UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
Departamento de Ciência da Computação

MAC5701 - Tópicos em Sistemas de Computação

Recuperação por Retrocesso para Aplicações BSP em Grades Computacionais

Raphael Yokoingawa de Camargo

Orientador: **Prof. Dr. Fabio Kon**

São Paulo, 25 de junho de 2004

1 Introdução

Sistemas distribuídos estão cada vez mais presentes em nossa sociedade. Um exemplo recente é o surgimento das grades computacionais que permitem o compartilhamento de recursos e conseqüentemente a utilização de recursos computacionais que de outro modo ficariam ociosos. Globus, Condor e InteGrade [GKG+04] são exemplos destes sistemas de grades computacionais.

Para que estas grades computacionais sejam utilizadas para pesquisa científica, é necessário que bibliotecas de programação sejam integradas a estas grades. No caso do InteGrade foi implementado parte de uma biblioteca para o modelo de programação paralela BSP [HMS+98, Val90]. Este modelo é baseado no conceito de superpassos. A execução é composta por um seqüência de superpassos, com uma fase de sincronização no final de cada. A comunicação entre processos ocorre apenas durante esta fase de sincronização. Deste modo, não existe troca de informações dentro de um mesmo superpasso.

Mas apesar das grades computacionais trazerem um grande ganho de poder computacional, surgem alguns problemas, como a maior propensão a falhas. No caso do InteGrade os nós que serão utilizados nas computações são tipicamente estações de trabalho. Estas podem sofrer diversos tipos de problemas, como travamentos, *reboots* e desconexões da rede. No caso de aplicações paralelas que possuem diversos processos sendo executados em diferentes máquinas este problema é ainda maior, dado que uma falha em qualquer um dos nós pode causar a falha da aplicação inteira. Mecanismos de tolerância a falha, como a recuperação por retrocesso [EAWJ02], visam resolver este tipo de problema.

2 Recuperação por Retrocesso

A **recuperação por retrocesso baseada em *checkpoints*** consiste em reiniciar a execução da aplicação a partir de um estado salvo durante sua execução, após a ocorrência de falhas em um ou mais de seus processos. Estes estados salvos são denominados *checkpoints*, e permitem que, no caso de falha em um de seus processos, a aplicação não precise retornar a seu estado inicial.

2.1 *Checkpointing* de aplicações paralelas

No caso de aplicações paralelas e distribuídas existe um problema extra que é a dependência entre os diversos processos que estão sendo executados simultaneamente. Esta dependência é gerada pela troca de informação entre processos. Um exemplo claro disto é no envio de uma mensagem de um processo A para um processo B. Após o processo B receber a mensagem enviada por A, passa a haver uma dependência entre ambos, pois para que o processo B tenha recebido a mensagem, é necessário que o processo A a tenha enviado.

Deste modo, não basta que cada processo simplesmente salve seu estado periodicamente de maneira independente, pois se escolhermos um conjunto de *checkpoints* contendo um *checkpoint* de cada processo, estes pode não formar um estado global consistente. No pior caso, é possível que a aplicação precise retornar a seu estado inicial, mesmo tendo salvo diversos *checkpoints*.

Existem diversas maneiras de se resolver este problema. O protocolo de *checkpointing* coordenado exige que todos os processos salvem seus estados de maneira coordenada, de modo que todos os estados globais gerados sejam consistentes. Isto garante que a aplicação sempre será reiniciada a partir do último *checkpoint* salvo, além de facilitar outros aspectos importantes como a coleta de *checkpoints* salvos obsoletos.

No outro extremo existe o protocolo não-coordenado, onde cada processo decide independentemente quando gerar seus *checkpoints*. Como já foi dito, o problema deste protocolo é que dependendo do caso

a aplicação pode precisar retornar ao seu estado inicial. A solução está em forçar os processos a salvar alguns *checkpoints* extras de modo que estados consistentes sejam formados. O objetivo é eliminar determinadas classes de dependências entre processos de modo que *checkpoints* globais consistentes possam ser gerados. Para tal, os processos participantes enviam informações extras junto com as mensagens trocadas, de modo que os receptores destas mensagens podem decidir se devem ou não gerar um *checkpoint* extra. Estes protocolos que forçam a geração de *checkpoints* extras são denominados protocolos induzidos pela comunicação.

No caso do modelo de programação BSP, o protocolo coordenado é a escolha natural, uma vez que neste modelo os processos precisam sincronizar sua execução a cada superpasso. Pode-se então utilizar esta etapa de sincronização para gerar os *checkpoints*. Para evitar que um número excessivo de *checkpoints* seja gerado, pode-se definir um intervalo de tempo mínimo entre *checkpoints*.

3 Salvando o estado de um processo

Existem duas abordagens possíveis para salvar o estado de um processo. Na primeira, o estado é salvo por uma *thread* ou processo independente que roda em paralelo com o processo original. Os dados do processo original são obtidos diretamente de seu espaço de memória. Esta abordagem é denominada *checkpointing* no nível do sistema. Na segunda abordagem, o código para salvar o estado do processo está presente no código-fonte da aplicação. Esta abordagem é denominada *checkpointing* no nível da aplicação.

3.1 *Checkpointing* no nível do sistema

Esta abordagem é utilizada em trabalhos como [PaGKL95, LTBL97] para que se possa salvar o estado da aplicação. A grande vantagem desta abordagem é ser totalmente transparente para o usuário. Além disso, ela é independente da linguagem de implementação. Sua principal desvantagem é que este mecanismo é naturalmente não portável, já que os dados são salvos diretamente da memória, sem nenhum tipo de informação semântica sobre estes dados. Além disso, esta abordagem normalmente resulta em *checkpoints* maiores, já que dados desnecessários podem ser salvos.

Existem diversas técnicas para melhorar o desempenho do *checkpointing*. A técnica de *checkpointing* incremental consiste em salvar no novo *checkpoint* apenas os dados que diferem do *checkpoint* anterior. Outra técnica consiste em não parar a execução da aplicação enquanto o novo *checkpoint* é gerado. Para tal é preciso garantir que a aplicação não altere dados que ainda não foram salvos no *checkpoint*. Outras técnicas consistem em permitir à aplicação algum controle sobre o processo de *checkpointing*. Um exemplo é a exclusão de trechos de memória que não precisam ser salvos no *checkpoint*.

3.2 *Checkpointing* no nível da aplicação

Esta abordagem consiste em adicionar diretamente ao código-fonte da aplicação linhas de código para que o programa passe a salvar seu estado periodicamente. Além disso, é inserido código para que a aplicação reinicie sua execução a partir do último *checkpoint* salvo em caso de reinicialização após uma falha.

A principal vantagem desta abordagem é que como informações semânticas sobre os dados da memória estão disponíveis, pode-se reduzir a quantidade de dados que são salvos. Além disso, é possível salvar estes dados de maneiro portável. O principal problema desta abordagem é a necessidade de se adicionar código extra à aplicação. Se feito diretamente pelo desenvolvedor, este é um processo bastante sucetível

a erros. A solução está em desenvolver um precompilador que realiza esta tarefa automaticamente. Esta abordagem foi adotada em trabalhos como [BMPS03, SR96, KBH01]

3.2.1 Salvando o estado da pilha de execução

A pilha de execução é formada pelas funções ativas em determinado momento da execução, junto com informações de controle das chamadas de funções e os valores dos parâmetros e variáveis locais destas funções.

Diferentemente do caso de *checkpointing* no nível do sistema, a pilha de execução não é diretamente acessível a partir do programa em C. Deste modo, o estado da pilha de execução deve ser armazenado e recuperado de modo indireto.

A técnica neste caso consiste em armazenar a lista de funções ativas no momento em que o *checkpoint* é gerado e os valores das variáveis locais e parâmetros destas funções. Para recuperar o estado da pilha na reinicialização da aplicação basta chamar estas funções na mesma ordem em que elas estavam quando o estado do processo foi salvo e ler os valores das variáveis locais do *checkpoint* armazenado.

3.2.2 Salvando o estado da memória dinâmica

Além das variáveis locais é necessário ainda salvar os dados contidos na memória alocada dinamicamente. Estes dados, que ficam armazenados numa área denominada *heap*, são acessíveis à aplicação através de ponteiros.

Quando o *checkpoint* é gerado, deve-se salvar os dados contidos no endereço apontado pelos ponteiros naquele momento. Para que isto seja possível, é necessário manter uma lista dos endereços de memória alocados e o tamanho da memória alocada para aquele endereço. Deste modo, a partir do endereço de memória contido no ponteiro, pode-se salvar todo o trecho de memória alocado dinamicamente.

Um caso especial é o de ponteiros compostos, onde um ponteiro aponta para um endereço contendo um outro ponteiro. Estes ponteiros compostos podem formar grafos dirigidos de ponteiros, que precisam ser salvos e posteriormente reconstruídos. Existem diversos problemas a serem tratados como o caso onde estes grafos contém ciclos ou quando dois ponteiros apontam para o mesmo endereço de memória.

3.2.3 Reinicialização da aplicação

Durante a reinicialização, a pilha de execução é reconstruída chamando as funções que estavam ativas no último *checkpoint* gerado na mesma ordem em que estas estavam. Além disso, é necessário declarar todas as variáveis locais novamente e recuperar seus valores do arquivo de *checkpoint*. O restante do código não deve ser executado, caso contrário, as computações realizadas antes da falha estarão sendo executadas novamente.

No caso da linguagem C, as declarações são feitas no início de cada bloco de código[1]. Uma vez que estes blocos de código podem estar aninhados, deve-se determinar quais variáveis estão no escopo da função a ser chamada e recuperar apenas os valores destas variáveis. Isto é bastante simples, dado que todas as variáveis declaradas em blocos externos ao da chamada da função estarão em seu escopo.

Quando passamos para o C++ esta situação se torna muito mais complicada, uma vez que variáveis podem ser declaradas em qualquer posição destes blocos. No caso em que existem diversos blocos de código aninhados, com declarações de variáveis e chamadas de funções espalhadas por este bloco, este processo se torna relativamente complexo.

[1]Neste trabalho chamamos de bloco de código um grupo de linhas de código localizados que pertencem a um mesmo escopo, por exemplo a implementação de uma função ou o código no interior de um laço

4 Implementação

Foi implementado um sistema de recuperação por retrocesso baseado em *checkpoints* para a biblioteca BSPLib do InteGrade. Este sistema é composto por dois módulos:

- *Precompilador:* Modifica um código fonte em C/C++, adicionando linhas de código extra para que o programa passe a salvar seu estado periodicamente. Além disso, é inserido código para que a aplicação reinicie sua execução a partir do último *checkpoint* salvo em caso de falha.
- *Biblioteca de* Checkpointing*:* Fornece uma API com funções que permitem que aplicações salvem seu estado em um arquivo e posteriormente recuperar este estado. Esta biblioteca possui ainda um timer para que se possa especificar um tempo mínimo entre diferentes *checkpoints* gerados.
- *Coordenador de* Checkpointing*:* Realiza a coordenação entre os processos de uma aplicação BSP de modo que estes gerem *checkpoints* globais consistentes. Além disso, monitora a execução dos processos e coordena a reinicialização da aplicação BSP em caso de falha em algum de seus processos.

Nas próximas subseções estes módulos são descritos com detalhe.

4.1 Precompilador

Para implementar o precompilador, foi utilizada a ferramenta OpenC++ [Chi95]. Esta é uma ferramenta que analisa um código-fonte C++, constrói uma árvore sintática e então gera código-fonte C++ novamente. A vantagem em se utilizar esta ferramenta é que é muito mais fácil manipular a árvore sintática do que o código-fonte diretamente.

4.1.1 Determinando as funções que precisam ser modificadas

Nem todas as funções da aplicação devem ser modificadas. Dados todos os possíveis estados da pilha de execução do processo no momento da geração de um *checkpoint*, o precompilador deve modificar apenas as funções que estão presentes na pilha de execução em pelo menos um destes estados.

Denominemos por ϕ o conjunto das funções que devem ser modificadas. Uma função $f \in \phi$ se e somente se:

1. f faz uma chamada à função checkpoint_candidate ou,
2. f faz uma chamada à uma função pertencente à ϕ.

O precompilador determina quais funções pertencem a *phi* de maneira recursiva. Inicialmente são adicionadas a este conjunto todas as funções que fazem chamadas à função checkpoint_candidate. Em seguida, em cada passo da recursão são adicionadas todas as funções que fazem chamadas a funções pertencentes ao novo conjunto ϕ. Este processo é repetido até que nenhuma nova função seja adicionada a ϕ.

4.1.2 Salvando o estado da pilha de execução

Para salvar as variáveis locais, utilizamos uma estrutura de dados do tipo pilha onde as variáveis locais são empilhadas. Sempre que uma nova variável entra no escopo durante a execução, ela tem seu endereço de memória adicionado à pilha. Quando ela sai do escopo ela é imediatamente retirada desta pilha. Uma variável entra no escopo quando é declarada e sai do escopo quando o bloco onde esta variável foi declarada termina ou quando um comando que sai do bloco, como return, break ou continue é executado.

É interessante notar que o armazenanto do endereço de memória da variável na pilha, ao invés de seu valor, garante que este endereço sempre conterá o valor atualizado da variável. Como a pilha contém o endereço de todas as variáveis locais que estão no escopo naquele momento, o conteúdo desta pilha é suficiente para recuperar o valor de todas as variáveis locais necessárias para a reinicialização da aplicação.

Para armazenar quais as funções estavam ativas no momento de gerar o *checkpoint*, o precompilador adiciona uma nova variável local currentGotoLabel às funções modificadas. O código é então modificado de modo que antes de realizar a chamada de alguma função pertencente a ϕ, o valor desta variável é modificado. Deste modo, a partir do valor desta variável é possível determinar qual função estava ativa no momento da geração do *checkpoint*.

4.1.3 Salvando o estado da memória dinâmica

No caso da memória dinâmica, foi implementado um gerenciador de memória denominado HeapController que armazena o endereço dos blocos de memória alocados e seus respectivos tamanhos.

O precompilador substitui as chamadas de sistema malloc, realloc e free por chamadas a funções equivalentes na biblioteca de *checkpointing*. Estas funções por sua vez passam os endereços de memória que estão sendo alocados ou liberados para o HeapController e então realizam as chamadas de alocação de memória na biblioteca do sistema.

No momento o precompilador fornece suporte apenas a ponteiros não-compostos. Numa futura versão o precompilador irá também fornecer suporte a ponteiros compostos. Além dos problemas descritos na seção 3, como loops e endereços de memória replicados, é também mais complicado colocar na pilha os endereços de memória a serem salvos. Por exemplo, no caso de uma variável do tipo int ** representando uma matriz de inteiros será necessário armazenar um número de endereços dependente da quandidade de memória alocada.

4.1.4 Variáveis globais e estruturas

Para salvar o estado de variáveis globais, basta adicioná-las à pilha de dados a serem salvos antes de qualquer outra variável local. Isto pode ser facilmente obtido modificando a função main() para empilhar as variáveis globais antes de suas variáveis locais.

Um caso que precisa ser tratado à parte são as estruturas. No caso em que esta possui apenas tipos de dados primitivos, basta salvar o conteúdo contido no endereço da variável que representa esta estutura. No caso em que esta contém ponteiros entre seus membros, é necessário salvar também o endereço de memória apontado por este ponteiro. No caso de classes pode-se utilizar uma abordagem semelhante ao de ponteiros, dado que o conteúdo a ser salvo correponde aos membros desta classe.

No momento o precompilador não fornece suporte para salvar automaticamente variáveis globais, estruturas e classes, mas este suporte já está sendo implementado.

4.1.5 Reinicialização da aplicação

Para reconstruir a pilha de execução deve-se chamar todas as funções que estavam ativas no momento que o *checkpoint* foi gerado. Para identificar quais funções devem ser chamadas, utiliza-se o valor da variável currentGotoLabel armazenado no último *checkpoint* gerado.

Além disso, é necessário declarar todas as variáveis locais que presentes no escopo da chamada da próxima função da pilha de execução. Deste modo, para cada possível função deve-se determinar quais variáveis estão em seu escopo e executar apenas as linhas de código correspondentes a estas variáveis.

Para determinar as variáveis pertencentes ao escopo de cada chamada de função, o precompilador cria uma lista com todos os blocos de código presentes na função contendo informações sobre a linha e o bloco onde estes blocos iniciam. Também é armazenada informações sobre o bloco e linha do bloco de cada chamada de função e declaração de variável. Utilizando esta informação é possível determinar para cada par (variável, função) se a variável pertence ao escopo desta função.

4.1.6 Chamadas da API BSP

No caso de aplicações paralelas BSP, é necessário modificar algumas das chamadas à biblioteca que implementa este modelo. Em particular, uma aplicação BSP normalmente precisa fazer chamadas às funções bsp_begin(), para inicializar a biblioteca BSP, e bsp_sync(), para fazer a sincronização entre os processos.

A idéia é o precompilador substitui estas chamadas por chamadas à funções equivalentes no coordenador de *Checkpointing*. Estas funções, denominadas bsp_begin_ckp() e bsp_sync_ckp() são responsáveis por inicializar e realizar a cordenação do processo de *checkpointing*.

Outro ponto que precisa ser tratado é que durante a reinicialização deve-se executar os chamadas à biblioteca BSP que realizam o registro de endereços de memória para a comunicação do tipo Direct Remote Memory Access (DRMA).

No estágio atual o desenvolvedor precisa manualmente realizar estas modificações para executar o código, mas a automação deste processo através do precompilador já está sendo implementado.

4.1.7 Exemplo de código modificado pelo precompilador

Vemos abaixo o código de uma função em C:

```
int functionA () {

  int mainInt = 0 ;

  // Do computations
  (...)

  function0 (  ) ;

  // Do computations
  (...)

  function1 (  ) ;
```

```
  // Do computations
  (...)
}
```

Agora o mesmo codigo após ser modificado pelo precompilador:

```
int functionA () {

  int currentGotoLabel = -1;
  ckp_push_data(&currentGotoLabel, sizeof(int), 1);
  if (ckp_restoring==1)
    ckp_get_data(&currentGotoLabel, sizeof(int), 1);
  int mainInt = 0 ;
  ckp_push_data(&mainInt, sizeof(int), 1);
  if (ckp_restoring==1) {
    ckp_get_data(&mainInt, sizeof(int), 1);
    if(currentGotoLabel == 0)
      goto ckp0;
    if(currentGotoLabel == 1)
      goto ckp1;
  }

  // Do computations
  (...)

 ckp0:
  currentGotoLabel = 0;
  function0 (  ) ;

  // Do computations
  (...)


 ckp1:
  currentGotoLabel = 1;
  function1 (  ) ;

  // Do computations
  (...)

  ckp_npop_data(2);
}
```

Neste trecho de código vemos o código para inserção e remoção das de variáveis locais na pilha e como as chamadas de funções são identificadas. Apesar do tamanho do código ter aumentado bastante,

na prática isto não é problema, dado que apenas uma parte das funções será modificada e que o exemplo mostrou apenas a parte do código da função que é modificado pelo precompilador. Na prática, o tempo de execução de uma aplicação científica é tipicamente dominado por pequenos trechos de código que não são modificados. Neste exemplo estes trechos de código estariam nas partes indicadas por (...).

4.2 Biblioteca de *Checkpointing*

Foi implementada uma biblioteca que fornece diversas funcionalidades para facilitar o *checkpointing* no nível da aplicação. Esta biblioteca fornece suporte para que aplicações possam salvar seu estado em um arquivo e posteriormente recuperar este estado, além de um timer para que se possa especificar um tempo mínimo entre diferentes *checkpoints*. Ela possui uma API em C que permite que esta seja utilizida em qualquer linguagem de programação que forneça suporte para chamadas em C.

O timer é implementado por uma classe C++ `CkpTimer` que fornece métodos para definir o intervalo de tempo desejado, para iniciar a contagem de um novo intervalo e para verificar se o último intervalo já terminou. Para realizar a contagem do tempo, a classe `CkpTimer` cria uma nova *thread* assim que a contagem de um novo intervalo é iniciada. Esta *thread* entra em modo *sleep* por um pelo intervalo de tempo detertminado. Quando esta thread é reativada, ela então atualiza uma variável sinalizando que o intervalo de tempo terminou.

Esta metodologia apenas garante que, quando o método que verifica se o último intervalo de tempo expirou retorna verdadeiro, o intervalo de tempo realmente foi completado. Mas pode acontecer deste método retornar falso e mesmo assim o intervalo de tempo definido já ter expirado. Isto ocorre porque a *thread* pode demorar um tempo muito maior que o definido no intervalo de tempo para ser reativada. Mas como o objetivo é apenas garantir um intervalo mínimo entre *checkpoints* e não um tempo máximo, esta abordagem satisfaz os requerimentos.

Para salvar o estado da aplicação são fornecidas chamadas que permitem que a aplicação salve dados de blocos de memória a um arquivo. A aplicação fornece uma lista com os blocos de memória a serem salvas, junto com o tamanho de cada bloco e a biblioteca coloca estes dados de maneira contígua num arquivo especificado.

Para recuperar os dados salvos, a biblioteca fornece uma função que retorna um endereço de memória contendo todos os dados presentes no arquivo. É a aplicação que tem a obrigação de extrair os dados das variáveis locais deste bloco de memória. Esta opção foi feita para diminuir a quantidade de dados que precisam ser salvos no arquivo de *checkpoint*. Se fosse feita a opção de retornar os dados de uma maneira mais estruturada seria necessário gravar informações de controle nestes arquivos, o que aumentaria seu tamanho.

Neste momento a biblioteca fornece apenas a opção de salvar os dados no sistema de arquivos local. O problema desta abordagem é que se a máquina onde a aplicação estava executando fica inacessível por algum motivo os dados do *checkpoint* passam a ficar inacessíveis. A exceção é quando se utiliza um sistema de arquivos distribuído, como o NFS. Neste caso, pode-se acessar os arquivos de qualquer nó conectado a este sistema de arquivos. Numa versão futura, será oferecida a opção de se salvar os dados em locais remotos através do uso de CORBA.

4.3 Coordenador de *Checkpointing*

Este módulo realiza a coordenação entre os processos de uma aplicação BSP de modo que *checkpoints* globais consistentes sejam gerados. Nesta implementação, os processos são numerados com valores entre `0` e `n-1`, onde `n` é o número de processos. O processo `0` é eleito o coordenador e possui a função de

coordenadar a geração dos *checkpoints* dos demais processos. Para garantir um estado global consistente, os *checkpoints* são sempre gerados imediatamente após o passo de sincronização do modelo BSP. Isto garante que não existem dependências entre os processos. Para determinar se um novo *checkpoint* deve ser gerado após uma passo de sincronização, o coordenador utiliza o *timer* fornecido fornecido pela biblioteca de *checkpointing*.

O sistema de detecção de falhas nos processos de foi implementado utilizando *heartbeats*. Nesta abordagem, um processo envia mensagens (*heartbeats*) periódicas que sinalizam que o processo que envia as mensagens está operacional. Este intervalo entre as mensagens pode ser ajustado.

Cada processo monitora o processo com valor imediatamente inferior ao seu, com exceção do processo 0 que monitora o processo n-1. Quando um processo p permanece por determinado período sem enviar nenhum *heartbeat*, o processo m que o monitora interpreta como uma falha em p.

Após detectar esta falha, o processo m se torna o coordenador da reinicialização da aplicação. Ele primeiro sinaliza a falha aos demais processos para que estes finalizem sua execução. Para garantir que todos os processos concordem com relação ao coordenador da reinicialização, o protocolo *two phase commit* é utilizado. Isto é importante para o caso em que dois processos tentam simultaneamente assumir o papel de coordenador. Neste caso o processo com o menor número é selecionado. Para realizar a reinicializar a aplicação, o coordenador de reinicialização resubmete a aplicação para execução na Grade.

5 Código da Implementação

O código da implementação está no endereço `http://www.ime.usp.br/~rcamargo/raphael_src.tar.gz`. Dentro deste arquivo existem dois diretórios. O diretório ckpCompiler contém o código do precompilador. O diretório ckpLib contém as bibliotecas de *checkpointing*, incluindo o coordenador.

Para compilar o precompilador, basta entrar no diretorio 'ckpCompiler' e digitar 'sh configure' e em seguida 'make all'. Existe um arquivo no diretório 'ckpLib' que pode ser utilizado para testar o precompilador. Para tal, basta entrar no diretório 'ckpCompiler/src' e digitar './occ -E -ckp ../../ckpLib/'

Dentro do diretório 'ckpCompiler/src', os arquivos metafunction.h e metafunction.cc foram escritos neste trabalho. Além disso, diversos trechos de código foram adicionados aos arquivos classwalk.cc, walker.cc e driver.cc. Estes trechos de código modificado podem ser facilmente identificados, pois estão entre linhas comentadas com "//————— RYC —————".

No caso das bibliotecas de *checkpointing*, estas podem ser compiladas entrando no diretório 'ckpLib' e digitando 'make'. Para rodar um código BSP no entanto é necessário fazer a instalação do InteGrade, que possui o código e instruções de instalação disponíveis no site `http://gsd.ime.usp.br/integrade`.

6 Conclusões e Perspectivas Futuras

Neste trabalho foi implemetnada uma versão inicial de um sistema de recuperação baseado em retrocesso para aplicações BSP rodando no InteGrade. O precompilador que modifica o código-fonte da aplicação pode ser utilizado tanto para aplicações isoladas como para aplicações BSP.

No caso do InteGrade, o objetivo é que este sistema permita a migração de processos entre máquinas. Dado que o InteGrade rodará em estações de trabalho compartilhadas, onde a chance de ocorrência de falhas é muito maior que no caso de máquinas dedicadas, este sistema de recuperação por retrocesso será bastante importante para o bom funcionamento da grade. Isto é especialmente importante no caso de

aplicações paralelas onde a falha de apenas um de seus processos normalmente requer a reinicialização de todos os demais.

Este trabalho consistiu em duas partes. A primeira parte consistiu em entender o funcionamento interno da ferramenta OpenC++. Dada que a documentação disponível sobre este funcionamento é praticamente inexistente, esta parte do trabalho foi basatnte demorada.

A segunda parte foi modificar o OpenC++ para funcionar como um precompilador que adicoina código de *checkpointing* e escrever as bibliotecas de suporte. A parte mais trabalhosa aqui consistiu em tratar das diversas pecularieda des das linguagens de programação C e C++, por exemplo o fato de variáveis poderem ser declaradas em posições arbitrárias num código em C++.

O próximo passo será melhorar o suporte do precompilador às linguagens C/C++, em especial o suporte a ponteiros compostos. Mas outros detalhes precisam ainda ser tratados como tratar a herança em C++ e salvar automaticamente estruturas e classes contendo ponteiros.

Além disso, futuramente será implementado um mecanismo de armazenamento distribuído de *checkpoints* que seja tolerante a falhas. Este mecanismo deverá ser implementado utilizando CORBA. Está nos planos ainda estudar a possibilidade de tornar este compilador portável, provavelmente utilizando a linguagem de representação de dados de CORBA.

Referências

[BMPS03] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. In *Proceedings of the 9th ACM SIGPLAN PPoPP*, pages 84–89, San Diego, USA, 2003.

[Chi95] Shigeru Chiba. Compiler-assisted heterogeneous checkpointing. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 285–299, October 1995.

[EAWJ02] Mootaz Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, May 2002.

[GKG+04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 2004. Accepted for publication on March/April 2004.

[HMS+98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[KBH01] Feras Karablieh, Rida A. Bazzi, and Margaret Hicks. Compiler-assisted heterogeneous checkpointing. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 56–65, New Orleans, USA, 2001.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[PaGKL95] J. S. Plank, M. Beck amd G. Kingsley, and K. Li. Libckpt: Transparent chieckpointing under unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, pages 213–323, 1995.

[SR96] Volker Strumpen and Balkrishna Ramkumar. Portable checkpointing and recovery in heterogeneous environments. Technical Report UI-ECE TR-96.6.1, University of Iowa, June 1996.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.