Comunicação Coletiva no Ambiente de Programação Paralela – DECK (*Distributed Execution and Communication Kernel*)

Rafael Ennes Silva, Rafael Bohrer Ávila, Philippe Olivier Alexandre Navaux

Instituto de Informática – LabTeC – DELL – UFRGS
Av. Bento Gonçalves 950, Bloco IV.
Caixa Postal 15064, CEP 91501-970 Porto Alegre – Brazil
Phone: +55 51 3316-6165 – Fax: +55 51 3316-7308
{resilva, avila, navaux}@inf.ufrgs.br

Introdução

Entende-se por paralelismo como sendo uma técnica de dividir tarefas grandes e complexas em tarefas menores que serão distribuídas e executadas simultaneamente em vários processadores. Paraleliza-se para atingir ganhos de desempenho e diminuir o tempo gasto no processamento [STE 99]. DECK é um ambiente de programação de aplicações paralelas e distribuídas, que oferece recursos de multiprogramação (*threads*), comunicação e sincronização; bem como serviços mais elaborados para aplicações específicas [BAR 00]. O modelo de programação paralela utilizado no DECK, para comunicação entre processos , é o de "Message Passing" ou passagem de mensagens que se caracteriza pelo uso de múltiplos processadores, porém cada processador possui uma memória própria e independente (memória distribuída) [BUY 99]. A comunicação no DECK é realizada através de *mailboxes*. A comunicação coletiva existe para dar suporte a aplicações regulares, que fazem uso intenso de operações matemáticas com vetores e matrizes. Ela caracteriza-se pela comunicação entre processos pertencentes a um mesmo grupo. A proposta da presente pesquisa é remodelar os algoritmos das funções de comunicação coletiva da biblioteca DECK para melhorar o desempenho. Tais funções são equivalentes as encontradas na biblioteca MPI (*Message Passing Interface*) [PAC 97, MPI 94] . O objetivo dessa similaridade é a facilidade que um usuário de MPI encontrará ao utilizar uma biblioteca com características semelhantes.

Comunicação Coletiva

A comunicação coletiva apresenta o conceito de ser do tipo grupo fechado [TAN 95], isto é, somente membros do grupo podem enviar e receber mensagens, coibindo assim, o acesso de membros exteriores. Esse tipo de comunicação é utilizado em situações em que o esforço conjunto de um número de processos é necessário para solucionar um determinado problema.

Existem algumas primitivas para se estabelecer esse tipo de comunicação, as quais são: *Broadcast* – os dados de um processo são enviados para os demais processos; *Barrier* – assegura que todos os processos do grupo cheguem no mesmo ponto de

execução para então, continuar a aplicação; *Gather* – os processos do grupo enviam seus dados para um único processo que ordena os pacotes conforme o número do processo; *Scatter* – os dados de um processo são divididos e distribuídos entre todos os processos do grupo; *Allgather* – são agrupados os dados dos processos do grupo e esse agrupamento é enviado para todos os processos; *Reduce* - faz uma operação (soma, produto, máximo, mínimo e outras) com os dados de todos processos do grupo e o resultado é entregue à um processo; *Allreduce* – faz uma operação (soma, produto, máximo, mínimo e outras) com os dados de todos os processos e o resultado é gravado em todos os processos do grupo.

Descrição de Algoritmos

Foram estudados alguns algoritmos para melhorar a distribuição do trabalho entre os processos e diminuir a latência. Em termos de desempenho de um algoritmo, a complexidade, taxa de crescimento dos recursos (tempo) necessários no que refere-se aos tamanhos dos dados que processa [LOU 00], é um fator decisivo.

A configuração imaginada foi a árvore de processos com complexidade O(log n), cujo a taxa de crescimento (tempo) é pequena em relação ao número de nodos envolvidos. A árvore é atravessada em ordem de nível, começando pelo processo raiz e prosseguindo para baixo visitando os nodos de cada nível da esquerda para direita. Essa árvore é balanceada para a esquerda porque todos os níveis ocupam as posições mais à esquerda no último nível[LOU 00].

A implementação da Broadcast em árvore levou em conta as primitivas de *post* e *retrieve* do DECK, ou seja, o nodo raiz faz um *post* para o nodo seguinte, e este faz um *retrieve* e envia para o nodo seguinte caso exista. Já no *Barrier* , a árvore é percorrida das folhas para a raiz enviando-se uma mensagem vazia. Quando o último processo é recebido pelo nodo raiz significa que todos os membros do grupo chegaram ao mesmo ponto de execução. Sendo assim a árvore começa a ser percorrida no sentido normal através do primeiro *post* disparado pelo nodo raiz e liberando demais os processos, bloqueados pelo *retrieve*, para seguirem sua execução.

Nas primitivas *Gather* e *Scatter* inicialmente, na passagem de parâmetros, era necessário um *buffer* de envio e um *buffer* de resposta, porém nas funções os dados tinham que ser empacotados e enviados para o processo destino e depois recebidos e desempacotados para só assim, serem utilizados. Para minimizar este trabalho, ao invés de *buffers*, serão passados mensagens como parâmetros. Para isso, foi montado um mecanismo de divisão de mensagens. A tarefa dele é subdividir a mensagem em pedaços conforme o número processos do grupo.

Análise de Desempenho

Para validar a configuração de árvore de processos na comunicação coletiva, este algoritmo foi implementado primeiramente, nas funções *Barrier e Broadcast.* Essas funções foram testadas no cluster do LabteC (Laboratório de Tecnologia em Cluster) – DELL, composto por um servidor de 4 processadores Xeon 1.8GHz e 1GHz de memória RAM e 20 nodos duais *Pentium* III 1.13GHz e 1GB de memória RAM.

Os testes foram realizados utilizando a biblioteca do DECK, desenvolvido pelo GPPD-UFRGS, e a biblioteca MPICH, desenvolvida pela *Argonne National Lab*. Eles procederam da seguinte forma: primeiramente foram desenvolvidas duas aplicações equivalentes, uma com primitivas MPI, e outra com primitivas DECK para se obter uma comparação de desempenho; as simulações de chamada a *Barrier* e a *Broadcast* foram executadas cerca de cem mil vezes para obter uma melhor precisão, variando-se o número de nodos do cluster, e no caso da *Broadcast*, variou-se também o tamanho da mensagem à ser enviada; o tempo de cada acesso foi medido e após o término das iterações foi gerada uma média com os valores medidos. Na figura 1, é ilustrada a simulação da *Barrier* para o DECK e para o MPI. Na Figura 2, está o gráfico que relaciona a *Broadcast* do MPI com a *Broadcast* do DECK para 1Kb de dados na mensagem. Na Figura 3, novamente é comparado a *Broadcast,* porém com mensagem de 1Mb de tamanho.

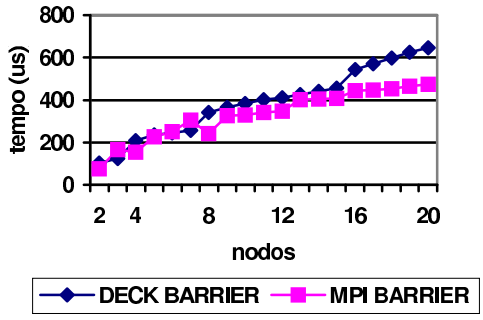

Figura 1 *Barrier* DECK x MPI.

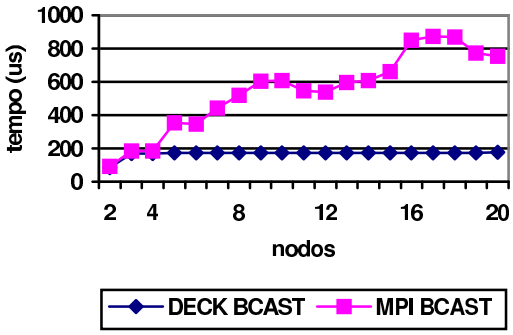

Figura 2 *Broadcast* DECK x MPI para 1KB de dados na mensagem.

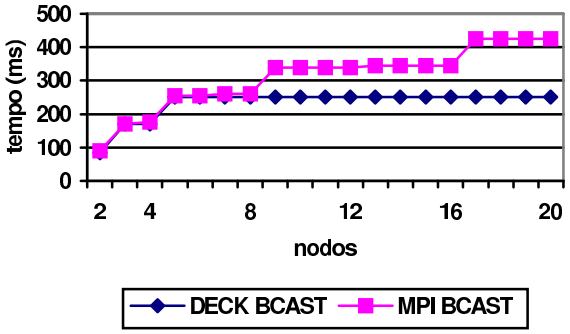

Figura 3 *Broadcast* DECK x MPI para 1MB de dados na mensagem.

Pela análise dos resultados obtidos nos gráficos, foi possível deixar as primitivas do DECK equiparáveis às primitivas do MPI. Na *Barrier,* o desempenho foi relativamente sendo compensado entre as duas bibliotecas conforme se aumentava o número de nodos até 8. Após, o desempenho do DECK começou a cair . Para o caso da *Broadcast* para mensagens de 1KB foi notável a quase estabilização do tempo de execução, porém ele aumenta lentamente. Quando o tamanho da mensagem foi aumentado, o comportamento da *Broadcast* do DECK foi similar ao do MPI. Neste

caso, se fosse colocado mais nodos para a simulação da *Broadcast*, talvez o comportamento fosse análogo ao simulado com mensagem de dados de 1KB. O comportamento apresentado na *Broadcast* é justificado pelo fato da comunicação ser síncrona ou assíncrona dependendo da disponibilidade de *buffers* no Kernel do Linux.

Conclusões

No mecanismo de comunicação coletiva existe uma propriedade relativa a escalabilidade. No que se refere aos tipos de algoritmos encontrados, muitos funcionam bem quando os grupos são pequenos, e a medida que aumenta-se o número de membros do grupo, o desempenho começa a ser prejudicado devido às perdas de tempo de comunicação. E isto também é observado nos testes realizados.

Alcançados os objetivos de desempenho com essas simulações iniciais, torna-se possível à partir de agora, criar as primitivas do DECK que também dependem do mesmo algoritmo de árvore que são a A*llgather* e *Allreduce*. Já as primitivas *Gather*, R*educe* e *Scatter*, em virtude de não terem sido totalmente otimizadas, não foram incluídas nessas simulações.

Referências

[BAR 00] BARRETO, M. A . **DECK – Um Ambiente para Programação Paralela em Agregados de Multiprocessadores.** Dissertação de Mestrado, Porto Alegre: PPGC/UFRGS, 2000.

[BUY 99] BUYYA, R. (Ed.). **High performance cluster computing**: architectures and systems. Upper Saddle River: Prentice Hall PTR, 1999. 849p.

[LOU 00] LOUDON, K. et al. **Dominando Algoritmos com C**. Rio de Janeiro: Editora Ciência Moderna Ltda, 2000, 193p.

[MPI 94] MPI FORUM. **The MPI message passing interface standard**. Knoxville: University of Tennessee, 1994.

[PAC 97] PACHECO, P. S. (Ed.). **Parallel Programming with MPI.** San Francisco, California: Morgan Kaufmann Publishers, Inc, 1997. 65p

[STE 99] STERLING, T. L. et al. **How to build a Beowulf**: a guide to the implementation and application of PC clusters. Cambridge: MIT, 1999. 239p.

[TAN 95] TANENBAUM, A. S. et al. **Sistemas Operacionais Modernos**. Rio de Janeiro: Prentice-Hall do Brasil Ltda,, Feb. 1995. 304p