Paralelismo de Tarefas em Arquiteturas Híbridas Multi-CPU e Multi-GPU

João V. F. Lima[1], **Nicolas Maillard**[1]

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

{joao.lima, nicolas}@inf.ufrgs.br

1. Introdução

Os sistemas de alto desempenho tem passado por uma transição guiada pela limitação de energia e desempenho. A produção de processadores mais velozes atingiu limites técnicos e econômicos que a tornaram inviável. Hoje em dia, arquiteturas híbridas têm sido amplamente utilizadas como uma alternativa eficiente e viável na busca por maior poder de processamento [Asanovic et al. 2009]. Elas possuem várias unidades de processamento (PUs) altamente especializadas tais como processadores gráficos (GPUs) ou processadores heterogêneos (Cell BE). Da mesma forma, as arquiteturas híbridas têm PUs heterogêneos em termos de poder de processamento e modelo de programação. Os algoritmos para processadores *multi-core* usam o modelo *Multi-Instruction Multiple Data* (MIMD), enquanto que algoritmos para GPUs usam o modelo *Single Instruction Multiple Data* (SIMD). Sistemas distribuídos como agregados, que eram compostos de arquiteturas multiprocessadas e em geral homogêneas, apresentam atualmente arquiteturas híbridas com múltiplos níveis de paralelismo.

Dessa forma, em muitos casos, algoritmos amplamente conhecidos precisam ser repensados e reescritos de forma que tenham proveito das novas PUs. Os novos algoritmos podem ser híbridos com diferentes implementações de uma tarefa, otimizadas para cada PU. Um novo desafio nessas arquiteturas é distribuir o trabalho eficientemente a fim de ter proveito de todo o paralelismo disponível e contornar as diferenças em termos de poder de processamento entre as PUs. Outro aspecto importante a ser considerado corresponde ao sobrecusto para decidir onde a tarefas executará, na carga do código a ser executado e na cópia das entradas e saídas.

O roubo de tarefas, ou *Work Stealing*, por exemplo, é um algoritmo de escalonamento descentralizado em que as decisões de escalonamento dependem dos processadores ociosos, que roubam tarefas de outros processadores. Essa estratégia demonstra resultados teoricamente eficientes em arquiteturas de memoria compartilhada; todavia, sua aplicação em arquiteturas híbridas não é amplamente explorado devido aos diversos custos envolvidos na atribuição de tarefas a uma PU. Algumas ferramentas de programação paralela consolidaram-se em arquiteturas *multi-core* tais como Cilk, TBB e OpenMP [Frigo et al. 1998, Reinders 2007, Chapman et al. 2007] e interfaces para ambientes distribuídos como o MPI. Todavia, no caso das arquiteturas híbridas, alguns trabalhos foram publicados e continuam em crescente desenvolvimento como Charm++, StarPU, etc.

2. Objetivo e Contribuições Esperadas

O objetivo deste trabalho é estudar mecanismos de execução eficientes com paralelismo de tarefas em arquiteturas híbridas multi-CPU e multi-GPU. Para tanto, alguns dos passos

necessários serão: (1) definir uma interface de programação para expressar algoritmos híbridos, ou seja, com implementações otimizadas para cada PU; (2) estudar estratégias de transferência de dados para reduzir os custos envolvidos; (3) pesquisar algoritmos de balanceamento de carga a fim de amenizar tais custos de atribuição de tarefas para PUs especializadas.

Este trabalho faz parte da colaboração entre o Grupo de Processamento Paralelo e Distribuído (GPPD/UFRGS) e do grupo francês MOAIS (INRIA Rhône-Alpes) no desenvolvimento da ferramenta XKAAPI [Kaa]. O XKAAPI é uma reimplementação do KAAPI [Gautier et al. 2007] com suporte ao paralelismo de tarefas de grão fino. A versão 1.0 suporta ambientes multi-CPU através das interfaces C (baixo nível) e C++ (Athapascan e Kaapi++). A nova interface Kaapi++ suporta a definição de diferentes implementações de uma tarefa para cada PU através de uma *Task Signature*. Ela permite descrever os parâmetros de uma tarefa e seus modos de acesso (leitura, escrita, etc), assim como suas especializações para cada PU.

Atualmente, no contexto deste trabalho, o estudo dos custos de execução em GPUs NVIDIA com CUDA objetiva definir um mecanismo eficiente onde várias tarefas GPU serão executadas. O próximo passo previsto é a incorporação do suporte multi-GPU na interface Kaapi++ com dependência de dados.

References

KAAPI. http://kaapi.gforge.inria.fr.

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. (2009). A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67.

Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Cambridge, USA.

Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The Implementation of the Cilk-5 Multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223.

Gautier, T., Besseron, X., and Pigeon, L. (2007). KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proc. of The 2007 international workshop on Parallel symbolic computation, PASCO'07*, London, CAN. ACM.

Reinders, J. (2007). *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly & Associates, Inc., Sebastopol, USA.