Escalonamento Tolerante a Falhas para Clusters Multicores

Brevik Ferreira da Silva, Wellison Moura dos Santos, Idalmis Milián Sardiña, Livia de Mesquita Teixeira, Felipe de Albuquerque

[1]Escola de Ciência e Tecnologia – Universidade Federal do Rio Grande do Norte (UFRN)
Av. Sen. Salgado Filho, 3000 - Lagoa Nova Natal - RN, 59078-970 – Natal – RN – Brazil

{idalmismilian}@ect.ufrn.br

Abstract. *Large-scale parallel applications running with increased performance and fault-free is a challenge of the high performance computing. However, for best results is important the efficient use of available resources, exploring for example the shared and distributed memories of the new multicores architectures. This paper proposes a hybrid approach for fault-tolerant scheduling on clusters based on multicores processors. A case study is proposed for a parallel application modeled by a Directed Acyclic Graph (GAD) using the hybrid programming OpenMP and MPI. The proposal should discuss the advantages that this model can bring to these architectures, compared with the previous approach.*

Resumo. *Um dos principais desafios da computação de alto desempenho é executar aplicações paralelas de grande porte com maior desempenho e livre de falhas. Entretanto, para obter melhores resultados é primordial o aproveitamento eficiente dos recursos disponíveis, explorando por exemplo o uso das diferentes memórias compartilhadas e distribuídas em novas arquiteturas multicores. Este trabalho propõe uma abordagem híbrida para o escalonamento tolerante a falhas sobre clusters baseados em processadores multicores. Um estudo de caso é proposto para uma aplicação paralela modelada por um Grafo Acíclico Direcionado (GAD) usando programação paralela híbrida OpenMP e MPI. O estudo proposto deve analisar as vantagens que este modelo pode trazer para estas arquiteturas, comparado com a abordagem anterior.*

1. Introdução

Clusters multicore tem se tornado uma plataforma popular em computação paralela ou de alto desempenho. Na lista publicada no Top500 de supercomputadores, a maioria das máquinas representam arquiteturas de *clusters* com processadores multicores. Intuitivamente, os processadores multicores podem dividir a carga de trabalho distribuindo as tarefas nos diferentes cores ou núcleos de processamento. Mesmo assim, comparados com *clusters* SMP ou NUMA, aplicações sobre *clusters* multi-cores não mostram sempre ótimos desempenhos nem escalabilidade. Atualmente, é fundamental um estudo maior das características destas arquiteturas e seus efeitos sobre o comportamento das aplicações que executam neles. Duas estratégias importantes empregadas nos atuais processadores multicores estão nos processadores da Intel e da AMD. Os da Intel provêm uma *cache* L2 compartilhada que igual a AMD emprega enlaces *HyperTransport* para rápidas transferências de dados. Assim, estas arquiteturas apresentam eficientes protocolos e ambas executam compartilhamentos rápidos de dados através dos cores. Entretanto,

diversos estudos são realizados para explorar as hierarquias de memórias, assim como analisar os efeitos e benefícios da execução das aplicações neles. Para muitas aplicações científicas, o custo da troca de mensagens sobre as redes e *clusters* de memória distribuída já é bastante conhecido, mas a análise de custo e efeitos explorando os múltiplos núcleos ou cores exige novas pesquisas.

Desenvolvedores de aplicações científicas e técnicas em geral, necessitam paralelizar altos volumes de código garantindo eficiência e portabilidade. O uso de arquiteturas multiprocessadas com memória compartilhada tem criado uma demanda urgente, precisando de uma forma diferente de programar as aplicações. A ferramenta OpenMP [Ope 2008], por exemplo, foi desenvolvida para direcionar estes aspectos e criar um padrão para a programação dos multiprocessadores. OpenMP consiste de um conjunto de diretivas de compilação e bibliotecas estendidas em Fortran, C, C++ para expressar o paralelismo da memória compartilhada. A programação paralela em OpenMP é bastante atual representando um padrão hoje. Por outro lado, a era da programação paralela marca a popularidade dos *softwares* MPI [MPI 2009] e Open MP como programação híbrida e o emprego de *clusters* multicores como as plataformas de *hardware* em inúmeras organizações e lares. Desta forma, surge uma necessidade eminente de estudantes e profissionais nestas áreas aprendam novas metodologias que combinem as funções tradicionais do padrão MPI com novas diretivas de OpenMP com o objetivo de obter melhores resultados.

Muito dos códigos híbridos usando MPI e OpenMP para executar as aplicações, são baseados em um modelo de estrutura hierárquica, que torna possível a exploração de ambas linguagens. O objetivo é tirar vantagens das melhores características de ambos os paradigmas de programação. A utilização da dupla MPI e OpenMP está emergindo como um padrão de fato [Rabenseifner et al. 2009b, Rabenseifner et al. 2009a, Su et al. 2004, Aversa et al. 2005, Chorley et al. 2009], ambos são dois padrões bem estabelecidos e possuem sólida documentação. No entanto, em cada caso ou aplicação deve ser analisado o modelo híbrido a ser utilizado, e sempre levando em consideração a arquitetura do *hardware*.

Por outro lado, para atingir alto desempenho ao executar aplicações paralelas de grande porte em MPI, diferentes trabalhos sobre escalonamento de tarefas, integram heurísticas de escalonamento com mecanismos tolerantes a falhas. Existem algoritmos de escalonamento que para tratar falhas empregam estratégias de replicação de tarefas baseadas no esquema primária-*backup* [Benoit et al. 2008, Qin and Jiang 2006, Al-Omari et al. 2005, Sardina et al. 2011a, Sardina et al. 2011b]. Neste caso, para a replicação ativa [Benoit et al. 2008, Anne Benoit and Robert 2008] ambas cópias da tarefa são executadas simultaneamente, o que pode sobrecarregar bastante o sistema distribuído. Já com a técnica de replicação passiva Qin2006, MilianBoeresDrummond2011], a *backup* de uma tarefa somente é ativada quando detectada falha na primária. A replicação passiva tem se mostrado na literatura uma alternativa interessante à replicação ativa, dado que não necessita de uso de recursos extras, com custos de execução de *backups* mais reduzidos.

Em [Sardina et al. 2011b] é proposto um algoritmo de escalonamento estático, inicialmente baseado em [Qin and Jiang 2006] ao empregar uma heurística do tipo *list scheduling* para escalonar as tarefas primárias e *backups*, com tolerância de uma falha *crash*. Esta abordagem adiciona o escalonamento bi-objetivo de [Sardina et al. 2011a] e acrescenta maior flexibilidade com a introdução de novos conceitos e critérios para o

escalonamento de *backups*.

Neste trabalho é estendida a pesquisa feita em [Sardina et al. 2011b] com o objetivo de explorar as novas arquiteturas multicores e propor novos estudos que apliquem as estratégias propostas de escalonamento e tolerância a falhas a estas arquiteturas. O artigo apresenta o estudo de caso de uma aplicação paralela baseada no algoritmo de escalonamento tolerante a falhas e que usa uma abordagem híbrida para a implementação paralela. Um dos desafios principais deste projeto é executar aplicações paralelas de problemas de grande porte com maior desempenho usando estratégias escalonamento de tarefas e tolerância a falhas que aproveitem o processamento multicore.

Inicialmente, a abordagem híbrida proposta para implementar o escalonamento tolerante a falhas é testada sobre um pequeno ambiente que permite executar programas paralelos. Depois os testes são realizados em outras arquiteturas maiores e mais complexas.

O artigo está organizado como segue. A seção 2 descreve o modelo da aplicação e da arquitetura empregados. A seção 3 mostra o algoritmo de escalonamento tolerante a falhas. A seção 4 explica a abordagem proposta neste trabalho para o escalonamento. Inicialmente, esta seção introduz conceitos de programação paralela híbrida e depois detalha um caso de estudo para um GAD de aplicação da Eliminação de Gauss. As últimas seções apresentam o ambiente de testes utilizado e discussões relacionadas.

2. Modelo do sistema

O trabalho proposto se aplica a um modelo de sistema que descreve características da aplicação e da arquitetura nas Seções 2.1 e 2.2, respectivamente. A modelagem parte da necessidade de representar o ambiente de trabalho a ser utilizado e deve suportar a execução de grandes e variadas aplicações paralelas.

2.1. Modelo da aplicação

Neste trabalho as aplicações são modeladas por *Grafos Acíclicos Direcionados* (GAD), com $G = (V, E, e, c)$, onde V é o conjunto de vértices (tarefas), E a relação de precedência, $e(v_i)$ com $v_i \in V$, o peso de execução associado à tarefa v_i e, $c(v_j, v_i)$ com $(v_j, v_i) \in E$, o peso de comunicação associado ao arco (v_j, v_i). Se $(v_j, v_i) \in E$ então, a execução de v_i não pode ser iniciada enquanto não seja completada a execução de v_j e os dados de v_j para v_i sejam recebidos por este. O conjunto de predecessores imediatos de v_i é denotado por $Pred(v_i)$, enquanto $Succ(v_j)$ são os sucessores imediatos de v_j. O GAD utilizado neste trabalho é uma paralelização do método matemático Eliminação de Gauss utilizado para solucionar sistemas de equações lineares. A estrutura deste grafo (Figura 1) possui vértices(tarefas) com pesos de computação variável, uma vez que os pesos das tarefas são maiores inicialmente na parte superior do GAD e diminuem a cada nível. Este GAD serve como modelo para estudar aplicações heterogêneas em relação às tarefas.

2.2. Modelo da arquitetura

Em relação a arquitetura, $P = \{p_0, p_1, ..., p_{m-1}\}$ é o conjunto de m processadores multicores, sendo que a cada p_j é associado o índice de retardo (*computational slowdown index*), denotado por $csi(p_j)$, sendo esta métrica inversamente proporcional ao poder computacional de p_j. O tempo de execução da tarefa v no processador p_j é dado

por $eh(v, p_j) = e(v) \times csi(p_j)$. Para duas tarefas adjacentes v_i e v_j alocadas em processadores distintos p_l e p_k, respectivamente, supõe que o custo associado à comunicação de $c(v_i, v_j)$ dados é definido como $ch(v_i, v_j) = c(v_i, v_j) \times L(p_l, p_k)$, onde a latência $L(p_l, p_k)$ é o tempo de transmissão por *byte* sobre o *link* (p_l, p_k). Cada processador p_j pode ter mais de um núcleo ou *core* n_i, e pode ser representado como $p_j = n_0, n_1, ..., n_{k-1}$.

São consideradas falhas permanentes de processador, eventos independentes entre si, e que ocorrem de acordo com uma distribuição de *Poisson* com probabilidade de falha $FP(p_j)\ \forall\ p_j\ \in\ P$ e valor constante, conforme [Qin and Jiang 2006]. $FP(p_j)$ representa a quantidade de falhas por unidade de tempo que podem ocorrer em p_j. O custo de confiabilidade $RC(v, p_j)$ de execução de v em p_j é definido como $RC(v, p_j)\ =\ FP(p_j) \times eh(v, p_j)$ e deve ser minimizado para aumentar a confiabilidade. Sendo $ltask(p_j)$ a lista de tarefas atribuídas a p_j, o custo de confiabilidade associado à p_j é $RC_p(p_j) = \sum_{\forall v \in ltask(p_j)} RC(v, p_j)$. Para um sistema P com m processadores, o custo de confiabilidade de escalonar uma aplicação pode ser definido como $RC(G, P)\ =\ \sum_{\forall p_j \in P} RC_p(p_j)$. Assim, a confiabilidade da aplicação G é dada por $R = e^{-RC(G,P)}$.

3. Algoritmo de Escalonamento Estático Tolerante a Falhas

O algoritmo de escalonamento estático proposto em [Sardina et al. 2011b] utiliza uma política do tipo *list scheduling* para escalonar as tarefas, incluindo mecanismos que permitem tolerar falhas permanentes de processador. Para isto, emprega uma técnica de replicacão passiva chamada primária-*backup* [Al-Omari et al. 2005, Qin and Jiang 2006]. Um dos objetivos deste algoritmo é minimizar o tempo total de execucção *(makespan*) de tal maneira que mais tarefas possam ser executadas. Outro objetivo é obter um sistema mais confiável reduzindo o custo de confiabilidade durante o escalonamento e permitir que falhas permanentes de processador possam ser toleradas. Portanto, a estratégia de escalonamento estático proposta em [Sardina et al. 2011b] apresenta os objetivos: encontrar uma alocação para as primárias das tarefas da aplicação sobre uma arquitetura proposta; de acordo com a alocação das primárias, encontrar uma alocação para as *backups* que permita tolerar falhas permanentes de processador; e minimizar o *makespan* e maximizar a confiabilidade da aplicação mesmo na presença de falhas. O algoritmo consiste das seguintes 3 etapas:

1. Ordenar tarefas usando um critério de prioridade: OrdenaTarefas();
   Na primeira etapa as tarefas são ordenadas por seus *b-levels* (*static bottom level*), conforme [Topcuouglu et al. 2002], denotado por $prior(v),\ \ \forall v \in V$. A lista de tarefas V_{ord} em ordem decrescente de $blevel(v)$, definido como:

$$blevel(v) = \begin{cases} \overline{et(v)} & \text{se } succ(v) = \emptyset \\ max_{u \in succ(v)} \left\{ \overline{et(v)} + \overline{ct(v,u)} + blevel(u) \right\} & \text{em outro caso} \end{cases}$$

   onde $\overline{et(v)}$ é o tempo de execução médio de v considerando todos os processadores e $\overline{ct(v,u)}$ é o tempo de comunicação médio, considerando todos os *links*. Em heurísticas *list scheduling* sobre ambientes heterogêneos, a escolha da tarefa usando *b-level* tem mostrado melhor desempenho para um número maior de GADs [Topcuouglu et al. 2002].

2. Escalonar cópias primárias: EscalonaPrimarias();
   Na segunda etapa, usando um *list scheduling* para escalonar as tarefas, a lista ordenada por $prior(v_i)$ é percorrida e, para cada tarefa da lista é efetuada uma procura pelo melhor processador de acordo com os seguintes critérios. Seja $v \in V$ a próxima tarefa a ser escalonada, o melhor processador $p_j \in P$ a ser escolhido para execução de v é aquele que minimiza a função de custo ponderada $f(v, p_j) = \alpha_1 EFT(v, p_j) + \alpha_2 RC(v, p_j)$. Ou seja, é escolhido o processador que satisfaz $F(v, p) = min_{\forall p_j \in P'} \{f(v, p_j)\}$. De acordo com a função $f(v, p_j)$, tarefas mais críticas em relação ao seu tempo de execução e comunicação tendem a ser escalonadas em processadores mais confiáveis e mais rápidos, aumentando o desempenho e a probabilidade da aplicação não falhar [Sardina et al. 2011b]. Dependendo do caso, uma tarefa v com granularidade mais fina pode ser alocada diretamente num core n_i daquele processador p_j permitindo que outras tarefas, por exemplo tarefas paralelas, sejam alocadas no mesmo processador mas em cores distintos. Novos critérios de escalonamento como este são utilizados para aproveitar o processamento multicore e reduzir comunicação, dependendo do caso.
   Para calcular o tempo de início $EST(v, p_j)$ da tarefa v no processador p_j é considerada uma política de inserção de tarefas em espaços ociosos de processador conforme [Topcuouglu et al. 2002]. O peso α_i, associado a cada critério, é uma variável $0 \leq \alpha_i \leq 1$ que representa o nível de importância de cada critério ($i = 1, 2$).
3. Escalonar cópias *backups*: EscalonaBackups(). Para obter o escalonamento das *backups* é utilizado também um *list scheduling* com a mesma função ponderada e critérios de escalonamento do passo anterior (2), junto a uma estratégia primária-*backup* como proposto em [Sardina et al. 2011b]. O escalonamento das tarefas *backups* (ver Figura 1) é realizado logo depois das primárias e verifica-se que seja satisfeito o conjunto de critérios propostos em [Sardina et al. 2011b].

As informações geradas pelo escalonamento final deste algoritmo são utilizadas posteriormente por uma ferramenta gerenciadora para executar a aplicação paralela e recuperar a aplicação em caso de falhas.

4. Abordagem Híbrida para o Escalonamento da Aplicação

A aplicação escalonada usando o algoritmo proposto, deve ser executada em um ambiente apropriado de computação paralela e com linguagens de programação específicas como C++ ou Fortran, que podem incluir as diretivas e funções necessárias para a programação paralela, por exemplo o padrão MPI. O artigo [Sardina et al. 2011a] mostra a implementação de uma ferramenta MPI ou *middleware* [Boeres and Rebello 2004, de P. Nascimento et al. 2005, Sardina 2010] (SGA) que foi modificado para gerenciar o escalonamento tolerante a falha das tarefas da seção anterior e executar a aplicação paralela em caso de falhas. Estas implementações em MPI não consideram as características multicore dos processadores. Este trabalho propõe uma nova abordagem híbrida para a implementação do escalonamento tolerante a falhas. Inicialmente, é feita nesta seção uma introdução do estilo de programação que pode ser utilizada e as razões que levam a esta escolha. Logo depois é apresentado um caso de estudo para uma aplicação específica utilizando a estratégia híbrida proposta.

4.1. Programação Híbrida: OpenMP e MPI

A biblioteca MPI (Message-Passing Interface) [MPI 2009] é uma especificação ou interface para troca de mensagens que representa um paradigma de programação paralela. Os dados são movidos de um espaço de endereçamento de um processo para o espaço de endereçamento de outro processo, através de operações específicas para intercâmbio de mensagens. As principais vantagens do estabelecimento de um padrão deste tipo são a portabilidade e a facilidade de utilização. Em um ambiente de comunicação de memória distribuída é vantajoso ter a possibilidade de utilizar rotinas em mais alto nível ou abstrair a necessidade de conhecimento e controle das rotinas de passagem de mensagens, como por exemplo *sockets*. Como a comunicação por troca de mensagens é padronizada por um fórum de especialistas, MPI tende oferecer eficiência, escalabilidade e portabilidade.

MPI além de ter como alvo de sua utilização as plataformas de *hardware* de memórias distribuídas, o mesmo pode ser utilizado também em plataformas de memória compartilhada como as arquiteturas SMPs e NUMA. E ainda, torna-se mais aplicável em plataformas híbridas, como cluster de máquinas NUMA. Todo o paralelismo em MPI é explícito, ou seja, de responsabilidade do programador, e implementado com a utilização dos construtores da biblioteca no código.

OpenMP [Ope 2008] é uma API (Application Program Interface) para programação em C/C++, Fortran entre outras linguagens, que oferece suporte para programação paralela em computadores que possuem uma arquitetura de memória compartilhada. O modelo de programação adotado pelo OpenMP é bastante portável e escalável, podendo ser utilizado numa gama de plataformas que variam desde um computador pessoal até supercomputadores. OpenMP é baseado em diretivas de compilação, rotinas de bibliotecas e variáveis de ambiente. O OpenMP provê um padrão suportado por quase todas as plataformas ou arquiteturas de memória compartilhada. Um detalhe interessante e importante é que isso é conseguido utilizando-se um conjunto simples de diretivas de programação. Em alguns casos, o paralelismo é implementado usando 3 ou 4 diretivas. É, portanto, correto afirmar que o OpenMP possibilita a paralelização de um programa sequencial de forma amigável. Como o OpenMP é baseado no paradigma de programação de memória compartilhada, o paralelismo consiste então de múltiplas *threads*.

Entretanto, OpenMP precisa de suporte para alocação distribuída de estruturas compartilhadas, podendo causar gargalos em alguns sistemas. Uma vez que o conjunto de dados é muito grande, há muitos *misses* na *cache*, afetando severamente o desempenho e a escalabilidade; otimizações nos códigos OpenMP não são simples, ou seja, demandam alto conhecimento do programador. Para minimizar esses problemas, trabalhos como [Su et al. 2004], por exemplo, utilizam recursos de alocação disponíveis na máquina da SGI nos testes. A implementação do OpenMP da SGI (Silicon Graphics, Inc) aceita parametrizações que afetam o modo como o sistema aloca os dados e como os dados são transferidos em tempo de execução para minimizar a latência de acesso aos dados na memória.

A programação híbrida com ambos paradigmas OpenMP e MPI é o resultado de mesclar a paralelização explícita de grandes tarefas em MPI, com a paralelização de tarefas mais simples em OpenMP [Rabenseifner et al. 2009b]. Em um alto nível, o programa fica hierarquicamente estruturado como uma série de tarefas MPI, cujo código sequencial está enriquecido com diretivas OpenMP para adicionar *multithreading* e aproveitar as

características da presença de memória compartilhada e multiprocessadores dos nós. O uso de OpenMP deve acrescentar *multithreading* aos tradicionais processos MPI, o que de certa maneira, deixa o desenvolvimento dos programas mais complexos.

Quando não há transmissão de mensagens dentro do nó, as otimizações do MPI não são necessárias. As partes do OpenMP devem ser otimizadas para a arquitetura do nó pelo programador, por exemplo, empregando cuidados com a localidade dos dados na memória em nós NUMA, ou usando mecanismos de afinidade. Portanto, a maneira mais simples e segura de combinar o MPI com o OpenMP é utilizar as directivas MPI apenas fora das regiões paralelas do OpenMP. Quando isso acontece, não há qualquer problema nas chamadas da biblioteca do MPI.

4.2. Abordagem proposta

Em trabalhos anteriores [Sardina et al. 2011a, Sardina et al. 2011b], uma ferramenta MPI gerenciadora recebe como entrada os arquivos com a informação gerada pelo algoritmo de escalonamento estático tolerante a falhas. Para cada processador, um arquivo mostra a maneira em que foram escalonadas as tarefas primárias e *backups*. O escalonamento das *backups*, descreve como será o comportamento da aplicação em caso de falha, ou seja, diferentes execuções possíveis da aplicação, dependendo do processador que falha.

Um mecanismo para deteção e recuperação de falhas forma parte da ferramenta híbrida, e redefine as principais funções MPI que a aplicação, modelada por um GAD, pode usar (MPI_Send e MPI_Recv). As modificações destas funções permite tratar as falhas em conjunto com processos gerentes que monitoram e processam a ocorrência de falhas no sistema. Para a aplicação tornar-se tolerante a falhas deve ser compilada com esta ferramenta e incluir os arquivos necessários com as informações geradas pelo escalonamento proposto para recuperar a aplicação. Portanto, para executar a aplicação, nenhum código é alterado e como resultado da compilação, a tolerância a falhas é automaticamente inserida no programa. Consequentemente, a aplicação compilada fica pronta para executar em paralelo e com a capacidade de recuperar-se em caso de falhas.

Inicialmente, foi implementada a detecção com o uso de *error handlers*, onde funções redefinidas de MPI (envio e recebimento) interceptam os erros que ocorrem durante a execução da aplicação. Assim, uma identificação de falhas pelo MPI é habilitado através de *error handlers* associados aos comunicadores. A cada comunicador é associado o *MPI ERRORS RETURN*, o qual permite que as funções retornem um código de erro na ocorrência de falhas. Outros mecanismos de tolerância a falhas são utilizados seguindo os trabalhos [Sardina 2010, da Silva 2010] para recuperar a aplicação. A seção 4.2.1 descreve um exemplo de como procede o mecanismo híbrido para executar e recuperar a aplicação paralela em caso de falha.

4.2.1. Caso de uso: Aplicação de Gauss

A Figura 1 mostra um exemplo do GAD da aplicação paralela Eliminação de Gauss para 9 tarefas. Esta aplicação está formada por dois tipos de tarefas em função da posição no grafo, denotadas por $T_{k,k}$ e $T_{k,j}$ respectivamente. De forma geral, podemos descrever o código híbrido (MPI e OpenMP) proposto para esta aplicação como mostrado

a seguir, onde $T_{k,k}$ representa as tarefas $V_0(T_{1,1})$, $V_4(T_{2,2})$ e $V_7(T_{3,3})$ na Figura 1 e $T_{k,j}$ são as outras tarefas restantes do GAD.

1. **para** $k = 1 \ldots m-1$ **faça (MPI)**
2. $T_{k,k}$: {
3. **recebe** $(col_k, T_{k-1,k})$
4. **para** $i = k+1 \ldots m$ **faça (OpenMP)**
5. $a_{ik} = a_{ik}/a_{kk}$
6. **envia** $(col_k, T_{k,j})$ }
7. **para** $j = k+1 \ldots m$ **faça (MPI)**
8. $T_{k,j}$: {
9. **recebe** $(col_k, T_{k,k})$
10. **para** $i = k+1 \ldots m$ **faça (OpenMP)**
11. $a_{ij} = a_{ij} - a_{ik} * a_{kj}$
12. **envia** $(col_j, T_{k+1,j})$ }

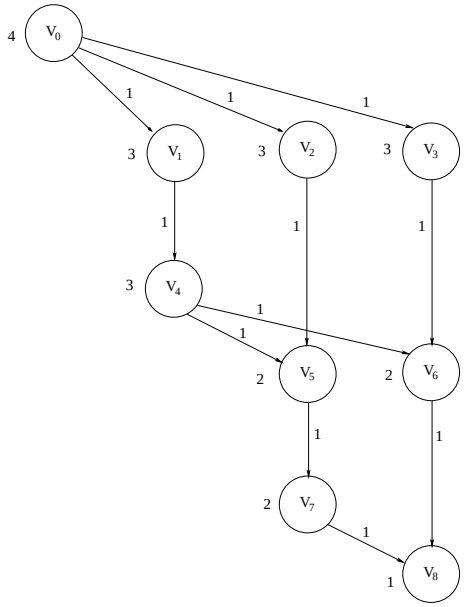

Figura 1. Eliminação de Gauss com 9 tarefas

Uma abordagem para explorar o *hardware* multicore disponível na arquitetura, mesmo na presença de falhas, é atribuir as tarefas T_{kk} e T_{kj} do GAD de Gauss (Figura 1) aos mesmos processadores p_j escolhidos pelo escalonamento estático tolerante a falhas da Seção 3. Dentro do *loop* de cada tarefa (T_{kk} ou T_{kj}) o código é paralelizado somente com OpenMP decompondo a tarefa em subtarefas menores. Para isto, é associada uma *thread* a cada core n_i de cada processador p. Na parte mais externa do código das tarefas do GAD T_{kk} e T_{kj}, a decomposição segue a distribuição do GAD, utilizando MPI para realizar a troca de mensagens entre as tarefas na forma que mostra a Figura 1. Observe que as tarefas T_{kj} recebem as mensagens das tarefas T_{kk}, ou seja, $V_1(T_{12})$, $V_2(T_{13})$ e $V_3(T_{14})$ na Figura 1, recebem da tarefa $V_0(T_{11})$ a coluna 1 da matriz calculada, usando as diretivas de comunicação de MPI. As tarefas T_{kk}, exceto a primeira, recebe mensagem da tarefa T_{k-1k} ($j = k$), assim neste caso $V_4(T_{22})$ recebe a coluna 2 de $V_1(T_{12})$.

Para o modelo da arquitetura proposto (Seção 2.2), o escalonamento pode também alocar uma tarefa considerando um core n_i no lugar de um processador p_j, em casos como este a comunicação MPI entre algumas tarefas pode ser desnecessária, a mensagem é lida pela mesma memória compartilhada reduzindo custos de comunicação. Neste caso de Gauss por exemplo, várias tarefas T_{kj} (primárias ou *backups*) de um mesmo nível k na Figura 1, podem ser alocadas em cores diferentes n_i de um mesmo processador p_j. Assim uma única mensagem MPI, comum a todas as tarefas T_{kj}, é recebida pelo processador p_j no lugar de j mensagens. Para isto, as tarefas T_{kj} são associadas a *thread*s com OpenMP. Estudos comparativos analisando a diferença de escalonar mais tarefas, por exemplo T_{kj} (não dependentes ou paralelas), dentro de um mesmo processador em cores diferentes, em vez destas mesmas tarefas em processadores separados são realizados. Isto permite comparar custos de comunicação usando memória distribuída contra custos de execução na memória compartilhada em função do tipo de escalonamento escolhido para cada tarefa. Novos critérios de escalonamento podem surgir desta pesquisa, assim como uma nova função ponderada que considere estes custos como objetivos ou parâmetros do escalonamento.

Desta forma, a execução paralela segue o escalonamento proposto na Seção 3 e em caso de falha de processador p_j, as *backups* das tarefas pertencentes a p_j (e que ainda não executaram), serão alocadas em novos processadores ou cores escolhidos pelo escalonamento tolerante a falhas. Neste caso, aplica-se dentro do código de cada tarefa *backup* a mesma decomposição OpenMP da sua tarefa primária como explicado no código híbrido. Para o caso de tarefas *backups* T_{kj} do mesmo nível k pode ser aplicada a agregação OpenMP em um mesmo processador p analisada antes. Espera-se que o tempo de execução de cada tarefa *backup* (T_{kk} ou T_{kj}) seja menor que o da mesma *backup* executando com a abordagem MPI anterior [Sardina 2010] ou custos de comunicação sejam reduzidos dependendo do caso.

5. Ambiente de Testes

Inicialmente a proposta foi testada em uma única máquina Intel Core i7 e sistema operacional Linux, com o objetivo de observar o funcionamento da implementação com a nova abordagem híbrida. Depois deve ser estendida a outras arquiteturas maiores. O objetivo é pesquisar em distintas arquiteturas com características multicores e de maior escala, para assim analisar hierarquias de memórias com seus efeitos ou benefícios para a execução das aplicações na presença de falhas. Por exemplo, realizar estudos em relação às hierarquias de memória, vertical e horizontal, a avaliação do desempenho da comunicação MPI pela troca de mensagens nos *clusters*, ao gerenciamento de caches paralelas hierárquicas, compartilhamento de caches de diferentes níveis e suas limitações entre outros aspectos. Desta forma, propor novos algoritmos multicores que maximizem a localidade dos dados e explorem a estrutura hierárquica das caches.

Os estudos comparativos no ambiente tem como objetivo analisar quanto muda o desempenho da aplicação com a nova abordagem híbrida, sem falha e/ou com falha. Para testar a tolerância a falhas inicialmente é simulada a ocorrência de uma falha permanente de processador mediante a execução de um arquivo *script* que contêm a chamada ao comando *killall*, para matar os processos que formam parte do processador com falha. Este arquivo pode ser disparado enquanto a aplicação ainda está sendo executada. Os testes estão sendo realizados com o ambiente de execução em modo exclusivo e principalmente

para 2 cenários de execução da aplicação: com a ferramenta sem falha e com a ferramenta com falha para as duas abordagens comparadas. Medições de tempo de execução e do número de mensagens são registradas, variando o número de tarefas.

Para avaliar a proposta os próximos testes serão realizados em três ambientes maiores diferentes e com outras aplicações alvo modeladas como descrito na seção 2.1. O primeiro ambiente é uma rede ethernet Linux de computadores no laboratório da ECT, o segundo uma rede WiFi de *notebooks* e o último o *Cluster* SGI Altix do Instituto Internacional de Física da UFRN.

6. Discussões e Trabalhos Futuros

A programação híbrida com o MPI e o OpenMP, adequa-se às arquiteturas atuais baseadas em *clusters* multiprocessores. Permite diminuir o número de comunicações entre os diferentes nós e aumentar o desempenho de cada nó sem que ocasione um incremento considerável dos requisitos de memória. Aplicações que possuem dois níveis de paralelismo podem utilizar processos MPI para explorar paralelismo de granularidade grossa/média, trocando mensagens ocasionalmente para sincronizar informação e/ou distribuir trabalho, e utilizar *threads* para explorar paralelismo de granularidade média/fina por partilha do espaço de endereçamento. Aplicações que tenham restrições ou requisitos que possam limitar o número de processos MPI que podem ser usados podem tirar partido do OpenMP para explorar o poder computacional dos restantes processadores disponíveis. Aplicações cujo balanceamento de carga seja difícil de conseguir apenas por utilização de processos MPI, podem tirar partido do OpenMP para equilibrar esse balanceamento, atribuindo um diferente número de *threads* a diferentes processos MPI em função da respectiva carga.

Em resultados preliminares obtidos para a aplicação de Gauss, observa-se que o tempo de execução e o número de mensagens diminuim com o incremento do número de tarefas. A execucão da aplicação com a nova abordagem obteve melhor desempenho quando comparada com resultados com a abordagem MPI anterior não híbrida. A aplicação com a ferramenta modificada, com falha e sem falha obteve maior desempenho quando comparado com o cenário da abordagem anterior. Portanto, nesta análise inicial, métricas como tempo de execução e número de mensagens apresentam uma diminuição nos experimentos realizados variando a quantidade de tarefas.

Nesta etapa inicial estuda-se quanto a estratégia é viável sobre uma arquitetura multicore menor e como diversos fatores podem influenciar no desempenho da aplicação, tais como: o número de processos criados ao mesmo tempo, o número de tarefas escalonadas por processador e o número máximo de mensagens que a estrutura de memória usada pelo processador pode armazenar. A ferramenta híbrida implementada nesta primeira etapa tem um escopo menor. Uma segunda etapa será estendê-la a *clusters* e redes maiores, adaptada ao *middleware* de [Boeres and Rebello 2004, de P. Nascimento et al. 2005] e com variações do algoritmo de escalonamento de tolerância a falhas propostos. Em relação ao modelo de falhas, levar-se em consideração a ocorrência de múltiplas falhas.

Referências

(May, 2008). OpenMP application program interface version 3.0 complete specifications. http://www.openmp.org/mp-documents/specs30.pdf.

(Nov, 2009). MPI: A message-passing interface standard version 2.1. `www.mpi-forum.org/docs/mpi21-report.pdf`.

Al-Omari, R., Somani, A. K., and Manimaran, G. (2005). An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems. *J. Parallel Distrib. Comput.*, 65(5):595–608.

Anne Benoit, M. H. and Robert, Y. (April 14-18, 2008). Fault tolerant scheduling of precedence task graphs on heterogeneous platforms. In *Proceedings of the 22th ACM/IEEE International Parallel Distributed Processing Symposium IPDPS'08 - APDCM'08 IEEE Computer Society Press*, Miami, Florida, USA.

Aversa, R., Di Martino, B., Rak, M., Venticinque, S., and Villano, U. (2005). Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Comput.*

Benoit, A., Hakem, M., and Robert, Y. (2008). Realistic models and efficient algorithms for fault tolerant scheduling on heterogeneous platforms. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, pages 8–12, Portland, Oregon, USA.

Boeres, C. and Rebello, V. E. F. (2004). Easygrid: towards a framework for the automatic grid enabling of legacy MPI applications: Research articles. *Concurrency And Computation : Practice And Experience*, 16(5):425–432.

Chorley, M. J., Walker, D. W., and Guest, M. F. (2009). Hybrid message-passing and shared-memory programming in a molecular dynamics application on multicore clusters. In *Int. J. High Perform. Comput.*, pages 196–211.

da Silva, J. A. (2010). *Tolerância a Falhas para Aplicações Autônomas em Grades Computacionais*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil.

de P. Nascimento, A., da C. Sena, A., da Silva, J. A., de C. Vianna, D. Q., Boeres, C., and Rebello, V. E. F. (2005). Managing the execution of large scale MPI applications on computational grids. *17th. International Symposium on Computer Architecture and High Performance Computing*.

Qin, X. and Jiang, H. (2006). A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems. *Parallel Computing*, 32(5):331–356.

Rabenseifner, R., Hager, G., and Jost, G. (2009a). Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core smp nodes. *Proceedings of the Cray Users Group Conference 2009 (CUG 2009)*.

Rabenseifner, R., Hager, G., and Jost, G. (2009b). Hybrid MPI/OpenMP parallel programming on clusters of multi-core smp nodes.

Sardina, I. M. (2010). *Escalonamento Estático de Tarefas Bi-objetivo e Tolerante a Falhas para Sistemas Distribuídos*. PhD thesis, Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brasil.

Sardina, I. M., Boeres, C., and Drummond, L. M. A. (2011a). An efficient weighted bi-objective scheduling algorithm for heterogeneous systems. *Parallel Computing*, 37:349–364.

Sardina, I. M., Boeres, C., and Drummond, L. M. A. (2011b). Escalonamento estático bi-objetivo e tolerante a falhas para sistemas distribuídos. In *Simposio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Campo Grande.

Su, M. F., El-Kady, I., Bader, D. A., and Lin, S. (2004). A novel fdtd application featuring openmp-mpi hybrid parallelization. In *Proceedings of the 2004 international Conference on Parallel Processing ICPP*, pages 373–379, Washington, DC, USA. IEEE Computer Society.

Topcuouglu, H., Hariri, S., and Wu, M. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions Parallel Distributed Systems*, 13(3):260–274.