Controle de Granularidade com *threads* em Programas MPI Dinâmicos*

João Vicente 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

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

1. Introdução

O principal problema na programação paralela é decompor um problema inicial em subproblemas de tal forma que executem em paralelo. Problemas estáticos podem prever sua carga de trabalho e utilizam a decomposição por dados. Por outro lado, alguns problemas desconhecem sua carga antes da execução devido às características do algoritmo ou da arquitetura. Uma decomposição recursiva pode solucionar esta questão ao dividir recursivamente um problema inicial em menores até que sua carga seja conhecida. Esta divisão pode atribuir os subproblemas para tarefas criadas dinamicamente em tempo de execução.

Um fator importante para o desempenho de aplicações paralelas é a granularidade, que representa o grau de agrupamento de tarefas em um determinado processador. Entretanto, um controle de granularidade em aplicações dinâmicas depende do ambiente de programação devido ao grafo de dependências entre as tarefas ser desconhecido antes da execução. Alguns ambientes de programação como Cilk [Blumofe et al. 1995] e KAAPI [Gautier et al. 2007] propõem um controle de granularidade com tarefas abstratas. Todavia, estes ambientes possuem limitações de arquitetura e complexidade de programação. O padrão MPI possibilita dinamismo na criação de processos, mas as implementações não dispõem de um controle de granularidade.

Este trabalho propõe uma biblioteca (libSpawn) que incorpora um controle de granularidade no MPI através de tarefas, que podem ser processos ou *threads*. A implementação consiste na alteração da chamada `Spawn` para a escolha entre processos ou *threads*. A granularidade é definida em tempo de execução através de parâmetros que avaliam o contexto de execução: *cores* da arquitetura, carga de sistema e recursos do sistema operacional disponíveis. Estes parâmetros são avaliados localmente para cada processo e proporcionam distribuir a carga de trabalho para outros nós disponíveis. Neste trabalho, nenhum tipo de controle distribuído ou centralizado de tarefas é utilizado.

A libSpawn implementa a troca de mensagens de acordo com o padrão MPI de forma que mensagens entre processos usem chamadas MPI e mensagens entre *threads* usem transferências em memória compartilhada. A biblioteca manipula a comunicação entre *threads* através de procedimentos em memória compartilhada com exclusão mútua. A descrição de uma mensagem é recriada para incluir além do *message envelope* [Forum 1997] três campos de controle. As mensagens são armazenadas em duas listas para cada `Send` e `Receive` onde uma *thread* busca mensagens correspondentes para depois completa-las. Dessa forma, esta implementação elimina impasses entre *threads* e proporciona comunicações eficientes.

*Trabalho financiado pela CAPES

2. Resultados Obtidos e Conclusão

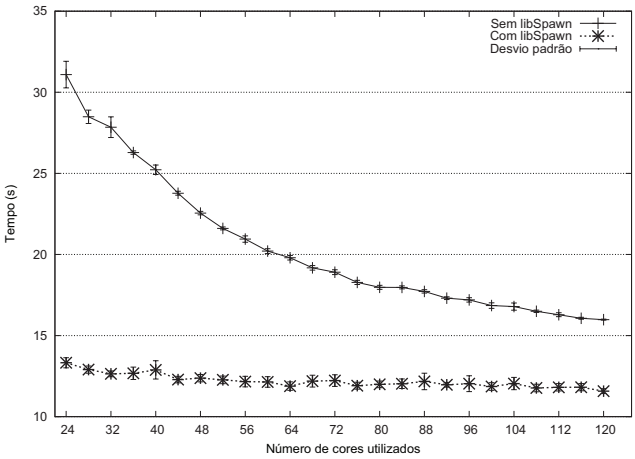

Figura 1. Resultados obtidos com a ordenação de números

O experimento consiste em uma ordenação *Mergesort* chamado *Cilksort* que acompanha o Cilk. Essa ordenação decompõe sua entrada com a técnica de Divisão-e-Conquista (D&C) onde cada divisão resulta em tarefas criadas dinamicamente. O gráfico da figura 1 mostra os tempos obtidos na execução com processos e com o controle proposto. A entrada é de 2 milhões de elementos, em ordem aleatória, que geram 10813 tarefas. As execuções foram realizadas em 30 nós da grade francesa Grid'5000 com 2 Intel Xeon Dual Core 2.33 GHz, em um total de 120 *cores*, e rede Gigabit Ethernet. As médias e desvio padrão foram obtidas a partir de 20 execuções independentes. As execuções começam com resultados acima de 32 *cores* devido às limitações na implementação MPICH2 que não permitem execuções com um número menor de *cores* para a entrada utilizada.

Os resultados obtidos demostram ganhos de 27% (120 *cores*) até 57% (24 *cores*) com relação ao uso de processos e sem controle de granularidade. Este ganho se deve à criação de tarefas como *threads* e às comunicações em memória compartilhada entre *threads*. Como trabalhos futuros, um mecanismo para a gerência do número de processos a serem criados será avaliado com o intuito de maximar o ganho de desempenho. Além disso, a inclusão de um escalonador em 2 níveis será estudado para melhorar a distribuição de carga e, por conseguinte, aumentar a eficiência.

Referências

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An Efficient Multithreaded Runtime System. In *PPOPP '95*, pages 207–216, New York, NY, USA. ACM Press.

Forum, M. P. I. (1997). MPI-2: Extensions to the Message-Passing Interface. Technical Report CDA-9115428, University of Tennessee, Knoxville, Tennessee.

Gautier, T., Besseron, X., and Pigeon, L. (2007). KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07*. ACM.