems, students can acquire experience in analyzing metrics, such as speed and latency, among others that are commonly used in this sort of scenario. The type of practice provided by SIMC promotes a way of fixing the several hardware related concepts involved as well as to enlarge and refine student´s skills in programming. For the case study described in this paper, the validation of SIMC has been carried out by means of solving a relatively trivial problem i.e., that of the execution of simple morphological filters, where the allocation of tasks can be optimized for improving either the execution speed or latency. SIMC allows a direct comparison of values of both metrics, as well as a quantitative evaluation of the implemented network as a whole.**

***Index Terms* – Software simulator, many-core, digital image processing, parallel processing.**

I. Introdução e Contextualização

Em cenários tecnológicos atuais o alto volume de novos problemas que têm surgido, com complexidades cada vez mais elevadas para serem resolvidos, vêm estimulando a pesquisa, investigação e desenvolvimento de sistemas computacionais mais robustos, flexíveis e, particularmente, com alto desempenho [1].

Aplicações em diversas áreas de conhecimento que exigem alto desempenho computacional, tais como as relacionadas a mapeamentos genéticos [2], biologia computacional [3], detecção de anomalias em arquiteturas de alto desempenho [4], pesquisa médica [5], pesquisa espacial [6], processamento digital de imagens [7] e diversas outras em geral exigem sistemas computacionais de alto desempenho, seguros quanto a falhas, de baixo consumo energético e de tamanho reduzido [12-13]. Em vários cenários científicos e/ou industriais, a ênfase em maior poder de processamento computacional se deve, principalmente, aos tratamentos que os imensos volumes de dados requerem, bem como ao volume de cálculos a ser realizado.

Particularmente, muitas aplicações que envolvem a área de processamento digital de imagens requerem operações complexas em imagens ou vídeos de alta definição. Para a realização de tais operações muitas vezes é necessário o uso de arquiteturas não convencionais de processamento, com o objetivo de garantir que tais operações possam ser realizadas em tempo real. A aplicação de filtros morfológicos em processamento de imagens em tempo real é um tema bastante discutido com inúmeros trabalhos que buscam estratégias para otimizar a geração e o processamento deste tipo de filtro [8,10,11].

Como será evidenciado ao longo deste artigo, para a concepção de arquiteturas de sistemas computacionais de alto desempenho, a utilização de arquiteturas *many-core* para processamento rápido e eficiente tem se mostrado bastante atrativa [12-13].

Em um contexto computacional, entende-se por processador *many-core* (muitos núcleos) um processador que tem dezenas ou centenas de *cores* (núcleos) de processamento no interior de um único *chip*. Tais *cores* que são conectados entre si por meio de uma rede de comunicação identificada como *intra-chip* (Network-on-Chip (NoC)). *Cores* têm a funcionalidade de dividirem e executarem a carga de trabalho entre si, caracterizando assim o ambiente de

trabalho multitarefa. Um cenário de comunicação de dados *intra-chip* requer que as aplicações envolvidas possam ser distribuídas de forma balanceada entre os diversos *cores* ativos de um chip *many-core*, favorecendo assim uma maior eficiência energética e/ou um melhor desempenho de processamento [9,12,13].

O projeto de arquiteturas de sistemas *many-core*, entretanto, é complexo e envolve conhecimentos especializados provenientes de diversas áreas de pesquisa [9**,**13]. A implementação deste tipo de sistema, por exemplo, contempla técnicas tais como roteamento eficiente de pacotes entre os processadores envolvidos, bem como a alocação inteligente de tarefas para tais unidades de processamento (*cores*). Arquiteturas *many-core* estão sendo utilizadas como uma alternativa para solucionar problemas de engenharia que envolvem computação de alto desempenho. Na literatura podem ser encontrados diversos trabalhos que usam tais conceitos nos mais diversos cenários; uma breve abordagem de alguns deles é feita na sequência.

No trabalho descrito em [12] são exploradas malhas reconfiguráveis em FPGA (*Field-Programmable Gate Array*) para implementação de um sistema *many-core*. Para validar o sistema os autores de tal trabalho realizaram testes envolvendo a execução do sistema em quatro domínios de aplicação distintos: (1) operações aritméticas, (2) ordenação, (3) grafos e (4) imagens. A arquitetura *many-core* proposta pelos autores permitiu executar de forma eficiente os quatro *benchmark*s considerados. Resultados obtidos em dois cenários distintos, *many-core* e *single-core*, evidenciaram a superioridade, em termos de eficiência e desempenho, do primeiro cenário.

O trabalho apresentado em [13] propõe um acelerador que combina dois algoritmos de ordenação de números inteiros com representação de 32 bits; o acelerador pode ser customizado pelo projetista para a criação de arquiteturas *many-core* heterogêneas. Os experimentos realizados mostraram que o hardware proposto teve um desempenho superior quando comparado aos resultados obtidos por um processador Intel Core i7-4770 na realização da mesma tarefa de ordenação.

Ainda tendo o foco em pesquisas associadas ao uso de arquiteturas *many-core*, vários trabalhos sobre NoC discorrem sobre formas de conectar e rotear com eficiência pacotes em uma determinada rede de processamento. Sistemas *many-core* geralmente são organizados em um *mesh* 2D (representado por uma matriz de *cores* 2D), em que o roteamento XY, dentre os muitos possíveis, é um dos mais utilizados. Em [14,15] são discutidos tais roteamentos e apresentadas métricas de medida de desempenho de sistemas *many-core*, algumas delas utilizadas neste trabalho (ver Seção 2.B).

É importante lembrar que a alocação eficiente de tarefas em sistemas *many-core* é um tema de pesquisa bastante amplo, como evidenciam os vários trabalhos na área [12,15,16,17]. A alocação de tarefas pode ser realizada de maneira a satisfazer um conjunto de possíveis objetivos, como será visto na Seção IV.

No trabalho descrito em [9] é explorada a alocação de tarefas visando diminuir a distância de transferência de pacotes, levando em consideração a taxa de tolerância a falhas (*throughput*), uma característica de sistemas *many-core* que permite que uma tarefa realizada por um *core* possa ser transferida para outro, em caso de falha.

A referência [11] descreve uma aplicação que envolve o uso de filtros morfológicos em imagens, em um ambiente de arquitetura *many-core* e que tem por objetivo otimizar, simultaneamente, a qualidade de execução desses filtros e de seus respectivos tempos de processamento.

O texto em [18] apresenta um algoritmo para a alocação de tarefas em *cores*, de modo a maximizar a eficiência energética do NoC. Os resultados dos experimentos realizados pelos autores evidenciaram que o algoritmo proposto é rápido, robusto e energeticamente eficiente, quando comparado a uma solução *ad-hoc*.

Os trabalhos brevemente comentados anteriormente utilizam diversas técnicas para realizar a alocação de tarefas. O uso de algoritmos genéticos é uma das técnicas mais utilizadas para implementar a alocação inteligente, como feito em [10,18-19].

A breve contextualização de trabalhos relacionados a arquiteturas *many-core* feita nos parágrafos anteriores já é uma evidência da importância e abrangência da pesquisa na área. Pode-se perceber nos trabalhos que há uma forte tendência em direcionar esforços, tanto na modelagem de arquiteturas *many-core* e NoC, quanto no estudo de alternativas para tornar tais arquiteturas viáveis em um número maior de aplicações que envolvam processamentos mais complexos de imagens.

Atualmente, o número crescente de problemas a serem resolvidos que envolvem um grande volume de dados tem provocado o aumento de cenários que favorecem o uso de arquiteturas *many-core*. Com isso, o número de restrições relacionadas a desempenho e consumo de energia para a resolução desses problemas também aumenta. Em muitos casos não é possível fazer uso de um computador robusto com GPU e, portanto, alternativas mais

eficientes devem ser cogitadas. Embora existam simuladores disponíveis online, tais como SIMICS (https://www.intel.com/content/www/us/en/developer/articles/technical/simics-technology.html), SNIPER(https://snipersim.org//w/The_Sniper_Multi-Core_Simulator), GEM5 (gem5.org), o uso de tais simuladores demanda investimento no aprendizado por parte do usuário que pode ser demorado.

As considerações levantadas nessa seção reforçam a relevância do projeto e desenvolvimento de um simulador de sistemas *many-core* em software, capaz de simular a alocação de tarefas em suas unidades virtuais de processamento. Tal prática vai de encontro às necessidades educacionais quando do aprendizado de conceitos relacionados às arquiteturas *many-core*, em disciplinas de cursos universitários e técnicos voltados à computação de alto desempenho.

Entre outros benefícios, além dos educacionais já mencionados, o sistema SIMC proposto e apresentado neste artigo pode também ser de grande utilidade para a comunidade científica, servindo como uma ferramenta útil para testes, coleta e análise de resultados. Também pode servir à comunidade desenvolvedora como uma ferramenta de prototipação que permite testar o que foi desenvolvido, antes de sintetizar o sistema de hardware.

Como comentado anteriormente, neste trabalho é proposto e descrito um simulador de arquitetura *many-core* (SIMC) em software. Sua validação é feita por meio de um estudo de caso de seu uso no processamento de filtros morfológicos. Para tanto, foram utilizadas operações de filtragem genéricas, sem efeito prático, feitas com o objetivo único de validar o simulador implementado.

II. Materiais e Métodos

A. Ambiente de Pesquisa e Preparação

No desenvolvimento do simulador foi utilizado um computador pessoal com as seguintes configurações: processador Intel Core i3-40300 CPU 1.90GHz, 4GB de memória RAM e sistema operacional Ubuntu 16 LTS de 64 bits. Para a programação do sistema foi utilizada a linguagem Python. Esta linguagem foi escolhida pela facilidade que oferece para trabalhar com módulos de MPI (*Message Passing Interface*) e *OpenCV*, que foram utilizados na implementação do sistema SIMC e, também, pela maior familiaridade dos autores com ela.

B. Métricas Utilizadas para Validação e Análise

Para a validação do SIMC foram utilizadas diversas métricas de desempenho medidas durante a execução do simulador no estudo de caso considerado. Os trabalhos descritos em [16,17] foram utilizados como referência para a escolha das métricas Elas são:

1. *Tempo de Processamento* (T_{proc}): Tempo total utilizado para a execução de todas as operações na arquitetura do SIMC atribuídas. O seu cálculo utiliza o tempo de início de processamento (T_{inicio}) e o tempo de fim de processamento (T_{fim}) e é dado pela expressão (1).

$$T_{proc} = T_{inicio} - T_{fim} \quad (1)$$

2. *Taxa de Transferência Total* (Q_{transf}): valor que quantifica o número $Q_{recebidos}$ de pacotes que foram recebidos pelos seus destinatários no período de tempo do processamento (T_{proc}) e dada pela expressão (2).

$$Q_{transf} = Q_{recebidos}/T_{proc} \quad (2)$$

3. *Taxa Média de Transferência por Nó* (Q_{no}): valor que mede a Taxa de Transferência Total (Q_{transf}) dividida pelo número de nós (NN) do simulador e dada pela expressão (3).

$$Q_{no} = Q_{transf}/NN \quad (3)$$

4. *Média da Latência de Pacotes* (M_{Lat}): valor que fornece a média do período de tempo (Lat) decorrido desde o tempo de envio do pacote T_{envio} até o tempo de seu recebimento $T_{recebimento}$ pelo destinatário, calculada para cada pacote e dada pela expressão (4), em que Lat_i (i=1,..., NoPacotes) representa a latência do pacote i.

$$\begin{aligned} Lat &= T_{envio} - T_{recebimento} \\ M_{Lat} &= (Lat_1+Lat_2+..+Lat_{NoPacotes})/NoPacotes \end{aligned} \quad (4)$$

5. *Média do Atraso de Pacotes* (M_{Atraso}): valor que fornece a média do valor Atraso associado a cada pacote, que representa a diferença do tempo de latência (Lat) de um pacote com relação ao tempo de latência do pacote que chegou ao destinatário em menor tempo (i.e. em tempo Lat_{min}). Os cálculos de Atraso e M_{Atraso} estão mostrados nas expressões em (5), em que $Atraso_i$ representa o atraso do pacote i.

$$\begin{aligned} Atraso &= Lat - Lat_{min} \\ M_{Atraso} &= A/NoPacotes, \text{ em que} \\ A &= (Atraso_1 + Atraso_2 + ... + Atraso_{NoPacotes}) \end{aligned} \quad (5)$$

5. *Eficiência* (E): valor que representa a porcentagem de pacotes que chegaram ao destinatário ($Q_{recebidos}$) em relação ao total de pacotes gerados ($Q_{gerados}$) pelo sistema e mostrado na expressão (6).

$$E = (Q_{recebidos} \times 100)/Q_{gerados} \quad (6)$$

6. *Taxa de Perda de Pacotes* (P): valor que representa a porcentagem de pacotes que foram perdidos ($Q_{perdidos}$) em relação aos pacotes gerados ($Q_{gerados}$) no sistema, que é calculado por (7). É importante informar que quando um pacote é enviado de um nó a outro nó, e o buffer associado ao nó destino estiver cheio, o pacote é considerado perdido.

$$E = (Q_{perdidos} \times 100)/Q_{gerados} \quad (7)$$

III. Planejamento e Desenvolvimento do Simulator SIMC

Para o projeto e desenvolvimento do SIMC, devido à complexidade inerente associada à proposta e ao desenvolvimento em software de simuladores *many-core*, foi feito um extenso e cuidadoso planejamento, com um acompanhamento paralelo de especificações bem detalhadas de módulos envolvidos. Na sequência cada uma das próximas subseções aborda aspectos do sistema SIMC.

A. Componentes e Arquitetura do Sistema SIMC

O planejamento do projeto e desenvolvimento do SIMC foi iniciado com a definição da arquitetura do sistema, abordando-a como um conjunto de módulos agregados. Cada módulo foi planejado com o objetivo específico de implementar uma funcionalidade do simulador. Os módulos especificados foram nomeados como: (1) Módulo Grafo de Inicialização, (2) Módulo de Gerenciamento de Pacotes, (3) Módulo de Roteamento de Pacotes e (4) Módulo de Processamento.

Como se trata de uma simulação de arquitetura em software, cada unidade de processamento do simulador é representada por um processo do sistema operacional utilizado.

Para a comunicação entre processos foi utilizado o *Message Passing Inferface* (MPI), que é um padrão para comunicação de dados em computação paralela, em que processos de comunicação estão organizados de forma a simular uma arquitetura de *cores* distribuída em uma configuração matricial. Assim, cada *core* só pode se comunicar com *cores* que estejam imediatamente acima, abaixo, à direita ou à esquerda de sua posição na matriz.

A Fig. 1 exibe um diagrama geral do sistema computacional SIMC que implementa em software um simulador de arquitetura *many-core*. O diagrama exemplifica como cada unidade de processamento (*core*) se conecta com seus pares, com as entradas e com a saída, configurando assim um *mesh* de *cores* de processamento morfológico. C*ores* têm seus fluxos de execução representados por um grafo de inicialização, que leva em consideração a aplicação (no estudo de caso considerado, filtros morfológicos), em que cada *core*, associado a um nó do grafo, representa uma operação morfológica e as arestas do grafo representam conexões entre *cores*.

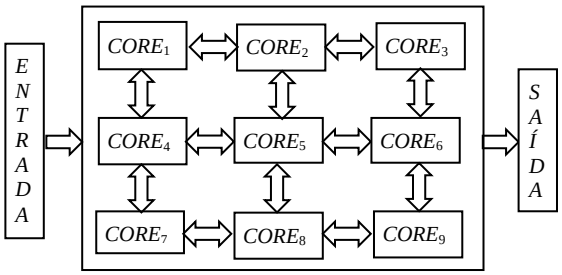

Fig. 1. Diagrama geral do sistema SIMC.

A inicialização do SIMC é feita via processamento do grafo de inicialização. Assim que o SIMC é inicializado, o grafo de inicialização é processado por meio da atribuição a cada *core* de uma tarefa a ser realizada no *mesh* de processamento. A Fig. 2 mostra a estrutura interna de um *cor*e, que é definida por meio de quatro módulos cujas respectivas funcionalidades compõem o processo de controle local em um *core*. Deve-se observar ainda que esse processo ocorre de forma cíclica, até que o processamento total do sistema esteja completo.

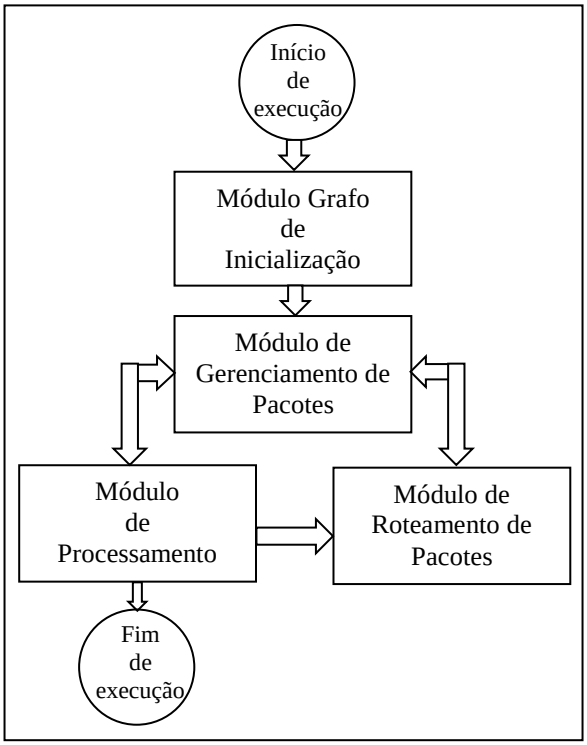

Fig. 2. *Core* – fluxo de execução do programa principal.

O módulo inicial (Módulo Grafo de Inicialização) é responsável pelo carregamento do grafo de inicialização. Tal grafo determina como o *mesh* irá funcionar, tanto com relação às tarefas que cada *core* deve realizar, quanto com relação ao fluxo de pacotes. O posicionamento das tarefas é feito de forma estática pelo usuário.

O Módulo de Gerenciamento de Pacotes é executado na sequência e é responsável por avaliar cada pacote que chega ao *core*, decidindo se tal pacote será enviado para o Módulo de Processamento ou para o Módulo de Roteamento. O Módulo de Processamento é responsável por realizar a tarefa específica alocada ao *core* contida no pacote que chega a ele. No estudo de caso tratado tal tarefa específica, por exemplo, pode ser uma operação de erosão a ser realizada sobre a imagem recebida.

O Módulo de Roteamento, por sua vez, é responsável por enviar o pacote ao nó de destino de maneira correta, de acordo com o algoritmo de roteamento implementado. Os quatro módulos são apresentados com mais detalhes nas próximas quatro subseções.

B. Módulo Grafo de Inicialização

Como comentado anteriormente, todo *mesh* é gerenciado por um grafo de inicialização. Este grafo é responsável por fornecer instruções específicas a cada *core*. A cada nó do grafo está associado um vetor que contém um conjunto de informações necessárias para a realização da tarefa alocada ao *core*. As informações contidas no vetor unidimensional com 5 posições, associado ao nó são descritas a seguir, por meio de um exemplo.

A Tabela 1, que pode ser considerada um recorte dos operadores apresentados em [11], mostra um exemplo contendo a descrição de apenas 5 operações morfológicas e seus Ids associados que poderiam ser usados para construir o vetor de informações associado ao *core*.

TABELA I
EXEMPLO DE TABELA DE OPERAÇÕES (RECORTE DO CONTEÚDO DA TABELA APRESENTADA EM [6]).

Id	Operação	Descrição
1	NOP	Nenhuma operação.
2	SUB	Subtração aritmética.
4	ERO_S_3	Erosão com elemento estruturante quadrado de dimensão 3 × 3.
10	DIL_S_3	Dilatação com elemento estruturante quadrado de dimensão 3 × 3.
28	OR	Operação lógica OR.

1. *Posição 1 do vetor:* contém o Id associado à uma operação, dentre aquelas mostradas na Tabela 1, a ser executada pelo *core*.

2. *Posição 2 do vetor:* contém o código (-1) referente à imagem a ser utilizada como primeiro argumento da operação indicada na posição 1 do vetor.

3. *Posição 3 do vetor:* contém o código referente à saída de um determinado nó que será utilizada como segundo argumento da operação do nó corrente (indicada na posição 1 do vetor).

4. Posição 4 do vetor: contém o código referente ao *core* que deverá processar a operação indicada na posição 1 do vetor.

5. Posição 5 do vetor: contém o código referente ao *core* que deverá receber o resultado da operação indicada na posição 1 do vetor.

O fluxo geral de operação no *mesh* pode ser brevemente descrito como segue, considerando o processamento de uma dada imagem X, conforme mostra a Fig. 3. O grafo de inicialização é responsável por inicializar os vetores associados a cada *core* do *mesh*, em que a imagem X é o segundo elemento do vetor. Além disso, o grafo informa se o *core* em questão deve utilizar a imagem X ou, então, utilizar a imagem produzida por algum outro *core*. O grafo também informa para qual *core* a saída do *core* em questão deverá ser enviada. Na figura o nó C está mostrado com mais detalhes, por meio do vetor a ele associado.

A primeira posição do vetor associado ao *core* que representa o nó C armazena o valor 1, indicando que o *core* associado ao nó C deve realizar a operação identificada por 1 (NOP – ver Tabela I) envolvendo a imagem X e o resultado do *core* associado ao nó B como argumentos. Como indicado no vetor associado ao *core* que representa C, o seu resultado deve ser enviado para o *core* associado ao nó D. Pode-se deduzir pelo grafo que o *core* de número 2 está associado a B e o de número 4 a D, sendo este último o *core* final, que finaliza o processamento com a produção da imagem de saída.

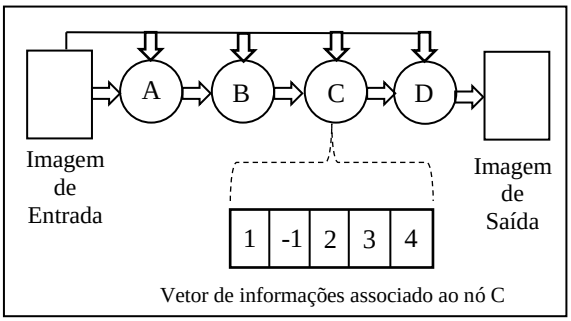

Fig. 3. Exemplo de uso do Grafo de Inicialização.

C. *Módulo de Gerenciamento de Pacotes*

Como comentado anteriormente, um *core* é capaz de obter, processar e enviar pacotes. A representação alto nível utilizada para pacote contém as informações sobre o *core* que o gerou, o *core* ao qual o pacote se destina e o dado (imagem) que ele produziu.

No estudo de caso em questão cada *core* espera receber duas imagens de entrada. Eventualmente, as duas imagens podem não chegar a um determinado *core* ao mesmo tempo, o que faz com que ele fique ocioso. Após o recebimento de ambas as imagens, o resultado da operação associada ao *core* é enviado ao *core* destino pelo Módulo de Roteamento de Pacotes.

D. *Módulo de Roteamento de Pacotes*

O Módulo de Roteamento de Pacotes tem como objetivo fazer com que cada *core* envie pacotes para o *core* vizinho, seguindo o algoritmo de roteamento implementado. O Módulo recebe o dado a ser enviado bem como a identificação do *core* destino, e cria um pacote para envio. Recebe também como parâmetro o algoritmo de roteamento que deve utilizar.

O algoritmo padrão do sistema SIMC é o Roteamento XY [15]. O algoritmo de roteamento conhecido como XY é largamente usado como algoritmo para roteamento em ambiente NoC e é adequado para topologias de redes regulares e irregulares. O algoritmo é determinístico, estático e não provoca *deadlocks* ou *livelocks* [20]. O XY usualmente segue o caminho mais curto e o único caminho determinado para o pacote. Pode ser confirmado na literatura que muitos trabalhos acadêmicos usam esse algoritmo implementado em topologia 2D, com o objetivo de aumentar a taxa de transferência e reduzir a latência do sistema.

E. *Módulo de Processamento*

O Módulo de Processamento é responsável por fazer o *core* realizar a tarefa para a qual foi alocado. Este módulo implementa as várias operações morfológicas, indexadas e definidas na Tabela 1. Como discutido anteriormente, um *core* tem seu funcionamento determinado pelo Módulo de Inicialização.

A Fig. 4 exemplifica alguns dos grafos utilizados para testes. Os números em cada nó demonstram em qual unidade de processamento da matriz ele será alocado. A numeração das unidades é feita tomando-se como elemento 0 o superior esquerdo da matriz e incrementando a contagem da esquerda para a direita e de cima para baixo.

Para o estudo de caso escolhido neste trabalho, foram utilizadas operações de tratamento de imagens. Foram utilizados 30 operadores, como apresentados e discutidos em [11]. Entre os operadores usados estão aqueles relacionados a filtros morfológicos, operadores lógicos e operadores aritméticos para imagens.

Considere uma situação em que dois *cores*, Y e Z, estão conectados em série no *mesh*. Com base nas informações disponibilizadas na tabela, se o Módulo de Inicialização alocou ao *core* Y a operação 2 e ao *core* Z a operação 4, levando em consideração a arquitetura do *mesh*, o *core* Y irá realizar a operação SUB entre as suas imagens de entrada e o *core* Z irá realizar a operação de erosão aplicada ao resultado produzido por Y por meio de um elemento estruturante quadrado 3×3 (ver ERO_S_3) na Tabela I. Uma vez feito isso o Módulo de Processamento envia o resultado para o Módulo de Roteamento, que o direcionará para o *core destino* para continuação do processamento ou, então, para a sua finalização, caso o *core destino* seja o *core final* do processamento.

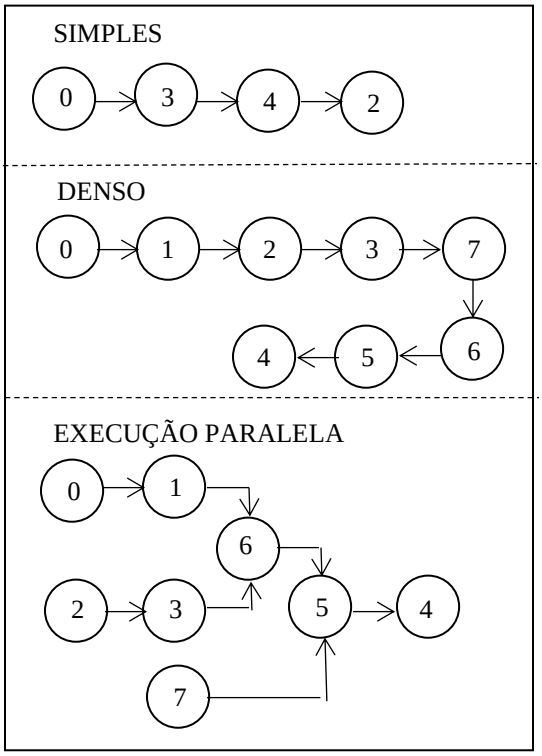

Fig. 4 Exemplos de grafos simples, denso e de execução-paralela.

IV. Resultados e Discussão

A. *Estudo de Caso*

Para validar o sistema SIMC foram usadas, como base de testes, *meshes* com dimensões 3×3, 4×4 e 5×5, contendo 9, 16 e 25 *cores*, respectivamente.

Foram selecionadas três configurações de grafos de inicialização para a validação do funcionamento do SIMC.

A primeira representada por grafos caracterizados como simples, com pouca carga de tarefas e definidos por apenas 4 nós, cujos *cores* associados são executados sequencialmente.

A segunda, representada por grafos denominados densos, cujo número de nós é menor que metade da dimensão do *mesh* escolhido e cujos *cores* devem ser executados sequencialmente.

A terceira configuração é caracterizada por um grafo cuja representação permite que os processos associados aos respectivos *cores* que representam possam ser executados de maneira paralela. Nessa configuração o número de nós do grafo também deve ser menor que metade da dimensão do *mesh* escolhido.

Para cada uma das configurações de grafo foram consideradas duas versões: a primeira, na qual os nós estão alocados de forma a promover uma maior velocidade de processamento e a segunda, em que os nós estão alocados de forma a promover uma maior tolerância a falhas.

Os grafos foram gerados de forma manual pelos autores e as operações consideradas não produzem qualquer resultado prático. Somente operações genéricas foram utilizadas para a validação do sistema computacional SIMC.

Como mostrado anteriormente, a Fig. 4 ilustra alguns dos grafos utilizados para a validação do estudo de caso. Na figura o número mostrado em cada nó indica a qual *core* do *mesh* ele será associado. A numeração dos *cores* é feita adotando o seguinte procedimento: o *core* na posição superior esquerda é identificado como 0 (zero) e progride da esquerda para a direta e de cima para baixo, até o *mesh* ser finalizado. Para economizar espaço com as figuras dos demais grafos aplicados optou-se por suprimi-las. No total foram utilizados 24 grafos:

- 3 grafos simples para VP (velocidade de processamento),
- 3 grafos simples para TF (tolerância a falhas),
- 3 grafos densos para VP,
- 3 grafos densos para TF,
- 3 grafos de execução paralela para VP,
- 3 grafos de execução paralela para TF,
- 3 grafos correspondentes à versão de execução sequencial da mesma tarefa dos grafos de execução paralela para VP e
- 3 grafos deste mesmo tipo para TF.

B. Resultados Obtidos

As Tabelas II, III e IV mostram na coluna O a métrica que foi usada i.e., VP (velocidade de processamento) ou TF (tolerância a falhas).

A coluna NProc informa o número de *cores* envolvidos, a coluna NOp informa o número de *cores* ativos referentes aos nós do grafo de inicialização. A coluna TProc informa os valores da métrica de tempo total de processamento. A coluna QTransf informa a taxa de transferência total do sistema e a coluna QNó informa a taxa de transferência por nó, considerando as representações em grafos simples (Tabela II), densos (Tabela III) e de execução paralela (Tabela IV) em *meshs* com 9, 16 e 25 *cores*, respectivamente. Os valores relacionados a tempo estão em segundos.

TABELA II
TEMPO DE PROCESSAMENTO E TAXA DE TRANSFERÊNCIA PARA GRAFOS SIMPLES.

O	NProc	NOp	TProc	QTransf	QNó
VP	9	5	11,03	0,36	0,04
VP	16	8	11,56	0,59	0,04
VP	25	13	12,04	0,41	0,02
TF	9	5	13,03	0,61	0,07
TF	16	8	15,03	0,86	0,05
TF	25	13	33,02	0,64	0,02

TABELA III
TEMPO DE PROCESSAMENTO E TAXA DE TRANSFERÊNCIA PARA GRAFOS DENSOS.

O	NProc	NOp	TProc	QTransf	QNó
VP	9	5	10,07	0,49	0,05
VP	16	8	15,05	0,59	0,04
VP	25	13	42,25	0,33	0,01
TF	9	5	30,06	0,36	0,04
TF	16	8	34,08	0,59	0,04
TF	25	13	83,08	0,39	0,02

TABELA IV
TEMPO DE PROCESSAMENTO E TAXA DE TRANSFERÊNCIA PARA GRAFOS DE EXECUÇÂO PARALELA.

O	NProc	NOp	TProc	QTransf	QNó
VP	9	5	11,04	0,54	0,06
VP	16	8	24,16	0,45	0,03
VP	25	13	30,40	0,52	0,02
TF	9	5	14,12	0,56	0,06
TF	16	8	29,03	0,61	0,04
TF	25	13	54,05	0,75	0,03

Na Tabela V a coluna O indica qual métrica está sendo avaliada e o tipo de grafo de inicialização utilizado (P: paralelo) e (S: sequencial). A tabela permite uma comparação entre os valores das métricas VP e TF nos dois cenários, paralelo e sequencial.

O experimento cujos resultados estão na Tabela V foi realizado com o objetivo de coletar dados para evidenciar se de fato há ganho de tempo de processamento quando um processo representado por um dado grafo de inicialização é executado por um dos *meshes* considerados (9, 16 ou 25 *cores*).

TABELA V
COMPARAÇÃO DE VALORES DAS MÉTRICAS VP E TF CONSIDERANDO GRAFOS DE INICIALIZAÇÂO PARALELA E SEQUENCIAL.

O	NProc	NOp	TProc	QTransf	QNó
VP(P)	9	5	8.41	0,46	0,05
VP(P)	16	8	12,32	0,57	0,04
VP(P)	25	13	27,11	0,52	0,02
TF(P)	9	5	12,85	0,46	0,05
TF(P)	16	8	25,12	0,51	0,03
TF(P)	25	13	35,75	0,73	0,03
VP(S)	9	5	11,80	0,50	0,06
VP(S)	16	8	18,37	0,54	0,03
VP(S)	25	13	43,54	0,35	0,01
TF(S)	9	5	34,45	0,40	0,04
TF(S)	16	8	40,06	0,57	0,04
TF(S)	25	13	74,04	0,42	0,02

As Tabelas VI, VII e VIII mostram os valores das métricas de latência média (MLat), de atraso médio (MAtr), de eficiência ou taxa de envio de pacotes (E) e de taxa de perda de pacotes (P), para grafos simples, densos e paralelos, respectivamente.

TABELA VI
LATÊNCIA MÉDIA, ATRASO MÉDIO, EFICIÊNCIA (%) E TAXA DE PERDA (%) PARA GRAFOS SIMPLES.

O	NProc	NOp	MLat	MAtr	E (%)	P(%)
VP	4	4	11,03	0,36	100,0	0,0
VP	4	4	11,56	0,59	100,0	0,0
VP	4	4	12,04	0,41	100,0	0,0
TF	4	4	13,03	0,61	100,0	0,0
TF	4	4	15,03	0,86	100,0	0,0
TF	4	4	33,02	0,64	100,0	0,0

TABELA VII
LATÊNCIA MÉDIA, ATRASO MÉDIO, EFICIÊNCIA (%) E TAXA DE PERDA (%) PARA GRAFOS DENSOS.

O	NProc	NOp	MLat	MAtr	E(%)	P(%)
VP	9	5	2,38	2,35	100,0	0,0
VP	16	8	2,15	1,39	93,3	6,6
VP	25	13	2,95	2,15	90,0	10,0
TF	9	5	7,25	4,76	100,0	0,0
TF	16	8	7,84	3,47	100,0	0,0
TF	25	13	6,95	3,30	93,3	6,6

TABELA VIII
LATÊNCIA MÉDIA, ATRASO MÉDIO, EFICIÊNCIA (%) E TAXA DE PERDA (%) PARA GRAFOS DE EXECUÇÂO PARALELA.

O	NProc	NOp	MLat	MAtr	E(%)	P(%)
VP	9	5	4,40	4,38	100,0	0,0
VP	16	8	4,90	4,14	93,3	6,6
VP	25	13	3,99	3,36	90,0	10,0
TF	9	5	7,33	2,72	100,0	0,0
TF	16	8	5,20	4,78	96,6	3,3
TF	25	13	8,57	4,90	93,3	6,6

C. Análise dos Resultados Obtidos

Segue uma análise dos resultados obtidos nos experimentos realizados cujos resultados estão descritos em sete tabelas mostradas na Seção B.

Analisando os valores de VP e TF nas tabelas II e III, referentes a grafos simples e densos respectivamente, pode ser observado que os valores obtidos para as métricas TProc e QTransf são melhores para VP, quando comparados a seus valores correspondentes para TF. Entretanto, na Tabela IV, quando do grafo de execução paralela, pode ser notada a ocorrência de uma exceção com foco em VP, quando a maior velocidade de processamento acontece para TF, contrário ao esperado. A razão dessa ocorrência se deve ao aumento do número de *cores*, que provocou uma maior flexibilidade de roteamento.

Ainda com relação aos dados da Tabela IV, vale ressaltar que TF teve uma taxa de transferência mais elevada devido à necessidade de serem realizadas mais transmissões para que cada pacote chegasse ao seu destino, uma vez que os *cores* ativos estavam localizados mais distantes uns dos outros.

Os valores mostrados na Tabela V permitem realizar uma comparação de tempo de processamento entre os grafos de inicialização de execução paralela e suas versões de execução sequencial. Como esperado, uma taxa de desempenho maior foi obtida com a paralelização das tarefas. Já os valores obtidos com relação à taxa de transferência não foram conclusivos. Nos parágrafos seguintes são discutidos os resultados mostrados em Tabela VI, Tabela VII e Tabela VIII.

Com relação à latência, ao aumentar o número de *cores*, tanto para VP quanto para VF, os valores de latência se mantiveram relativamente constantes. Quando do uso de grafos simples o valor da eficiência chegou ao máximo (100%). Já para os grafos densos e de execução paralela, que requerem o uso de um número alto de *cores*, verificou-se uma quantidade de atrasos maior que aquela encontrada quando do uso de grafos simples.

Com o aumento do número de nós do grafo pôde ser observada a perda de pacotes. Este problema provavelmente se deve à forma como a comunicação entre os nós foi implementada em MPI. Para o sistema SIMC foi implementada a comunicação bloqueante entre os *cores*. Uma outra possibilidade seria a do uso da comunicação não bloqueante, em que espera-se maximizar a eficiência em todos os casos, cogitada a ser implementada na continuação do trabalho.

V. CONCLUSÃO & CONTINUIDADE

No que diz respeito ao protótipo construído do sistema computacional SIMC e considerando os resultados obtidos com o estudo de caso, pode ser observado que para um tamanho já considerável de rede, o sistema

funciona com a correta operação das principais funcionalidades envolvidas.

Os módulos implementados estão funcionando de maneira adequada, o que permite a execução, no sistema, de diversas configurações de grafo. Apesar do bom funcionamento do SIMC é possível ainda implementar refinamentos na técnica de comunicação utilizada. Para esta versão protótipo do SIMC descrita neste artigo foi utilizada a comunicação bloqueante de dados, que não se mostrou tão adequada quanto esperado, gerando perdas de pacote.

O trabalho descrito neste artigo terá continuidade com o desenvolvimento de uma versão mais refinada do protótipo, no que tange à comunicação, com a implementação de comunicação não bloqueante.

Também, pretende-se que o protótipo já desenvolvido esteja à disposição de universitários, para uso e aprendizado dos conceitos e técnicas computacionais utilizados na simulação de arquiteturas *many-core*, para uso em ambiente de ensino e pesquisa e, particularmente, para incentivar seu uso na solução de problemas outros que envolvendo processamento de imagens.

Não foi objetivo de o trabalho realizado fazer uma simulação mais realística do NoC, mas sim idealizar uma abstração para estudo das métricas apresentadas na Seção 2.B. Uma frente de investigação pretendida como continuidade do trabalho tem por foco o estudo e agregação ao SIMC de métricas relacionadas a consumo energético.

Outro aspecto a ser lembrado é referente à possibilidade do uso de placas dedicadas para estudos/pesquisas envolvendo arquiteturas de sistemas *many-core* em nível de graduação e pós-graduação. No entanto, na prática, existem dificuldades envolvendo tempo e custos associados ao desenvolvimento de projetos com tais placas, particularmente as relacionadas à importação o que, de certa forma, reforçam o uso de simuladores tais como o SIMC.

AGRADECIMENTOS

Os três autores agradecem ao DC-UFSCar pelo apoio recebido. A terceira autora também agradece ao CNPq.

REFERÊNCIAS

[1] I.H. Sarker, "AI-based modeling techniques, applications and research issues towards automation, intelligent and smart systems," *SN Computer Science*, vol. 3, no. 158, 2022, https://doi.org/10.1007/s42979-022-01043-x

[2] T. Schupbach, I. Xenarios, S. Bergmann and K. Kapur, "FastEpostasos: a high performance computing solution for quantitative trait epistasis," *Bioinformatics*, vol. 26, no. 11, pp. 1468-1469, 2010. https://doi.org/10.1093/bioinformatics/btq147

[3] D. A. Bader, "Computational biology and high-performance computing," *Communications of the ACM*, vol. 47, no. 11, pp. 35-41, 2004.

[4] S. Ghiasvand† and F. M. Ciorba, "Anomaly detection in high performance computers: a vicinity perspective," Proc. 18th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 112-120, 2019. doi:10.1109/ispdc.2019.00024

[5] E. J. Topol, "High-performance medicine: the convergence of human and artificial intelligence," *Nature Medicine*, vol. 25, pp. 44-56, 2019. https://doi.org/10.1038/s41591-018-0300-7

[6] G. W. Schoonderbeek, A. Szomoru, A. W. Gunst, L. Hiemstra, and J. Hargreaves, "UniBoard2, a generic scalable high-performance computing platform for radio astronomy", *Journal of Astronomical Instrumentation*, vol. 8, no. 2, 1950003, 2019. doi: 10.1142/S225117171950003X

[7] R. Gonzales and R. Woods, "Digital Image Processing," 3rd ed. Pearson, 2009. doi: 10.1117/1.3115362

[8] I. Yoda, K. Yamamoto and H. Yamada, "Automatic acquisition of hierarchical mathematical morphology procedures by genetic algorithms," *Image and Vision Computing*, vol. 17, no. 10, pp. 749-760, 1999. doi: 10.1016/s0262-8856(98)00151-6

[9] C. Bonney, P. Campos, N. Dahir and G. Tempesti, "Fault tolerant task mapping on many-core arrays," Proc. 2016 IEEE Symposium Series on Computational Intelligence (SSCI), pp. 1-8, 2016. doi: 10.1109/SSCI.2016.7850174

[10] M. Quintana, R. Poli and E. Claridge, "Morphological algorithm design for binary images using genetic programming," *Genetic Programming and Evolvable Machines*, vol. 7, no. 1, pp. 81-102, 2006. doi: 10.1007/s10710-006-7012-3

[11] E. Pedrino, D. Lima and G. Tempesti, "Multiobjective metaheuristic approach for morphological filters on many-core architectures," *Integrated Computer-Aided Engineering*, vol. 26, no. 4, pp. 383-397, 2019. doi: 10.3233/ICA-190607

[12] H. Giefers and M. Platzner, "An FPGA-based reconfigurable mesh many-core," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 2919-2932, 2014. doi: 10.1109/TC.2013.174

[13] R. Kobayashi and K. Kise, "FACE: fast and customizable sorting accelerator for heterogeneous many-core systems," Proc. IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, pp. 49-56, 2015, doi: 10.1109/MCSoC.2015.40

[14] M. Atagoziyev, "Routing algorithms for on chip networks," M.Sc. Thesis, Middle East Technical University, 79 pgs., 2007.

[15] T. Khan, "Performance Analysis of XY Routing Algorithm Using 2-D Mesh (M × N) Topology", M.Sc, Thesis, University of Victoria, 25 pgs., 2017. http://hdl.handle.net/1828/8465

[16] S. Pllana and F. Xhafa (Eds.), "Programming Multicore and Many-core Computing Systems", Wiley Series on Parallel and Distributed Computing, Wiley Publishers, 528 pgs., 2017.

[17] A. Vajda, "Programming Many-Core Chips," Springer, Boston, MA. Springer US, 2011. doi: 10.1007/978-1-4419-9739-5

[18] J. Hu and R. Marculescu, "Energy-aware mapping for tile-based NoC architectures under performance constraints," Proc. ASP-DAC Asia and South Pacific Design Automation Conference, pp. 233-239, 2003 doi: 10.1109/ASPDAC.2003.1195022

[19] T. Lei and S. Kumar, "A two-step genetic algorithm for mapping task graphs to a network on chip architecture," Proc. Euromicro Symposium on Digital System Design, pp. 180-

187, 2003, doi: 10.1109/DSD.2003.1231923

[20] S. D. Chawade, M.A. Gaikwad, M and R.M. Patrikar, "Review of XY routing algorithm for network-on-chip architecture," *International Journal of Computer Applications*, vol. 43, pp. 20-23, 2012.