Proposta de Suporte ao Padrão MPI sobre Infraestrutura de Comunicação de Baixo Nível no *Nanvix*

João Fellipe Uller[1], João Vicente Souto[1], Pedro Henrique Penna[2,3], Márcio Castro[1]

[1] Laboratório de Pesquisa em Sistemas Distribuídos (LaPeSD)
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, Brasil

[2]Laboratoire d'Informatique de Grenoble (LIG)
Université Grenoble Alpes (UGA) – Grenoble, França

[3]Grupo de Arquitetura de Computadores e Processamento Paralelo (CArT)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – Belo Horizonte, Brasil

joao.f.uller@grad.ufsc.br,joao.vicente.souto@grad.ufsc.br,
pedro.penna@sga.pucminas.br,marcio.castro@ufsc.br,cota@pucminas.br

Resumo. *Particularidades arquiteturais e o pouco suporte para lightweight manycores fazem com que o porte de aplicações seja árduo nesses processadores. Este trabalho propõe a implementação de um subconjunto do padrão MPI sobre os mecanismos de comunicação de baixo nível existentes no Nanvix, um SO distribuído com foco nessa classe de processadores, visando aumentar sua programabilidade sem perdas significativas de desempenho ou portabilidade.*

1. Introdução

Ao longo dos anos, a eficiência energética tornou-se fundamental para os projetos computacionais. A crescente necessidade de desempenho, em paralelo à exigência de menor consumo, fez com que a exploração de paralelismo entre *threads* se tornasse a solução mais viável para o uso eficaz do crescente número de transistores por *chip* [Totoni et al. 2012]. Nesse contexto, arquiteturas *multicore* interconectadas por uma *Network-on-Chip* (NoC) têm-se mostrado como um dos principais *designs* para alta eficiência energética [Huang et al. 2013]. Assim, os *lightweight manycores* surgem como uma ótima solução que provê altas taxas de paralelismo aliadas a um baixo consumo energético, tendo o Kalray MPPA-256 como um exemplo dessa classe de processadores.

No entanto, questões arquiteturais, como restrição de memória e topologia da NoC, fazem com que a tarefa de portar aplicações para essas arquiteturas seja não trivial. Por isso, faz-se necessária a disponibilização de ambientes de desenvolvimento que equilibrem melhor desempenho, escalabilidade e programabilidade. Este trabalho propõe a implementação de um subconjunto das funções do *Message Passing Interface* (MPI), visando aproveitar ao máximo as características dessas arquiteturas de maneira simplificada. Alguns trabalhos anteriores propuseram adaptar implementações existentes do MPI para processadores que utilizam NoCs [Totoni et al. 2012]. No entanto, essas soluções não são viáveis para sistemas com memórias restritivas como *lightweight manycores* [Ho et al. 2015], devido a pilhas de *software* excessivas. O objetivo é desenvolver uma solução viável para sistemas tão restritivos e que seja portável entre as diferentes arquiteturas, equilibrando desempenho e programabilidade. Isso será possível através do uso do *Nanvix*[1], um sistema operacional com foco em *lightweight manycores*.

[1]https://github.com/nanvix/

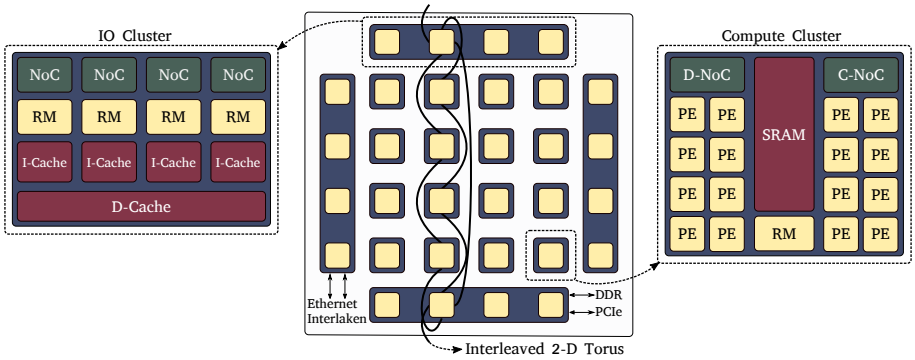

Figura 1. Visão geral simplificada do Kalray MPPA-256 [Penna et al. 2019b].

2. O Processador Kalray MPPA-256 e o *Nanvix*

O Kalray MPPA-256 é uma das arquiteturas suportadas pelo *Nanvix* e o processador no qual serão executados os experimentos do presente trabalho. Na Figura 1 tem-se uma visão geral do Kalray MPPA-256. Ele integra 16 *Clusters* de Computação (CCs) com 16 *Processing Element*s (PEs) e um *Resource Manager* (RM) cada, além de 4 *Clusters* de I/O (IOs), com 4 núcleos cada, para comunicação com periféricos, integrando ao todo 288 núcleos no mesmo *chip*. Cada CC conta com 2 MB de memória compartilhada entre os PEs, e tanto IOs quanto CCs são conectados por duas NoCs, uma para dados (D-NoC) e outra para comandos (C-NoC), para comunicação através de troca de mensagens.

O Nanvix é um Sistema Operacional (SO) de código aberto, compatível com o padrão POSIX, que tem como foco principal atender às necessidades dos *lightweight manycores* buscando o equilíbrio entre desempenho, portabilidade e programabilidade. Esse sistema utiliza uma estrutura de *multikernel*, onde cada *cluster* executa uma instância de um *microkernel* assimétrico, i.e., um núcleo executa exclusivamente o *kernel*, deixando os outros núcleos para propósito geral. Essa instância do *microkernel* é desenvolvida de modo a prover compartilhamento de recursos, serviços mínimos de SO, além de primitivas e abstrações de recursos a nível de *cluster* [Penna et al. 2019a]. Os diversos *clusters* comunicam-se através de trocas de mensagens utilizando abstrações de baixo nível fornecidas pelo módulo de comunicação do *kernel*. As abstrações fornecidas pelo SO que servirão de base para a implementação proposta pelo presente trabalho são: i) *syncs* para a criação de barreiras de sincronização; ii) *mailboxes* para troca de mensagens curtas e de tamanho fixo; iii) *portals* para transferências de dados densas e de tamanho variável.

3. Proposta de Suporte Parcial ao Padrão MPI no *Nanvix*

Apesar dos vários esforços que têm sido empreendidos, o MPI ainda não é um padrão suportado de maneira satisfatória nos *lightweight manycores* existentes, como o Kalray MPPA-256. A proposta do presente trabalho envolve implementar um *subset* de funções do padrão MPI, bem como algumas de suas abstrações como grupos e comunicadores, visando não apenas permitir o uso da biblioteca MPI para o porte de aplicações no *Nanvix*, mas principalmente viabilizar futuras extensões desse suporte e das funcionalidades disponibilizadas. Essa implementação será feita utilizando a infraestrutura de *Inter-Process Communication* (IPC) do *Nanvix* e suas abstrações – *Sync*, *Mailbox* e *Portal* [Souto et al. 2019] – que oferecem uma visão uniforme e padronizada, no nível do *kernel*, dos recursos de comunicação disponíveis nas diferentes arquiteturas suportadas.

Essa implementação coloca-se como uma opção de maior potencialidade ao *multikernel* provido pelo SO, uma vez que dá maior controle ao programador sobre as

comunicações e possui uma pilha de *software* menor, enquanto compensa um esperado *overhead* em relação ao uso direto das abstrações com uma maior programabilidade. Dentre as funções propostas estão: `MPI_Init`, `MPI_Finalize`, `MPI_Comm_create`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Comm_group`, `MPI_Group_free`, `MPI_Send` e `MPI_Recv`, onde a escolha pelas duas últimas se dá pelo fato do *Nanvix* suportar, atualmente, apenas o modo síncrono de comunicação.

No entanto, a implementação de um padrão como o MPI em arquiteturas restritivas como os *lightweight manycores* envolve vários desafios. O principal se refere à limitação de memória disponível em cada *cluster*, que inviabiliza a adaptação de soluções já existentes como OpenMPI[2] ou MPICH[3], e exige uma implementação desde o princípio que seja mínima. Outro desafio se encontra na necessidade de que o SO base ofereça suporte à virtualização e à multiplexação dos recursos de comunicação. Esse suporte é indispensável, uma vez que se espera estabelecer a ideia de múltiplos comunicadores e canais de comunicação de maneira transparente no nível de usuário, o que não acontece na maioria dos SOs desenvolvidos para essa classe de processadores.

O *Nanvix*, apesar de suportar virtualização, ainda não suporta a multiplexação dos canais. Assim, o presente trabalho apresenta uma solução de multiplexação para a abstração *Portal*, aliando a virtualização dos recursos ao endereçamento por portas lógicas, semelhante àquilo que acontece em redes de computadores. A virtualização consiste em armazenar *Portals* virtuais no *kernel* e depois mapeá-los para os recursos expostos pela Camada de Abstração de Hardware (HAL). Para a multiplexação, cada *Portal* virtual é associado a uma porta lógica, criada como um endereço no *microkernel* que está vinculada ao respectivo recurso de *hardware*. A partir daí, as trocas de mensagens são feitas endereçando o *cluster* e a porta lógica de destino, que representa a *thread* à qual se endereça a mensagem. Esse controle da multiplexação é feito totalmente no nível do *kernel*, sendo independente de arquitetura, uma vez que não altera a forma como o envio dos dados é feito pela HAL através da NoC. Um *header* com as informações do endereçamento é anexado no início do *buffer* que será transferido do lado do emissor, e após isso, o receptor é quem se encarrega de realizar o devido encaminhamento das mensagens recebidas pela NoC para as devidas portas lógicas no *kernel*.

4. Resultados Experimentais

Para avaliar o *overhead* decorrente da implementação de um modelo MPI, fez-se uma avaliação do impacto inicial, resultante de se adicionar a multiplexação dos canais de comunicação, observando a vazão obtida em transferências de dados utilizando *Portal*. Para isso, foi utilizada a rotina de comunicação *Ping-pong* [Wickramasinghe and Lumsdaine 2016], existente no *Nanvix*. Foram realizadas 40 iterações para cada cenário obtendo 95% de confiança com desvio padrão de 10%, obtido em relação à média do pior caso de cada iteração. Cada cenário consistiu em utilizar diferentes números de *threads* em cada *cluster*, avaliando a degradação do desempenho à medida que se aumenta a disputa pela utilização do único canal físico multiplexado. No IO, o número de *cores* utilizado foi fixado em 3, enquanto no CC esse número foi sendo aumentado gradativamente. Cada mensagem possuía 1 KB de dados, onde cada *thread* do CC enviava e recebia uma mensagem de alguma *thread* no IO.

A Figura 2 mostra os resultados obtidos. Nela pode-se ver que no caso de apenas uma *thread* em cada *cluster* (i.e., sem multiplexação) a vazão obtida foi próxima à obtida em [Souto et al. 2019], como esperado. No entanto, à medida que a disputa pelo canal au-

[2]https://www.open-mpi.org/

[3]https://www.mpich.org/

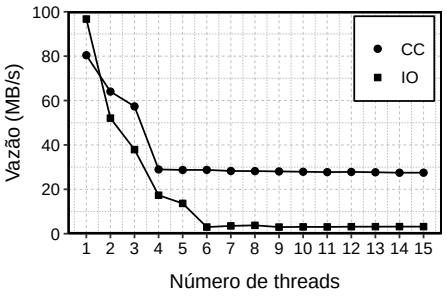

Figura 2. Resultados iniciais de vazão.

menta, a vazão cai consideravelmente, uma vez que perde-se mais tempo reencaminhando as mensagens corretamente no *kernel*, o que não acontece quando não se tem a necessidade de endereçamento. Pode-se ver também que no CC a vazão é maior e se estabiliza antes, mesmo que com um número maior de *threads* que o IO. Isso mostra que, mesmo que se tenha uma maior concorrência pelo canal desse lado, seu paralelismo maior, em detrimento à necessidade do IO de atender às outras portas sequencialmente, faz com que ele tenha um fluxo contínuo de dados e, consequentemente, uma maior vazão.

5. Conclusão

Neste trabalho foi feita a proposta de suporte a um *subset* inicial do MPI sobre a infraestrutura de comunicação de baixo nível do *Nanvix*. No entanto, como o mesmo ainda não suporta a multiplexação dos recursos, foi realizada uma implementação dessa multiplexação para a abstração *Portal*. Foi visto que é possível fazer essa multiplexação obtendo resultados satisfatórios, sendo uma possibilidade aumentar o tamanho dos pacotes para um ganho na vazão. Como trabalhos futuros, tem-se a implementação do suporte à multiplexação de *Mailboxes* e *Syncs*, antes da implementação do MPI propriamente dita.

Referências

Ho, M.-Q., Tourancheau, B., Obrecht, C., Dupont De Dinechin, B., and Reybert, J. (2015). MPI communication on MPPA Many-core NoC: design, modeling and performance issues. In *ParCo 2015*, Edinburgh, United Kingdom.

Huang, L., Wang, Z., Xiao, N., Wang, Y., and Dou, Q. (2013). Adaptive communication mechanism for accelerating mpi functions in noc-based multicore processors. *ACM Trans. Archit. Code Optim.*, 10(3).

Penna, P. H., Souto, J., Lima, D. F., Castro, M., Broquedis, F., Freitas, H. H., and Mehaut, J.-F. (2019a). On the Performance and Isolation of Asymmetric Microkernel Design for Lightweight Manycores. In *SBESC*, pages 1–8, Natal, Brazil.

Penna, P. H., Souza, M., Jr, E. P., Souto, J., Castro, M., Broquedis, F., de Freitas, H. C., and Méhaut, J.-F. (2019b). RMem: An OS service for transparent remote memory access in lightweight manycores. In *MULTIPROG*, pages 1–16, Valencia, Spain.

Souto, J. V., Penna, P. H., and Castro, M. (2019). An inter-cluster communication facility for lightweight manycore processors in the nanvix OS. Trabalho de Conclusão, UFSC.

Totoni, E., Behzad, B., Ghike, S., and Torrellas, J. (2012). Comparing the power and performance of intel's scc to state-of-the-art cpus and gpus. *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 78–87.

Wickramasinghe, U. and Lumsdaine, A. (2016). A survey of methods for collective communication optimization and tuning. *CoRR*, abs/1611.06334.