Paralelização do Solver Intervalar LSS

Alexandre Almeida[1], Carlos Amaral Hölbig[1]

[1]Curso de Ciência da Computação – Universidade de Passo Fundo (UPF)
Campus 1 – BR 285 – Bairro São José – CEP 91501-970 – Passo Fundo – RS – Brasil
alexandre.cpp@gmail.com, holbig@upf.br

Resumo. *Este artigo apresenta uma versão paralela do solver LSS para máquinas de memória distribuída baseada em uma versão sequencial otimizada, desenvolvida por meio da eliminação de cálculos redundantes do algoritmo original do solver. Além disso, com o objetivo de reduzir o custo da comunicação do programa paralelo, foram desenvolvidas funções de comunicação coletiva que suportam determinados tipos de dados da biblioteca C-XSC.*

1. Introdução

A resolução de sistemas densos de equações lineares é uma operação que possui várias aplicações nas áreas das ciências e engenharias, em especial para os campos da simulação e modelagem [Cláudio and Marins 2000].

O *solver* LSS (*Linear System Solver*) é um programa que tem como tarefa a resolução de sistemas lineares com matrizes densas, controlando a instabilidade numérica de máquina por meio do paradigma da Computação Verificada, com auxílio da biblioteca de alta exatidão C-XSC. Através disso, o *solver* é capaz de garantir a exatidão da solução final, porém ao custo de um baixo desempenho, como pode ser observado em [Zimmer and Krämer 2008].

Devido ao alto tempo de processamento do *solver*, este trabalho apresenta uma versão paralela do LSS para máquinas de memória distribuída, bem como uma versão sequencial otimizada, ambas mantendo tanto a qualidade numérica do *solver* original e suas características essenciais com relação à Computação Verificada. Para a implementação do programa paralelo, foi adotado um modelo de algoritmo por divisão e conquista com o auxílio da biblioteca MPI, que permite a exploração de um paralelismo entre-nós em um agregado, e através da criação de *POSIX threads* em cada processo, capaz de tirar proveito de arquiteturas *multi-core* de processadores.

Além disso, para efetuar a distribuição dos dados das matrizes entre os processos MPI, foram desenvolvidas funções de comunicação coletiva com suporte a determinados tipos de dados da biblioteca C-XSC, como uma tentativa de se reduzir o custo da comunicação entre os processos paralelos. O desenvolvimento dessas funções foi baseado nas funções para comunicação ponto-a-ponto apresentadas em [Grimmer 2005].

2. Solver LSS sequencial otimizado

O *solver* LSS é composto, basicamente, por dois algoritmos principais: uma etapa de precisão simples e uma etapa de precisão dupla. Quando um sistema linear é submetido

ao *solver*, o algoritmo de precisão simples executa uma série de sub-tarefas a fim de tentar encontrar uma solução para o sistema linear. Caso a etapa de precisão simples não seja capaz de encontrar uma solução (por exemplo, para sistemas mal-condicionados), o programa submete o sistema linear à etapa de precisão dupla, que tenta encontrar uma solução para o sistema utilizando uma maior precisão.

A versão sequencial do *solver* LSS foi otimizada por meio da eliminação de computações redundantes identificadas entre as sub-tarefas das etapas de precisão simples e dupla, e através da otimização no método de Gauss-Jordan, utilizado para efetuar a operação de inversão de matrizes.

Além da eliminação de recálculos de operações no algoritmo do *solver*, o método de Gauss-Jordan foi otimizado substituindo-se os objetos do tipo *real* da biblioteca C-XSC, por variáveis do tipo primitivo *double*, nativo da linguagem C++, com o objetivo de se fazer uma redução da sobrecarga introduzida pelo uso da biblioteca C-XSC na função. Entretanto, para que o método de Gauss-Jordan continuasse numericamente estável, o uso dos objetos do C-XSC foi mantido apenas nos pontos do código que efetuavam operações aritméticas, de tal forma que a instabilidade numérica ainda fosse capaz de ser controlada pela biblioteca.

Através das otimizações implementadas no LSS sequencial, foi possível observar um ganho considerável no desempenho do *solver*. Por exemplo, considerando a execução de um sistema de ordem 2048 em um computador com processador Intel Core 2 Duo E4500, à frequência de 2,2 GHz, e 2 GB de memória principal, o tempo de processamento foi reduzido de, aproximadamente, 455,7 minutos para 251,4 minutos na versão otimizada, o que representa uma diferença de 44,8% no tempo de execução.

3. Solver LSS paralelo

A versão paralela do *solver* LSS foi desenvolvida com base na versão otimizada descrita na Sessão 2. Através dos testes de desempenho do LSS sequencial apresentados em [Almeida 2008] foi possível observar que a degradação do tempo de processamento geral do *solver* é causada, principalmente, por determinadas sub-tarefas das etapas de precisão simples e dupla, sendo, portanto, os trechos do programa que foram paralelizados.

O modelo de algoritmo paralelo adotado foi por divisão e conquista, onde a execução do programa principal é realizada por um processo gerenciador que, ao atingir as sub-tarefas paralelas, distribui os dados das matrizes entre *n* processos trabalhadores (incluindo o próprio gerenciador). O método de particionamento das matrizes entre os processos foi por blocos de linhas contíguas (*block-striped partitioning*), onde as matrizes são quebradas em blocos homogêneos de linhas completas, que são alocadas aos processos. Além disso, foram implementadas funções de comunicação coletiva para a troca de dados da biblioteca C-XSC, sendo descritas na sessão 3.1.

3.1. Funções de comunicação coletiva com suporte ao C-XSC

Em [Grimmer 2005] são apresentadas funções MPI para a troca de dados do C-XSC entre processos, através de um modelo de comunicação ponto-a-ponto (*send/receive*). Como a comunicação entre processos paralelos é um fator que compromete diretamente a eficiência de aplicações de memória distribuída, foram implementadas funções de

comunicação coletiva para a distribuição dos dados das matrizes do *solver*. É importante salientar que as novas rotinas foram baseadas nas implementações de Grimmer (2005).

As novas funções MPI para comunicação coletiva suportam as operações de *MPI_Bcast*, *MPI_Gather*, *MPI_Scatter* e *MPI_Allgather* para o tipo *rmatrix* do C-XSC. Para os tipos *imatrix* e *ivector*, apenas as funções *MPI_Gather* e *MPI_Scatter* foram implementadas.

3.2. Análise de desempenho do *solver* paralelo

Os testes de desempenho da versão paralela foram conduzidos em um ambiente do tipo NOW (*Network of Workstations*) no Laboratório Central de Informática da Universidade de Passo Fundo. O agregado foi constituída por 32 computadores, cada um com um processador *dual-core* Intel Core 2 Duo E4500, à frequência de 2,2 GHz, e com 2 GB de memória principal. A comunicação entre os nodos é feita por uma rede *fast ethernet*, e intermediada por um *switch*.

O sistema operacional instalado em cada nodo foi o GNU/Linux Ubuntu 7.04 (*kernel* 2.6.22), com distribuição LAM/MPI 7.1.2, biblioteca C-XSC 2.2.2 e compilador GCC 4.1. Os tempos de processamento do *solver* paralelo foram obtidos para sistemas de ordem 64, 128, 256, 512, 1024 e 2048, com 2, 4, 8, 16, 32 e 64 processos, alocando-se 2 processos MPI por nodo do agregado. A média dos tempos foi feita com base em 10 repetições para cada caso.

O gráfico da Figura 1 ilustra o *speedup* da versão paralela do *solver* LSS. Embora tenha sido possível notar uma aceleração razoável para as sub-tarefas paralelas de forma isolada, os testes de desempenho do *solver* LSS como um todo mostraram um crescimento mais tímido para o *speedup*, e, consequentemente, uma queda bastante acentuada para a eficiência. Esses resultados foram obtidos devido ao fato de que a sub-tarefa de inversão de matrizes pelo método de Gauss-Jordan não foi paralelizada, sendo que essa sub-tarefa representa um dos gargalos do sistema.

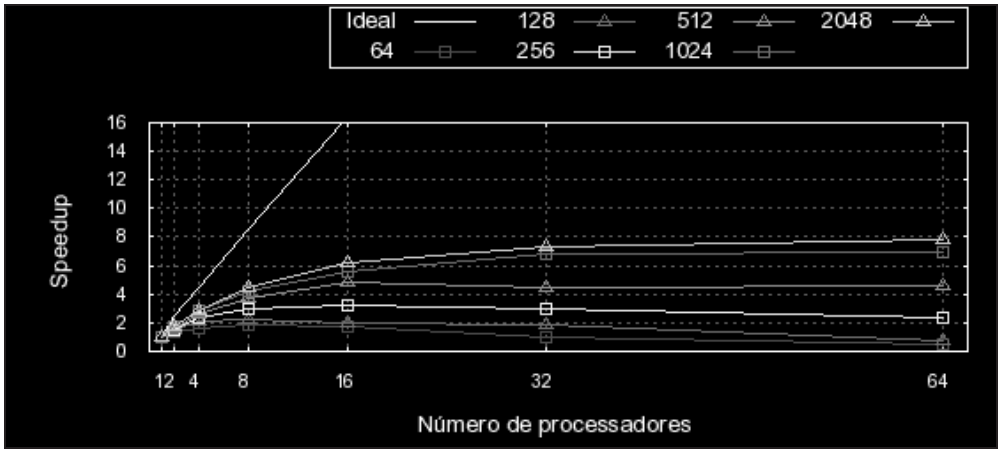

Figura 1. Speedup do *solver* paralelo

Com o objetivo de verificar o ganho em se utilizar um modelo de comunicação coletiva com relação a um modelo de comunicação ponto-a-ponto, foi feita uma comparação entre os tempos de comunicação para o envio de uma matriz do tipo

rmatrix de ordem 2048, utilizando a função de comunicação coletiva *MPI_Bcast*, e as funções de comunicação ponto-a-ponto *MPI_Send* e *MPI_Recv*. A matriz foi replicada entre 2, 4, 8, 16, 32 e 64 processos (dois processos por nodo).

Através dos tempos de comunicação obtidos, foi possível observar uma redução considerável no tempo de replicação da matriz. Com 64 processos, o tempo gasto pelas funções de *send/receive* foi de 408,1 segundos, enquanto que a função que efetua *broadcast* levou em torno de 14,4 segundos, representando uma diminuição de, aproximadamente, 96,5% no tempo de envio.

4. Considerações finais e trabalhos futuros

Por meio do detalhamento das sub-tarefas que compõe o algoritmo do *solver* foi possível obter um considerável ganho em desempenho na versão sequencial através da eliminação de recálculos efetuados no algoritmo, além da otimização do método de Gauss-Jordan, que também contribuiu para o ganho em desempenho dessa versão.

Em relação ao uso da biblioteca C-XSC em um ambiente paralelo, foi possível perceber que o uso de funções da biblioteca MPI para comunicação coletiva reduz consideravelmente o custo da comunicação de dados pela rede, se comparado com funções de comunicação ponto-a-ponto. Através do desenvolvimento dessas funções, estendeu-se ainda o conjunto de funções MPI para comunicação ponto-a-ponto dos tipos de dados do C-XSC, disponibilizadas por Grimmer (2005).

Embora as sub-tarefas paralelas, de forma isolada, apresentaram um *speedup* mais próximo do ideal, como é verificado em [Almeida 2008], o desempenho geral do *solver* LSS mostrou-se com um crescimento não muito próximo ao ideal à medida que a execução é particionada em mais processos. Esse efeito é causado, principalmente, pelo método de inversão de matrizes, que é realizado de forma sequencial no processo gerenciador.

Portanto, como trabalhos futuros ficam o estudo do método de Gauss-Jordan a fim de paralelizá-lo e a busca por métodos alternativos mais eficientes para realizar a inversão de matrizes como, por exemplo, o método da decomposição LU. Além do mais, o método de inversão a ser adotado deve continuar com a característica de ser numericamente estável, para que a qualidade numérica dos resultados possa ser mantida.

Referências

Almeida, A. (2008). *Paralelização do Solver Intervalar LSS*. Trabalho de conclusão de curso (Graduação em Ciência da Computação) – Universidade de Passo Fundo, Passo Fundo.

Cláudio, D. M. and Marins, J. M. (2000). *Cálculo Numérico Computacional: Teoria e Prática*. Editora Atlas.

Grimmer, M. (2005). An MPI Extension for the Use of C-XSC in Parallel Environments. In *WRSWT 2005/3*. Wuppertal, Üniversität Wuppertal.

Zimmer, M. and Krämer, W. (2008). Fast (Parallel) Dense Linear Interval Systems Solver in C-XSC Using Error Free Transformations and BLAS. In *Numerical Validation in Current Hardware Architectures*. IBFI, Schloss Dagstuhl, Germany.