UM SISTEMA DE PROGRAMAÇÃO E PROCESSAMENTO PARA SISTEMAS MULTIPROCESSADORES

Liria Matsumoto Sato

Laboratório de Sistemas Integráveis (LSI)
Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, travessa 3, nº.158
CEP: 05508 - São Paulo - SP
tel - (011) 8159322 ramal 3270
e-mail: liria@lsi.usp.br

RESUMO

A complexidade para utilizar com eficiência os recursos de paralelismo presentes em computadores multiprocessadores exige ambientes de programação adequados. É apresentado neste trabalho um ambiente de programação para sistemas multiprocessadores que permite através de uma linguagem de programação paralela explorar múltiplos níveis de paralelismo.

ABSTRACT

The complexity in using efficiently parallelism resources in multiprocessor computers requires appropriate programming environments. In this paper, it is presented a programming environment for multiprocessor systems that provides the parallelism exploitation on multiple levels by a parallel programming language.

1)Introdução

Computadores de alto desempenho estão sendo utilizados em aplicações em diversas áreas, desde a engenharia e a física, até a medicina e a biologia.

Algumas aplicações que envolvem altíssimos tempos de processamento tornar-se-ão viáveis apenas em computadores que ofereçam um alto grau de desempenho.

Na busca de obter alto desempenho, uma alternativa que tem sido empregada é a utilização de múltiplos processadores. Tais sistemas são conhecidos como sistemas multiprocessadores. Este artigo apresenta um ambiente de programação voltado para esta classe de máquinas.

Arquiteturas paralelas têm sido propostas, sendo que algumas resultaram em protótipos e outras tornaram-se produtos disponíveis no mercado.

Entretanto o suporte para a programação paralela se mostra ainda inadequado, não explorando satisfatoriamente os recursos de paralelismo presentes na máquina.

Aspectos de programação paralela e um sistema de programação e processamento que permite a expressão e a exploração de paralelismo no programa, em múltiplos níveis de granularidade, serão aqui apresentados.

2)Aspectos de Programação Paralela

A programação paralela, sendo um campo relativamente novo, é uma área onde muitas questões como padronização, portabilidade e ferramentas não estão ainda resolvidas. A variedade de arquiteturas paralelas propostas aumentam a complexidade destas questões. Linguagens diversas, sejam linguagens novas ou extensões de linguagens convencionais, têm sido propostas [1,2,3,4,5], visando cada uma a obtenção de alto desempenho para algoritmos implementados em uma máquina com uma arquitetura específica. Pesquisas estão sendo realizadas, visando a portabilidade [6].

Existe um compromisso entre alto desempenho, portabilidade e facilidades de programação. Pode-se ganhar muito em facilidades de programação, comprometendo, entretanto, o desempenho.

Buscando um equilíbrio entre estes parâmetros, as linguagens de programação paralela acabam envolvendo algum nível de complexidade para a programação.

Uma questão relevante é a especificação do paralelismo, que implica em como e qual paralelismo expressar [1]. Uma linguagem ideal deveria oferecer construç•es sintáticas para expressar o paralelismo em todos os níveis de granularidade. Entretanto, detectar as fontes de paralelismo é uma tarefa que dependendo do problema envolve uma certa complexidade. Sistemas paralelizantes eliminam esta tarefa do usuário [1]. Contudo, algumas

fontes de paralelismo são de difícil detecção para estes sistemas [12], podendo ser exploradas explicitamente. Uma solução, que reune as vantagens das paralelizações manual e automática, é a combinação das duas.

3)O Projeto do Sistema de Programação e Processamento CPAR

O projeto do sistema de programação e processamento CPAR tem como objetivos:

- oferecer um sistema que permita expressar com clareza e facilidade o máximo de paralelismo presente no programa, através da linguagem , mantendo um equilíbrio entre facilidade de programação e eficiência;
- implementar uma biblioteca de paralelismo que dê suporte para o processamento paralelo, proporcionando alto desempenho;

Este sistema permite ao usuário implementar algoritmos paralelos, mas também atendendo a uma de suas finalidades, oferece o suporte de programação e processamento que será utilizado pelas ferramentas que comporão o ambiente CPAR. Uma ferramenta que está em andamento é um sistema paralelizante, que deverá explorar o paralelismo em "loops" em programas escritos na linguagem CPAR, automatizando a exploração neste nível de paralelismo, mas não eliminando a possibilidade do usuário de expressar paralelismos explicitamente. Esta ferramenta terá como entrada o programa fonte escrito em CPAR e como saída um programa fonte paralelizado escrito em CPAR que será compilado pelo compilador CCPAR. Esta técnica tem a vantagem de permitir o uso do sistema paralelizante em diversas máquinas, independendo do seu ambiente de processamento paralelo. É utilizada, por exemplo, pelo sistema PARAFRASE [7] e pelo compilador paralelizante POWER C [8] e POWER FORTRAN [8].

O sistema de programação e processamento aqui apresentado e que já possui uma primeira versão implementada é composto de um compilador da linguagem CPAR [9] e de uma biblioteca de paralelismo.

4)O Paradigma de Programação

Paralelizar um programa implica em distribuir o trabalho que realiza em tarefas que podem ser executadas paralelamente. Entretanto entre tarefas paralelas não pode existir dependências de dados, pois isto exigiria uma seqüencialidade na execução.

Um bloco de tarefas seqüenciais pode ser executado paralelamente a outros blocos de tarefas seqüenciais. Este agrupamento em blocos contendo elementos que devem ser executados seqüencialmente pode ser realizado em múltiplos níveis. Obtém-se desta forma uma paralelização em múltiplos níveis de paralelismo.

Uma das caracteristicas do modelo de programação do sistema CPAR [10] é a

paralelização da função principal ("main()") em múltiplos níveis. Esta é uma paralelização de grossa granularidade ("coarse grain"). A figura 1 ilustra esta característica do modelo.

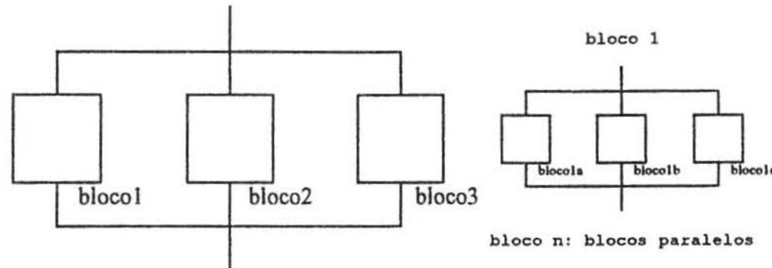

Figura 1: Processo principal com múltiplos níveis de paralelismo

No exemplo apresentado na figura 1 tem-se, não considerando o custo da paralelização:

t= t_maximo (t_bloco1, t_bloco2, t_bloco3)

t_bloco1= t_maximo(t_bloco1a, t_bloco1b, t_bloco1c)

É fato patente que devido ao custo inerente da paralelização, como por exemplo o custo referente à criação de processos e à sincronização, a granulação influi no desempenho do algoritmo.

Uma forma de amenizar o efeito deste custo é diminuir o custo relativo às criações de processos, criando processos e deixando-os prontos para execução. Esta técnica é utilizada por alguns sistemas de processamento paralelo [11] para explorar com eficiência paralelismo em granularidade fina, no nível de "loops" ou blocos de instruções paralelos, aqui referenciado como "microtasking". Uma outra alternativa, denominada "multitasking" ou "macrotasking", é explorar o paralelismo em níveis de granularidade grossa ("coarse grain"), por exemplo no nível de subrotinas. Alguns sistemas como o C concorrente [5] oferecem esta alternativa.

O sistema CPAR usa uma combinação das duas alternativas. A macrotarefa deve envolver uma quantidade substancial de processamento, caracterizando uma granulação grossa. Em cada macrotarefa podem estar presentes múltiplas microtarefas que são executadas paralelamente pelos processos criados e ativados no início da criação da macrotarefa. O sistema CPAR oferece recursos para explorar a hierarquia de memória presente em arquiteturas baseadas em agrupamentos de processadores ("clusters"). Variáveis compartilhadas globais podem ser acessadas pelas microtarefas de todas as macrotarefas. Variáveis compartilhadas locais à macrotarefa devem ocupar a memória do "cluster" ao qual pertencem os processadores que a estão executando, e podem ser acessadas apenas pelas suas microtarefas. Tem-se que locações visíveis para um maior número de processadores proporcionam um acesso mais caro do que aquelas visíveis para um menor número [12].

A figura 2 ilustra o modelo de programação do sistema CPAR, apresentando os

níveis de paralelismo.

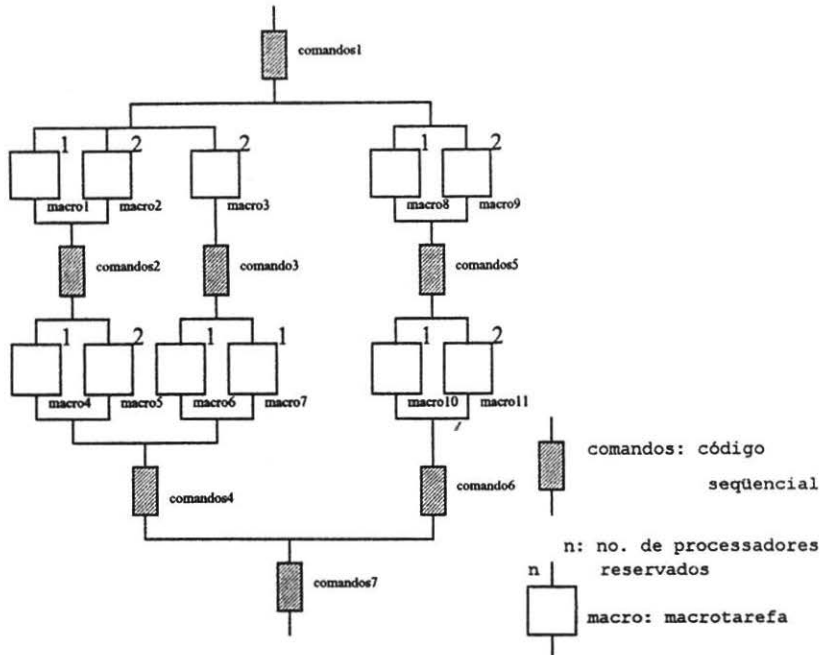

Figura 2: Modelo de programação do sistema CPAR

No exemplo mostrado na figura 2, não considerando o custo devido à paralelização, tem-se:

```
t = t_comandos1 +
   t_maximo(
      (  t_maximo(
         (  t_maximo(t_macro1,t_macro2) +
            t_comandos2 +
            t_maximo(t_macro4,t_macro5)),
         (  t_macro3+
            t_comandos3+
            t_maximo(t_macro6,t_macro7))) +
         t_comandos4),
      (  t_maximo(t_macro8, t_macro9) +
         t_comandos5 +
         t_maximo(t_macro10, t_macro 11 ) +
         t_comandos6)) +
   t_comandos7
```

Resumindo, tem-se, na programação em CPAR os seguintes elementos:

- elemento seqüencial : é uma porção de códigos, os quais devem ser executados seqüencialmente.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

- macrotarefa: é uma porção de códigos, ao nível de subrotina, que por sua vez contém um nível mais fino de paralelismo, as microtarefas. Macrotarefas podem ser executadas paralelamente. Quando uma macrotarefa é colocada em execução, o processo que a ativou continua a sua execução, podendo, inclusive ativar outras macrotarefas, promovendo a paralelização de suas execuções.
- microtarefa: é uma porção de código seqüencial, contida em um "loop" cujas iterações são executadas paralelamente ou em um bloco de instruções que é executado paralelamente a outros blocos.
- blocos paralelos: são porções do código da função principal ("main()") do programa que são executadas paralelamente. Um bloco pode conter blocos paralelos, ou seja, é permitido o aninhamento de blocos. Um bloco pode conter os seguintes elementos: elementos seqüenciais, macrotarefas e blocos paralelos.
- função principal: é a função principal do programa ("main()"). Pode conter os seguintes elementos: blocos, elementos seqüenciais e macrotarefas.

5) A Linguagem CPAR

A linguagem CPAR [9] apresenta as seguintes características:

- possui primitivas para explicitar blocos paralelos.
- possui primitivas para a declaração e a execução de macrotarefas.
- possui primitivas para explicitar microtarefas
- permite a declaração de variáveis locais e de variáveis compartilhadas globais e locais à macrotarefa.
- fornece um mecanismo de exclusão mútua, permitindo a utilização segura da memória compartilhada, denominado monitor, presente na linguagem Pascal Concorrente [8].
- a comunicação entre macrotarefas pode ser realizada por: passagem de mensagem ou memória compartilhada.
- a sincronização entre macrotarefas é efetuada por rotinas de biblioteca.

Uma descrição detalhada da linguagem CPAR encontra-se em [9]. Apresentaremos aqui uma descrição sucinta e as alterações realizadas na implementação atual.

5.1. Blocos paralelos

Esta é uma construção que foi adicionada à primeira versão da linguagem CPAR.

Blocos paralelos, já apresentados na seção 3, podem estar presentes na função

principal ("main") do programa. Podem conter elementos seqüenciais, macrotarefas e blocos paralelos aninhados. Têm a seguinte construção:

```
cobegin
        bloco 1 de comandos
also
        bloco 2 de comandos
also
        .....
also
        bloco n de comandos
coend
```

5.2. Macrotarefas

A declaração de uma macrotarefa consiste de:

```
task spec nome_macrotarefa( parâmetros )
        { declarações das entradas }
```

nome_macrotarefa: nome da macrotarefa

parâmetros: é opcional

declarações de transação: é opcional. É utilizada na passagem de mensagem entre macrotarefas. A declaração do corpo de uma macrotarefa tem a forma:

```
task body nome_macrotarefa( parâmetros )
        declaração de parâmetros
        { declarações de variáveis
        comandos }
```

Macrotarefas são criadas e ativadas, explicitamente, através do comando create.

```
create n,nome_macrotarefa( argumentos )
```

n: total de processadores para executar a macrotarefa.

argumentos: parâmetros passados por valor para a macrotarefa.

5.3. Microtarefas

Dentro de uma macrotarefa são possíveis dois tipos de paralelismo: o paralelismo homogêneo, onde as microtarefas executam o mesmo código para dados distintos, e o heterogêneo, onde as microtarefas executam códigos distintos.

O comando forall permite ao usuário especificar o paralelismo homogêneo, atribuindo um bloco de iterações a cada um dos n processadores alocados à macrotarefa.

```
forall i=1 to max {
```

```
a[i]=b[i]+1;
c[i]=a[i]*2 }
```

O paralelismo heterogêneo é especificado pelo comando parbegin.

```
parbegin
        a=a+sqr(b);
        x=x*2
also
        c=c-sqr(c);
        y=y-1
parend
```

As sequências de comandos são executadas paralelamente.

5.4. Monitor

Monitor é um mecanismo que permite o acesso seguro a variáveis compartilhadas, através de funções que são executadas somente uma de cada vez.

O monitor tem a forma:

```
monitor nome_monitor
        declarações de variáveis compartilhadas        declarações de variáveis locais
        { funções do monitor }
        { iniciação do monitor }
```

A iniciação do monitor é uma lista de comandos executada no início do programa, iniciando as variáveis compartilhadas. A chamada de uma função de um monitor tem a forma:

nome_monitor.função(argumentos)

5.5. Comunicação por passagem de mensagem

A passagem de mensagem entre macrotarefas é realizada através das entradas especificadas na declaração da macrotarefa por comandos de transmissão e recepção. Uma alteração foi realizada nas construções para passagem de mensagem sobre a primeira versão [9] da CPAR, buscando oferecer construções mais simples para a utilização.

- declaração de entrada:
  `entry nome_entrada(declaração do tipo da mensagem)`
- transmissão de mensagem:
  `send nome_macrotarefa.nome_entrada( mensagem )`
- recepção de mensagem:
  `receive nome_entrada (variável)`
- verificação de chegada de mensagem:

state nome_entrada (variável)

5.6. Semáforos e Eventos

Situações onde processos cooperam entre si, exigem que estes estejam sincronizados. CPAR oferece dois mecanismos de sincronização: semáforos e eventos.

Operações sobre semáforos e eventos são realizadas através de chamadas de rotinas da biblioteca de paralelismo, apresentadas na seção 6.

5.7. Espera pelo término das macrotarefas ativas

O comando wait_proc(nome_macrotarefa) coloca o processo em espera pelo término da macrotarefa especificada.

A rotina da biblioteca de paralelismo wait_all() coloca o processo em espera pelo término de todas macrotarefas ativas.

6)A biblioteca de paralelismo

A biblioteca de paralelismo contém funcões, cujas chamadas são geradas pelo compilador da linguagem CPAR, e outras disponíveis ao usuário. É através destas funções que é implementada a paralelização do programa.

A biblioteca contém as seguintes rotinas:

- int alloc_proc(nprocs): reserva nprocs processa-dores.
- mc_var_inic(): inicia variáveis compartilhadas internas da biblioteca de paralelismo.
- int exec_task(indice_macro, numero_procs, pidf): coloca a macrotarefa em execução.
- def_task (nome_macro, indice_macro):insere nome_macro em uma tabela de macrotarefas.
- int barrier(): início de barreira.
- end_barrier(): fim de barreira.
- end_task(): finaliza execução de macrotarefa.
- end_program(): finaliza execução de programa.
- wait_proc(indice_macro): espera o término da macrotarefa.
- wait_all(): espera o término de todas as macrotarefas em execução.
- create_sem(&semaforo, valor_inicial): cria semáforo e atribui valor inicial.
- rem_sem(&semaforo): retira semáforo.
- lock (&semaforo): obtém semáforo.
- unlock(&semaforo): libera semáforo.

- create_ev(&E): cria evento.
- rem_ev(&E): retira evento.
- set_ev(&E): ativa evento E.
- res_ev(&E): apaga evento E.
- int read_ev(&E): lê estado do evento E.
- wait_ev(&E): espera até que E seja setado.

7)Exemplos de aplicação em CPAR

No exemplo 1 é apresentado o uso de "loops" paralelos em uma macrotarefa.

```
/* multiplicação de 2 matrizes, resultado em uma terceira */
#include <stdio.h>
#define SIZE 10    /* memoria de dados compartilhada global*/
shared float a[SIZE][SIZE]; /* matriz 1 */
shared float b[SIZE][SIZE]; /* matriz 2 */
shared float c[SIZE][SIZE]; /* matriz 3 */
/* inicia matrizes */
task spec init_matrix();
task body init_matrix(){
 int i,j;

 forall i=0 to SIZE{
  for (j=0;j<SIZE;j++){
   a[i][j]= (float)i+j;
   b[i][j]= (float)i-j;
   c[i][j]= 0;
  }
 }
}

/* multiplicação de matrizes */
task spec matmul();
task body matmul()
{
 int i,j,k;

 forall i=0 to SIZE{
  for (j=0;j<SIZE;j++){
   for (k=0;k<SIZE;k++)
    c[i][j] += a[i][k]*b[k][j];
  }
 }
}

/* imprime resultado */
void print_mats()
{
 int i,j;

 for (i=0;i<SIZE;i++){
  for (j=0;j<SIZE;j++){
```

```
  printf("a[%d][%d]=%3.2f b[%d][%d]=%3.2f \n",i,j,
        a[i][j],i,j,b[i][j]);
  printf("c[%d][%d]=%3.2f \n",i,j,c[i][j]);
  }
 }
}

void main(){
 alloc_proc(7); /* 7 processos paralelos */
 create 7,init_matrix(); /*inicia matrizes paralelizando*/
 wait_proc(init_matrix); /* espera fim de init_matrix */
 create 7,mat_mul(); /*multiplica matrizes paralelizando*/
 wait_proc(mat_mul); /* espera fim de mat_mul */
 print_mats();
}
```

O Exemplo 2 mostra o uso de blocos paralelos e a hierarquia de memória compartilhada.

```
shared float result[500][500];
shared float res1[500][500];
shared float res2[500][500];
shared float res3[500][500];
task spec tarefa1();
task body tarefa1()
{
 shared float a[500][500];
 shared float b[500][500];
 int i,j,k;

 a[0][0]=1;
 a[0][1]=2;
 a[1][0]=1;
 a[1][1]=2;
 b[0][0]=4;
 b[1][1]=1;
 b[0][1]=5;
 b[1][1]=1;
 forall i=0 to 1 {
  for ( j=i+2;j<500;j=j+2)
   for (k=2;k<500;k++) {
    b[j][k]=a[j-2][k-2]*2 +b[j-2][k-2];
    a[j][k]=a[j-2][k-2]+b[j-2][k-2];
    res1[j][k]=a[j][k]+b[j][k];
   }
 }
}

task spec tarefa2();
task body tarefa2()
{
 int i,j,k;

 res2[0][0]=0;
 res2[0][1]=2;
 res2[1][0]=1;
```

```
res2[1][1]=2;
forall i=0 to 1{
 for (j=i+2;j<500;j=j+2){
  for (k=0;k<500;k++)
   res2[j][k]=res2[j-2][k-2]*2 +1;
 }
}
}

task spec tarefa3();
task body tarefa3()
{
 int i,j;

 forall i=0 to 499 {
 for (j=0;j<500;j++)
  result[i][j]=res1[i][j]*res2[i][j]+res3[i][j];
 }
}

void main()
{
 int i,j;

 alloc_proc(7);
 cobegin
  create 2,tarefa1();
  create 2,tarefa2();
  wait_proc(tarefa1);
  wait_proc(tarefa2);
  for (i=1;i<500;i++)
   for (j=1;j<500;j++)
    res1[i][j]=res1[i-1][j-1];
 also
  res3[0][0]=1;
  for (i=1;i<500;i++)
   for (j=1;j<500;j++)
    res3[i][j]=res3[i-1][j-1]*3-1;
 coend
 create 7,tarefa3();
}
```

8)Conclusão

Uma versão implementada utilizando apenas rotinas do sistema UNIX está instalada em uma máquina multiprocessadora com 8 processadores (SG4D da Silicon Graphics) e em estações SUN 3/60 e SPARC. Nas estações SUN o paralelismo é simulado pelo compartilhamento do tempo do processador, e o sistema pode ser utilizado no ensino ou pesquisa de programação paralela.

Em avaliações de desempenho realizadas na máquina paralela, obteve-se desempenhos significativos, atingindo em alguns casos, utilizando os 8 processadores, um ganho de 7.3 em relação a uma implementação seqüencial (Eficiência=0.91).

Está prevista a implementação de uma versão específica para o computador SG4D utilizando os recursos particulares de paralelismo desta máquina.

Ferramentas, tais como um sistema paralelizante de programas em CPAR, um depurador, e um sistema voltado para a especificação de projetos estão em andamento, e deverão ser componentes do ambiente de programação paralelo .

9) Bibliografia

[1] POLYCHRONOPOULOS, CONSTANTINE. D. "Parallel Programming Issues", Center for Supercomputing Research and Development, University of Illinois. CSRD Rpt 1004, junho, 1990.

[2] BAL, HENRY .E.; KAASHOEK,M. FRANS; TANENBAUM, ANDREW A. "Orca: a language for parallel programming of distributed systems", IEEE Transactions on software engineering, vol.18 n.3, março,1992

[3] GUARNA, J. VINCENT, "Extending the C language for parallel machines", CSRD Rp.t. N. 722, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign; 1988.

[4] CHANDRA, ROHIT; GUPTA, ANOOP; HENNESSY, JOHN L.; "COOL: a language for parallel programming"; in: "Languages and Compilers for Parallel Computing"; David A. Padua; Pitman Publishing; 1990; p.127-147.

[5] GEHANI, N. H.; ROOME, W. D.;"Implementing Concurrent C"; Software-Practice and Experience,v.22.p.265-285, março,1992.

[6] SNYDER, LAWRENCE; "The XYZ: abstraction levels Poker like languages"; in: "Languages and Compilers for Parallel Computing"; PADUA,David A.;Pitman Publishing;1990;p.470-489.

[7] POLYCHRONOPOULOS, CONSTANTINE D.; The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran; in: "Languages and compilers for parallel computing", GERLENTER,D.; Cambridge, MIT Press, 1990. p.423-453.

[8] BAUER, B. E.; "Practical parallel programming"; San Diego, Academic Press; 1992.

[9] SATO, LIRIA M. "CPAR: programação paralela em sistemas multiprocessadores", Anais da Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, março, 1991.

[10] SATO, Liria M.; MIDORIKAWA, Edson T. "Aspectos de Programação paralela", Anais da II Jornada EPUSP/IEEE em Sistemas de Computação de Alto Desempenho, 1992, maio.

[11] OUSTERHAUG, A.;"Guide to parallel programming on Sequent computer systems";2. ed.; Englewwod Cliffs,Prentice-Hall,1989.

[12] HARRISON, L.; AMMARGUELAT, Z. "A comparison of automatic versus manual parallelization of the Boyer-Moore Theorem Prover"; in: "Languages and Compilers for Parallel Computing"; PADUA,David A.; Pitman Publishing; 1991.