Extensões na LAM/MPI para Automatizar o Checkpoint e Tolerar Falhas em Cluster de Computadores

Antonio da Silva Martins Jr., Ronaldo A. L. Gonçalves
Universidade Estadual de Maringá - Departamento de Informática
Programa de Pós-Graduação em Ciência da Computação
Avenida Colombo, 5790, Zona 7, CEP 87020-900, Maringá - PR - Brasil
asmartins@uem.br; ronaldo@din.uem.br

Resumo

Os clusters de computadores estão se tornando cada vez mais comuns, em função do barateamento dos equipamentos e do potencial de processamento que eles podem dispor para a execução de aplicações complexas. Com isso, o desenvolvimento de técnicas de tolerância a falhas torna-se fundamental para garantir alto desempenho com confiabilidade. Em clusters com LAM/MPI (Local Area Multicomputer MPI), mecanismos de checkpoint disponíveis permitem a recuperação do estado seguro da aplicação após a ocorrência de falhas no sistema, mas são dependentes de intervenções do usuário.

O presente trabalho propõe a automatização tanto do checkpoint quanto da recuperação durante a ocorrência de falhas em um dos nós, provendo confiabilidade com praticidade ao sistema. As alterações necessárias no código da LAM/MPI são aqui apresentadas. Os resultados experimentais mostram que a perda de tempo causada pela ocorrência de falhas pode ser reduzida significativamente e de forma transparente para o usuário. Nos testes realizados com aplicação de cálculo matricial, a automatização pode prover uma redução de 55% no tempo total de execução da aplicação, quando um nó do cluster falhar após a execução de 90% do tempo de execução normal sem falhas.

1. Introdução

Problemas cada vez mais complexos surgem na medida em que as áreas do conhecimento humano avançam. Para determinado problema, esta complexidade é normalmente medida pela quantidade de operações necessária para solucioná-lo. Assim, quanto maior a complexidade, maior o tempo de resolução. Como exemplos, temos os cálculos para previsão climatológica, análise de imagens de satélites, simulações aerodinâmicas, reconhecimento de padrões, análise sísmica e o seqüenciamento de genoma.

Para fazer frente a esta demanda, o processamento paralelo pode ser usado eficientemente, dividindo as aplicações complexas em um conjunto de tarefas menores e mais simples, de forma que estas possam ser executadas simultaneamente por processadores independentes. Porém, a construção de máquinas paralelas específicas tem alto custo, cerca de um milhão de dólares por gigaflop de processamento. Por este motivo os clusters [1] do tipo Beowulf [2] têm sido usados, onde computadores comuns e relativamente baratos são conectados de forma a cooperar na solução de um mesmo problema. Neste modelo de computação, quanto maior a quantidade de nós, maior o potencial de processamento paralelo.

Entretanto, com o aumento da quantidade de nós, a queda da confiabilidade causada por falhas é quase proporcional, o que a torna um fator limitante na escalabilidade desses sistemas [3]. Para reduzir este problema, mecanismos de *checkpoint* são utilizados para salvar o contexto de uma aplicação em execução para posteriormente recuperá-lo em caso de falhas. Porém, em muitos ambientes MPI, esses mecanismos dependem do auxílio do usuário, tanto para a ativação do *checkpoint* como para a recuperação do processamento, como é o caso da LAM/MPI.

O presente trabalho, iniciado em [22], propõe a automatização tanto das operações de *checkpoint* quanto da recuperação do processamento em ambientes com LAM/MPI, de forma transparente para o usuário. Para isto, alterações foram feitas em algumas das funções da biblioteca LAM/MPI, inclusive com correções de código. A metodologia utilizada e os resultados obtidos são aqui mostrados.

Este texto está organizado da seguinte forma. A Seção 2 apresenta uma visão geral sobre o mecanismo de *checkpoint*. A seção 3 descreve as características do padrão MPI, enfocando a implementação LAM/MPI. A seção 4 apresenta a proposta e mostra como ela foi implementada. A seção 5 descreve os testes experimentais com aplicação matricial paralela e mostra os resultados. As conclusões e as referências bibliográficas estão nas seções 6 e 7, respectivamente.

2. O mecanismo de *checkpoint*

O objetivo do *checkpoint* é estabelecer um ponto de recuperação durante a execução de um programa e salvar informação suficiente do contexto da aplicação, para posteriormente restaurá-la em caso de falha, minimizando a perda de trabalho [4, 5]. Os algoritmos de *checkpoint* são usados em muitas aplicações, entre elas a tolerância a falhas, migração de processos, balanceamento de carga, troca de tarefas e depuração de erros [4, 6, 7].

Durante a sua execução, o contexto de um processo é composto pelos conteúdos da memória, registradores do processador e pelo contexto do SO (incluso o sistema de arquivos). Normalmente, a memória é dividida em quatro partes: código, variáveis, *heap* e pilha. O código normalmente não é armazenado, pois pode ser recuperado do arquivo de origem.

Se o sistema de *checkpoint* é implementado no nível do SO, as informações restritas do SO, necessárias a execução da aplicação, já estarão de posse do próprio mecanismo durante a restauração do estado anterior. Entretanto, se o mecanismo é implementado no nível de usuário, as informações restritas do SO devem ser salvas juntas com as demais informações do estado para serem restauradas na recuperação [4, 6]. Além das estruturas internas do SO e do sistema de arquivos, outras partes do ambiente externo que podem ser restauradas incluem a interface (gráfica e de texto) e o contexto de servidores externos. Muitos sistemas de *checkpoint* fornecem primitivas ao programador, permitindo o controle das informações que estão na memória global, na pilha e no *heap*, mas o programador continua responsável pela reconstrução do contexto de execução do programa na recuperação.

O *checkpoint* em sistemas de memória distribuída é mais complexo, pois é necessário garantir o estado global consistente do sistema para uma correta recuperação. A Figura 1 exemplifica esta situação. Nela, as linhas horizontais representam os tempos independentes de execução dos processos A, B e C, os pontos marcados com estrelas são pontos de *checkpoints* associados a cada processo e as setas de m1 a m4 representam mensagens transmitidas em uma rede de interconexão, sendo esta a única forma de comunicação entre os processos.

Observa-se que nem todo conjunto de *checkpoints* garante a recuperação de um estado global consistente. Por exemplo, o conjunto de *checkpoints* composto por A2, B2 e C2 não é consistente devido à mensagem m2, pois se a execução for restaurada nestes pontos, o processo B enviará novamente m2 que já havia sido recebida pelo processo C. Já o conjunto de *checkpoints* composto pelos pontos A3, B3 e C2 é consistente, pois nenhuma mensagem será perdida ou duplicada.

Na Figura 1, cada processo cria *checkpoints* periódicos, sem coordenação ou de forma independente. Isto dificulta ou até inviabiliza a restauração de um estado global consistente, podendo levar ao chamado efeito dominó, tal como segue. Supomos que cada processo armazene múltiplos pontos de *checkpoints* para precaver a ocorrência de diferentes falhas. Então, na ocorrência de falha, o sistema tentará recuperar-se a partir dos últimos pontos de *checkpoints*, mas isto pode não ser possível. Neste caso, o sistema fará outra tentativa de recuperação a partir de pontos de recuperação anteriores e caso não consiga novamente, as tentativas continuarão sucessivamente até que, na pior das hipóteses, retorne ao ponto inicial da execução, por não ter encontrado nenhum conjunto de pontos consistente, ocasionado pela falta de coordenação dos *checkpoints* [6, 8]. De maneira geral, existem duas formas de tratar o efeito dominó: *checkpoint* coordenado ou *checkpoint* com *log* de mensagens.

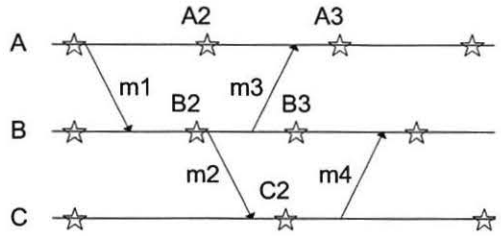

Figura 1. *Checkpoint* na execução multiprocessada

No *checkpoint* coordenado, os processos cooperam para salvar um conjunto de *checkpoints*, de forma síncrona, garantindo que as mensagens recebidas também sejam salvas durante os pontos de *checkpoint*. Em caso de falha, os processos são restaurados a partir do conjunto mais recente de *checkpoints*, composto pelo último *checkpoint* de cada processo. Isto garante que as próximas mensagens sejam naturalmente reenviadas e que as mensagens anteriormente salvas sejam naturalmente recuperadas. Uma vantagem neste modelo é que qualquer falha força todos os processos a usarem apenas o *checkpoint* mais recente, reduzindo os gastos de salvamento. Uma outra vantagem é que o algoritmo de recuperação de uma falha é trivial, conforme visto. Outros modelos requerem algoritmos complexos para recuperar o estado [6, 9, 10, 11].

No *checkpoint* com *log* de mensagens, o algoritmo assume que os programas em execução sejam parcialmente determinísticos, isto é, dado um estado inicial e um conjunto ordenado de mensagens, o programa irá sempre se comportar de uma mesma forma. Assim, enquanto um processo armazenar todas as mensagens recebidas, desde o último *checkpoint*, ele poderá reconstruir qualquer estado. Algoritmos de *log* de mensagens utilizam esta característica permitindo o *checkpoint* independente dos processos [6, 8].

3. O padrão MPI

O padrão MPI foi desenvolvido para aplicações

baseadas em troca de mensagens e as suas implementações formam a base para muitos sistemas paralelos de grande escala. Entretanto, ele não foi projetado para ser tolerante a falhas e as implementações mais usadas do MPI não adicionaram essa facilidade [3].

Para tentar resolver essa limitação, alguns projetos foram desenvolvidos com o intuito de acrescentar tolerância a falhas, baseada em *checkpoint*, em algumas implementações existentes do padrão MPI. **COCHECK** [12], **CLIP** [13], **MPICH-V** [19, 20] e **LAM/MPI-CR** [3, 7] são alguns exemplos.

3.1. A implementação LAM/MPI

A LAM/MPI (*Local Area Multicomputer MPI*) [3] utiliza um pequeno *daemon* de nível de usuário para o controle de processos, principalmente comunicação e redirecionamento de mensagens entre processos. Este *daemon* (*lamd*) é carregado no início de uma sessão, através do aplicativo *lamboot*. O *daemon* também auxilia na carga da aplicação, a qual é feita com o comando *mpirun*. Os diversos *daemons lamd* em conjunto formam a base do ambiente de execução LAM/MPI [16, 17, 18].

A interface com o usuário é constituída por aplicativos de linha de comando, os quais são responsáveis pelas diversas ações necessárias à criação e execução de aplicações distribuídas. Os três principais aplicativos são *lamboot* (responsável pela carga do *daemon lamd* e inicialização do ambiente de execução), *wrappers* de compilação mpicc, mpiCC/mpic++ e mpif77 (responsáveis pela execução do compilador e ligação do programa à biblioteca MPI, bem como pelo fornecimento de suporte às linguagens C, C++ e Fortran, respectivamente) e *mpirun* (aplicativo responsável pela carga e execução da aplicação).

O ambiente de execução de aplicativos distribuídos na LAM/MPI pode ser exemplificado na Figura 2. Nele, o aplicativo MPI é composto por três processos em execução em um cluster ideal formado por três nós. Sabe-se que a LAM/MPI cria um cluster lógico sobre os nós físicos onde são carregados os *daemons lamd*. O aplicativo MPI é executado sobre esse cluster lógico. Cada nó físico, ao executar uma cópia do *daemon lamd*, torna-se um nó lógico do cluster LAM/MPI.

O aplicativo *mpirun* é executado no nó mestre. Durante a carga dos processos da aplicação é possível selecionar o módulo MPI, que implementa as primitivas como *send* e *recv*, e o módulo RPI (*Request Progression Interface*) [20], que implementa a camada básica de comunicação dependente de hardware (TCP, *Myrinet*, SMP) ou de software (*checkpoint*) utilizados.

Ainda na Figura 2, nota-se a presença dos módulos CRMPI e CRLAM, que compõem o módulo RPI *crtcp* responsável pela implementação das rotinas de *checkpoint* tendo como base o protocolo de comunicação TCP/IP. O componente CRMPI implementa as primitivas de comunicação que são preparadas para o *checkpoint*, enquanto o componente CRLAM é responsável pela coordenação entre os processos da aplicação MPI durante os procedimentos de *checkpoint* e recuperação.

O *daemon lamd* também fornece um recurso de tolerância a falhas através de uma rotina de *heartbeat* que testa a conectividade entre os nós do cluster, reconfigurando o ambiente no caso de falha de comunicação de algum nó.

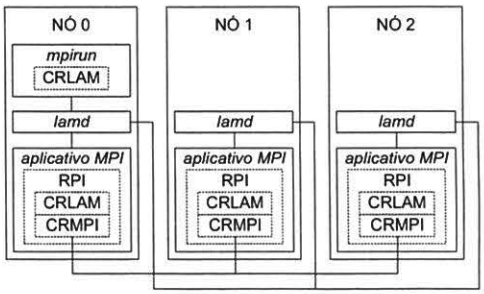

Figura 2. Um aplicativo distribuído na LAM/MPI

3.2. O *checkpoint* na LAM/MPI

A implementação de *checkpoint* utilizada pela LAM/MPI [3] vem com o pacote BLCR (Berkeley Labs *Checkpoint/Restart*) [7], contendo os comandos *cr_checkpoint* e *cr_restart*, que também podem ser utilizados pelo usuário para executar o *checkpoint* da aplicação. Sua API definida em [21].

O procedimento de *checkpoint* é bastante simples e a seqüência de passos de seu algoritmo pode ser resumida da seguinte forma:

1. **usuário** – executa o comando *cr_checkpoint* passando o *pid* do processo *mpirun*.
2. **mpirun** – recebe (via CRLAM) a solicitação do usuário.
3. **mpirun** – propaga a solicitação para cada processo MPI (via CRMPI)
4. **cada processo MPI** – negocia com os outros processos para atingir um estado global consistente.
5. **cada processo MPI** –executa também o comando *cr_checkpoint* passando o seu próprio *pid*.
6. **cr_checkpoint** – salva o contexto de cada processo
7. **cada processo MPI** – continua a execução normal após o retorno da chamada a *cr_checkpoint*.
8. **mpirun** – se prepara para uma possível recuperação e indica que está pronto para o *cr_checkpoint*.
9. **cr_checkpoint** – salva o contexto do *mpirun*.
10. **mpirun** – continua a execução normal após o retorno da resposta a *cr_checkpoint*.

De forma análoga, a seqüência de passos do algoritmo para o processo de recuperação pode ser descrita

conforme segue:

1. **usuário** – executa o comando *cr_restart* passando o nome do arquivo de contexto gerado pelo *cr_checkpoint*.
2. **mpirun** – executa a si mesmo (via *execve*), passando como um arquivo de configuração preparado por ele próprio para solicitar a recuperação de todos os processos MPI.
3. **mpirun** – reinicia a execução a partir do seu contexto recuperado, lê o arquivo de configuração e solicita a re-execução dos processos MPI através do comando *cr_restart* acrescido do nome do arquivo de contexto gerado pelo *cr_checkpoint*.
4. **cada processo MPI** – reinicia a execução a partir do seu contexto recuperado.
5. **cada processo MPI** – envia suas "novas" informações de processo (estrutura _gps) para o *mpirun*.
6. **mpirun** – constrói a tabela global de informações e propaga para todos os processos MPI.
7. **cada processo MPI** – recebe as informações sobre os outros processos MPI.
8. **cada processo MPI** – reconstrói os canais de comunicação com os outros processos MPI.
9. **cada processo MPI** – continua a execução normal.
10. **mpirun** – continua a execução normal.

Pode-se notar nesses algoritmos a presença de um tratador de *checkpoint*. A implementação utilizada pela LAM/MPI [3] vem com o pacote de *checkpoint* do linux BLCR (Berkeley Labs *Checkpoint/Restart*) [7], contendo os aplicativos *cr_checkpoint* e *cr_restart*, que também podem ser utilizados pelo usuário para executar o *checkpoint* da aplicação.

4. A proposta de *checkpoint* automático

O objetivo deste trabalho é a implementação de um mecanismo de *checkpoint* automático na LAM/MPI, a partir do mecanismo básico existente descrito na seção 3.2. Este mecanismo propõe gerar automaticamente pontos de recuperação (*checkpoints*) em intervalos de tempo pré-definidos pelo usuário e nos momentos onde a aplicação MPI esteja executando chamadas a operações coletivas do protocolo MPI [19]. As principais funcionalidades inseridas na LAM/MPI são relacionadas a seguir e explicadas nas próximas seções

1. Geração automática de *checkpoint*;
2. Detecção de falhas e recuperação automáticas.
3. Migração automática de processos;

4.1. Geração automática de *checkpoint*

A implementação da automatização do *checkpoint* utiliza uma abordagem baseada em *threads*, onde as principais funcionalidades a serem acrescidas na LAM/MPI são atribuídas a duas *threads* criadas durante a execução: *cr_ckpt_time()* e *cr_ckpt_coll()*. O aplicativo *mpirun* foi levemente modificado para disparar as *threads* pouco antes de entrar no modo de espera pelo resultado da aplicação MPI. Os algoritmos simplificados destas *threads* são mostrados na Figura 3.

A *thread cr_ckpt_time()* dispara o comando cr_*checkpoint*, em intervalos definidos pelo usuário. A rotina *cr_ckpt_coll* dispara o comando cr_*checkpoint* após a execução de funções MPI que envolvem sincronização, como as operações coletivas implementadas no módulo CRMPI: *MPI_Barrier*, *MPI_Scatter* e *MPI_Gather*, que também tiverem que ser adaptadas. O uso destas duas *threads* garante um conjunto de *checkpoints* consistente na LAM/MPI.

```
cr_ckpt_time():
ckpt_time = now + intervalo;
enquanto verdadeiro faça
  aguarde(ckpt_time - now);
  requisita(mutex);
  se ckpt_time <= now
  então {não foi feito pelo cr_ckpt_coll}
    cr_checkpoint; //isto gera um atraso
    ckpt_time = now + intervalo + atraso;
  {senão já foi feito pelo cr_ckpt_coll}
  fim-se;
  libera(mutex);
fim-enquanto;

cr_ckpt_coll():
enquanto verdadeiro faça
 se recv(sol checkpoint)
 então
   requisita(mutex);
   se (ckpt_time - atraso) <= now
   então {decorreu intervalo sem checkpoint}
     cr_checkpoint; //isto gera um atraso
     ckpt_time = now + intervalo + atraso;
   {senão ainda não decorreu intervalo}
   fim-se;
   libera(mutex);
 fim-se;
fim-enquanto;
```

Figura 3. Algoritmos *cr_ckpt_time* e *cr_ckpt_coll*

A modificação das rotinas do CRMPI consiste na inclusão de uma chamada a primitiva *send* para o envio de uma solicitação de *checkpoint* endereçada a *thread cr_ckpt_coll()*. Esta chamada é inserida no local da rotina coletiva em que todos os processos envolvidos estão em sincronismo aguardando o término da função. Por exemplo, na função *MPI_Barrier* original, o processo raiz, o de *rank* zero, é o responsável pelo sincronismo. Todos os demais processos enviam mensagens para ele e permanecem bloqueados aguardando uma resposta. Quando cada processo enviou a sua respectiva mensagem, ou seja, executou uma chamada a função *MPI_Barrier*, o processo raiz responde a todos enviando mensagens que

os desbloqueiam.

Nas versões adaptadas das funções coletivas, antes de enviar a mensagem de desbloqueio (conclusão da operação) é enviada uma solicitação de *checkpoint*, justamente no momento em que todos os processos estão parados aguardando o final da execução da operação coletiva.

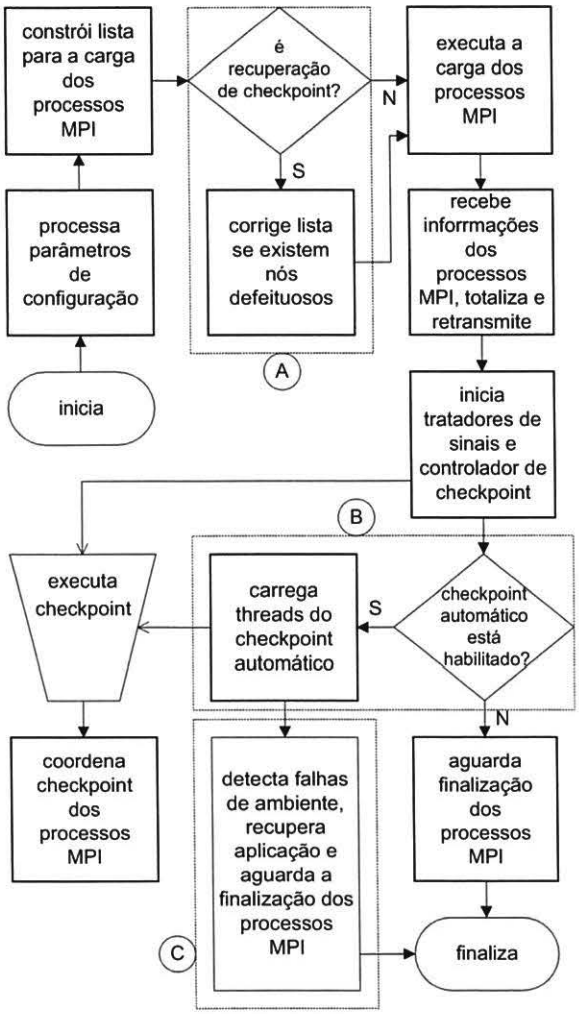

Figura 4. Alterações no aplicativo *mpirun*

A cooperação entre as *threads* de *checkpoint* é obtida através de controles de sincronização por *mutex*, os quais garantem a impossibilidade da execução de *checkpoints* simultâneos, bem como a atualização das variáveis de controle de tempo através de acesso a memória compartilhada. Uma heurística atualiza o intervalo entre os *checkpoints*, que são re-calculados utilizando o tempo médio gasto no *checkpoint* (atraso) adicionado ao intervalo solicitado. O momento de ativação destas *threads* no aplicativo *mpirun* pode ser visto na área "B" no diagrama da Figura 4. Esta figura mostra o algoritmo simplificado do *mpirun*, onde as seções destacadas indicam os locais das principais modificações realizadas neste trabalho.

4.2. Detecção de falha e recuperação automáticas

Para a implementação da detecção de falhas e recuperação automáticas, foi criada uma rotina no aplicativo *mpirun* que reconhece falhas dos nós do cluster através de consultas ao *daemon lamd*, que deve estar executando no modo *fail-safe* (detecção de falhas através de *heartbeat*). Esta rotina, chamada *cr_wait*, substitui a rotina original (*rpwait*), a qual aguarda a finalização dos processos em uma chamada à função de comunicação *nrecv* bloqueante.

A rotina *cr_wait* substitui a espera por suspensão, causada pela primitiva *nrecv* bloqueante da função *rpwait*, por uma espera semi-ocupada (laço com *sleep*) onde é usada a primitiva *nprobe* para testar a existência da mensagem. Se a mensagem não existe é iniciado uma consulta ao *daemon lamd* que, em caso de constatação de alterações no ambiente, encerra os processos restantes e reinicia o aplicativo *mpirun* para os nós restantes através da chamada a função C *execve*. A adição desta rotina ao aplicativo *mpirun* pode ser observada no diagrama da Figura 4 na área "C".

4.3. Migração automática de processos

No momento em que o processo MPI conecta-se ao ambiente de execução da LAM/MPI, durante a execução da função *MPI_Init*, ele cria um canal de comunicação via *pipe* com o *daemon lamd*. A localização e o nome do canal de comunicação são dependentes do nó físico em que o processo está sendo executado. Durante a operação de *checkpoint*, o nome do *pipe* é armazenado no contexto da aplicação por onde a comunicação é restabelecida. Desta forma, no caso de uma migração do processo entre nós físicos, mesmo encontrando-se no mesmo nó lógico, o nome do *pipe* recuperado se torna incorreto, não permitindo assim restabelecer a comunicação. A correção efetuada atualiza o nome do *pipe* antes de restabelecer a comunicação, gerando assim o nome correto em relação ao nó físico corrente. Este problema não estava relatado em [3].

Outras modificações maiores nas rotinas de recuperação do módulo CRLAM foram feitas para prover a migração entre os nós lógicos. Cada processo MPI guarda informações sobre todos os processos da aplicação em uma lista, representada pela estrutura *_proc*, onde a estrutura *_gps* atua como chave de busca. A estrutura *_gps* é composta pelo índice do nó lógico (*gps_node*), o índice do processo no ambiente Unix (*gps_pid*), o índice do processo no nó lógico (*gps_idx*) e pelo índice geral do processo na aplicação MPI (*gps_grank*).

A atualização das informações de localização foi

implementada na função de recuperação de processos da CRLAM do módulo *crtcp*. Utilizando as informações do ambiente e do processo, repassadas após a conexão, os dados referentes ao nó em que o processo se encontra no momento, são atualizadas na estrutura *_proc*, permitindo assim, que o restabelecimento dos canais de comunicação entre os processos seja realizado com sucesso, no caso de qualquer processo ter sido migrado.

A rotina *lb_fault_app* foi implementada no aplicativo *mpirun* para remanejar os processos que estavam em nós não mais existentes para entre os nós restantes. O ponto de inserção desta rotina é representado pela área "A" no diagrama da Figura 4.

5. Resultados experimentais

Com o objetivo de comprovar a eficiência da automatização do *checkpoint* e da tolerância a falhas, experimentos foram realizados em um cluster composto por 5 computadores Pentium de 200Mhz com 96MB de memória RAM, sendo que o nó mestre estava equipado com unidades de disco SCSI. Os nós escravos iniciam o sistema operacional remotamente via rede *Fast-Ethernet* e utilizam os discos do servidor via NFS.

Os testes foram feitos com uma aplicação que multiplica 200 vezes duas matrizes A e B, ambas de dimensão 500x500 e inicializadas com valores arbitrários, cujo algoritmo simplificado é mostrado na Figura 5. Cada experimento foi executado cinco vezes e o tempo médio foi considerado como resultado final.

Nesta aplicação, o processo mestre primeiramente inicializa as matrizes A e B usando a função *initMatrix*. Após, envia a dimensão das matrizes e a matriz B aos processos escravos, usando a função *MPI_Broadcast*. E uma parte diferente da matriz A, a cada processo escravo, usando a função *MPI_Send*, para que cada processo faça uma parcela da multiplicação total, dividindo desta forma o problema entre todos os processos. Os processos escravos, após receberem os dados usando as funções *MPI_Broadcast* e *MPI_Recv*, efetuam a multiplicação, em conjunto com o processo mestre, e retornam o resultado parcial ao processo mestre, que então totaliza os resultados na matriz C.

Para analisar o efeito da ocorrência de falhas no ambiente de execução, um dos nós escravos era desligado durante a execução da aplicação. A falha era induzida após a aplicação já ter sido executada uma porcentagem do seu tempo total previsto de execução normal. Desta forma, diferentes falhas foram ocasionadas após diferentes porcentagens de tempo decorrido. O objetivo foi o de identificar como reage o *checkpoint* automático nos diferentes momentos da execução.

Cada coluna da Figura 6 representa o tempo total de execução da aplicação por completo, obtido em determinada situação, a saber: 1) coluna "Sem CR": sem *checkpoint* e sem ocorrência de erro, ou seja, uma execução normal bem sucedida na LAM/MPI; 2) coluna "Com CR": com *checkpoint* automático ativado (periódico em intervalos de 1000 segundos e durante as operações coletivas) e sem ocorrência de erros, ou seja, uma execução normal bem sucedida na LAM/MPI com *checkpoint*; 3) colunas X%: com *checkpoint* automático ativado e com falha induzida após a execução da aplicação ter decorrido aproximadamente X% do seu tempo total normal, com X variando de 10 a 90, em intervalos de 10.

```
se rank == 0 //executa no mestre
então
  leia n
  MPI_Brodcast(n) //envia n
  A=createMatrix(n,n)
  B=createMatrix(n,n)
  C=createMatrix(n,n)
  A=initMatrix(n,n)
  B=initMatrix(n,n)
  m = n/p
  o = m
  MPI_Broadcast(B) //envia B
  para i=1 até p-1 faça
    MPI_Send(A.rows[o],i) //envia A
    o += m
  A.m = m
  doMatrixMul(A,B,C) //multiplica
  o = m
  para i=1 até p-1 faça
    MPI_Recv(C.rows[o],i) //recebe C
    o += m
senão //executa nos escravos
  MPI_Brodcast(n) //recebe n
  m = n/p
  A=createMatrix(m,n)
  B=createMatrix(n,n)
  C=createMatrix(m,n)
  MPI_Broadcast(B)    //recebe B
  MPI_Recv(A.data,0) //recebe A
  doMatrixMul(A,B,C) //multiplica
  MPI_Send(C.data,0) //envia C
fim-se

função doMatrixMul(A,B,C)
 para i=0 até A.m faça
  para j=0 até A.n faça
   soma = 0
   para k=0 até A.n faça
    soma += A.rows[i][k]*B.rows[k][j]
    C.rows[i][j]=soma
```

Figura 5. Multiplicação paralela de matrizes

Note que o gráfico não mostra os tempos das aplicações em situações de falhas onde não existe o mecanismo de *checkpoint*, pois é notório que, nestes casos, na melhor das hipóteses (caso o operador da aplicação perceba imediatamente a ocorrência da falha e reinicie a aplicação sem perda de tempo), a aplicação

deverá ser reiniciada e executada novamente por completo. Neste caso, o tempo total de execução será a somatória do tempo decorrido até a ocorrência da falha acrescido do tempo de execução normal por completo. Este tempo, subestimado é claro (visto que o operador pode perceber a ocorrência da falha somente na próxima semana), foi considerado para efeito de cálculo do ganho com o uso de *checkpoint*.

Nos casos de falha com *checkpoint*, o processamento é concluído sem a presença do nó falhado, reativando o processo do nó falhado em um dos nós escravos ainda remanescentes. Os tempos de execução da aplicação sem *checkpoint* (5.681 segundos) e com *checkpoint* ativado (5.784 segundos) mostram que a utilização de *checkpoint* automático, conforme apresentado, acarreta um acréscimo de somente 1,81% no tempo total de execução da aplicação, tempo este desprezível quando da ocorrência de falha, conforme explicado a seguir.

A Figura 6 mostra que quanto maior é o tempo já executado pela aplicação, quando uma falha ocorre, maior é o benefício do *checkpoint* automático, pois menor é o tempo restante para a conclusão da aplicação. Por exemplo, considere a ocorrência de uma falha após 90% de execução normal já decorrida. Sem *checkpoint*, a aplicação levará 10.794 segundos para ser executada com uma re-execução em 4 nós (quase o dobro do tempo de uma execução normal). Com *checkpoint* automático, que completará a execução utilizando apenas 3 nós após a falha (já que o nó com falha não pode ser reaproveitado) a aplicação levará 6.085 segundos. Assim, tem-se um ganho acima de 43% na redução do tempo de execução. Se for considerado que a re-execução da aplicação sem *checkpoint* é feita em apenas 3 nós, que é bem mais realista, o ganho atinge patamares em torno de 55%.

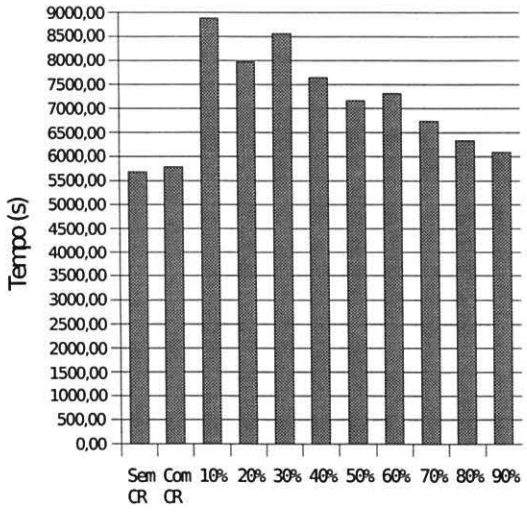

Figura 6. Tempos de execução

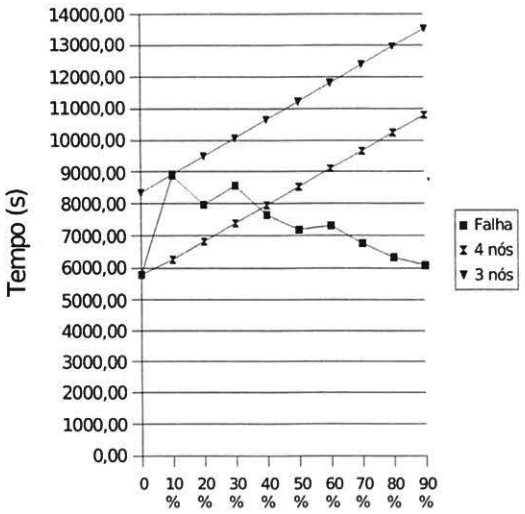

Figura 7. Desempenho de recuperação

Esta vantagem é mais perceptível na Figura 7, que mostra uma outra visão dos resultados obtidos através de 3 curvas, onde o eixo y indica a variação do tempo total de execução e o eixo x indica as porcentagens de tempo decorrido da execução da aplicação no momento da ocorrência de uma falha em um dos nós do cluster. As duas curvas retilíneas ascendentes representam o comportamento da aplicação sem *checkpoint* em um cluster de 3 e de 4 nós, respectivamente, com uma re-execução após a falha. A outra curva representa o comportamento da aplicação executada em 4 nós com *checkpoint* automático e continuação da execução em 3 nós após a falha.

6. Conclusões e trabalhos futuros

O presente trabalho apresenta a implementação de um mecanismo que automatiza a execução de *checkpoints* periódicos e durante as operações coletivas sincronizadas, bem como a recuperação em caso de falhas de hardware em um nó do cluster.

Os experimentos realizados mostram que o incremento no tempo de processamento causado pela adição do mecanismo proposto, o qual é diretamente proporcional ao tamanho da aplicação e o período definido para o *checkpoint*, pode ser desprezível se comparado como a economia de tempo proporcionada pela não reinicialização da aplicação. Nos experimentos aqui realizados esta economia de tempo atingiu valor em torno a 55%.

É importante ressaltar que o benefício do *checkpoint* automático foi alcançado delo fato do mesmo ser periódico, pois o *checkpoint* realizado nas operações coletivas somente serviu para garantir a consistência da restauração das mensagens trocadas nestas operações, que

na aplicação experimentada eram esporádicas. De qualquer forma, a utilização de *checkpoint* automático na LAM/MPI é comprovadamente uma solução importante para garantir confiabilidade ao alto desempenho.

Na continuidade deste trabalho, o mecanismo será testado em clusters maiores, na execução de aplicações que executam maior volume de troca de mensagens em funções coletivas. Além disso, o mecanismo será combinado com outro trabalho que visa o desenvolvimento de técnicas de balanceamento de carga em clusters MPI.

7. Referências

[1] BUYYA, R. "High Performance Cluster Computing: Architectures and Systems". V.1. N.J. Prentice-Hall, 1999.

[2] STERLING, T. "Beowulf Breakthroughs – The Genesis of Linux Clusters in High Performance Computing". Linux Magazine. Jun. 2003.

[3] SANKARAN, S. et al. "The LAM/MPI *Checkpoint*/ Restart Framework: System-Initiated *Checkpoint*". In: Proceedings of LACSI Symposium. Santa Fé, USA. 2003.

[4] WANG, Y-M.; et al. "*Checkpoint*ing and its Applications". In: 25th International Symposium on Fault-Tolerant Computing, Pasadena. 1995.

[5] PLANK, J. S.; et al. "Libckpt: Transparent *Checkpoint*ing under Unix". In: Usenix Winter 1995 Technical Conference, New Orleans, Jan 1995.

[6] PLANK, J. S. "An Overview of *Checkpoint*ing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance". University of Tennessee, TR UT-CS-97-372., Jul. 1997.

[7] DUELL, J.; HARGROVE. P. & ROMAN, E. "The Design and Implementation of Berkeley Lab's Linux *Checkpoint*/Restart". Berkeley Lab, TR LBNL-54941, 2003.

[8] ELNOZAHY, E. N.; JOHNSON, D. B. & WANG, Y. M. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems". C. Mellon University, TR CMU-CS-96-181,. 1996.

[9] MANIVANNAN, D.; NETZER, R. H. B.; SINGHAL M. "Finding Consistent Global *Checkpoints* in a Distributed Computation". In: IEEE Transactions on Parallel and Distributed Systems, vol. 8, n. 6, p. 623-627. 1997.

[10] NEVES, N. & FUCHS, W. K. "Coordinated *Checkpoint*ing without Direct Coordination". In: Proceedings of IEEE International Computer Performance & Dependability Symposium, pp. 23-31, Sep. 1998.

[11] VAIDYA, N. H. "Staggered Consistent *Checkpoint*ing". In: IEEE Transactions on Parallel and Distributed Systems. vol. 10, n. 7. p. 694-702. 1999.

[12] STELLNER, G. CoCheck: *checkpoint* and Process Migration for MPI. In: Proceedings of the 10th International Parallel Processing Consortium (IPPS 96). p. 526-531. 1996.

[13] CHEN, Y.; PLANK, J. S. & LI, K. "CLIP: A *checkpoint*ing tool for message-passing parallel programs". Princeton University, TR-543-97, May 1997.

[14] LITZKOW, M. et al. "*Checkpoint* and Migration of UNIX Processes in the Condor Distributed System". 1997. <www.cs.wisc.edu/condor/doc/ckpt97.ps>. Acesso em: 20 mar. 2004.

[15] BOSILCA, G.; et al. "MPICHV: Toward a Scalable Fault Tolerant MPI for Volatile Nodes". In: Proceedings of IEEE SuperComputing 2002 (SC2002). Nov. 2002.

[16] LAM/MPI TEAM. "LAM/MPI Installation Guide version 7.1.1". Set. 2004. Disponível em: <www.lam-mpi.org/download/files/7.1.1-install.pdf>. Acesso em: 25 out. 2004.

[17] LAM/MPI TEAM. "LAM/MPI User's Guide version 7.1.1". Set. 2004. Disponível em: <www.lam-mpi.org/download/ files/7.1.1-user.pdf>. Acesso em: 25 out. 2004.

[18] SQUYRES, J. M.; BARRET, B.; & LUMSDAINE, A. "Boot System Services Interface Modules for LAM/MPI". TR576, CS, Indiana University. Aug. 2003.

[19] SQUYRES, J. M.; BARRET, B.; & LUMSDAINE, A. "MPI Collective Operations System Services Interface Modules for LAM/MPI". TR577, CS, Indiana University. Aug. 2003.

[20] SQUYRES, J. M.; BARRET, B.; & LUMSDAINE, A. "Request Progression Interface System Services Interface Modules for LAM/MPI". TR579, CS, Indiana University. 2003.

[21] SANKARAN, S. et al. "*Checkpoint*-Restart Support System Services Interface (SSI) Modules for LAM/MPI". Technical Report TR578, CSD, Indiana University. 2003.

[22] MARTINS JR, Antonio da Silva; GONCALVES, Ronaldo A. L. *Checkpoint* Automático em Cluster MPI: Testes Preliminares. In: VI FITEM - FÓRUM DE INFORMÁTICA E TECNOLOGIA DE MARINGÁ, 2004, Maringa. 2004.