Uma Biblioteca de Processos Leves para a Implementação de Aplicações Altamente Paralelas*

Gerson Geraldo H. Cavalheiro Lucas Correia Villa Real† Evandro Clivatti Dall'Agnol‡
Programa Interdisciplinar de Pós-Graduação em Computação Aplicada
Universidade do Vale do Rio dos Sinos
São Leopoldo – RS – Brasil
{gersonc, lucasvr, ecd}@exatas.unisinos.br

Resumo

Um dos maiores problemas ligados à programação concorrente (ou paralela) não está relacionado somente à questão da identificação da concorrência do problema, mas também à exploração eficiente do paralelismo do hardware disponível. Neste sentido, diversos ambientes de programação/execução buscam realizar o mapeamento da concorrência do programa em execução ao paralelismo real da arquitetura sobre a qual a execução está se desenvolvendo. Em geral, estes ambientes apoiam-se em técnicas de escalonamento e modelos de programação. Neste trabalho é apresentada uma interface de programação, baseada no padrão threads *POSIX, voltada à descrição da concorrência em aplicações e o núcleo executivo associado.*

1 Introdução

Nos últimos anos, o desenvolvimento do processamento de alto desempenho (PAD) encontrou um grande aliado nos aglomerados de computadores (*clusters*) e nas arquiteturas multiprocessadoras com memória compartilhada (Symmetric Multi-Processors, ou SMPs). No entanto, a exploração dessas arquiteturas com o intuito de obtenção de bom desempenho de execução não é trivial, tendo sido desenvolvidos diversos ambientes de execução, dotados ou não de uma interface de programação especializada, para auxiliar o programador nesta tarefa. Este trabalho descreve Anahy, em especial sua interface de programação e seu mecanismo de escalonamento de tarefas. O objetivo de Anahy é permitir que o programador possa descrever a concorrência de sua aplicação de forma precisa e independente dos recursos computacionais disponíveis na arquitetura sobre a qual a execução do programa poderá vir a ser executado.

De custo relativamente baixo, os aglomerados e os SMPs têm aumentado sua participação como suporte ao desenvolvimento de programas para aplicações com alto custo computacional. Dentre as razões que motivam este fato, além do custo, está o potencial de desempenho que pode vir a ser obtido. Dados estes facilmente comprováveis através dos preços aplicados pelo mercado aos microcomputadores bi- e quadri-processadores e pela incidência de aglomerados na lista das 500 máquinas mais potentes em operação.[1] No entanto, a programação dessas máquinas envolve, além da codificação do problema propriamente dito, o mapeamento da concorrência da aplicação, ou seja, as atividades concorrentes no programa, nas unidades de suporte ao cálculo (processador e memória) da arquitetura. A esse mapeamento estão ligadas questões referentes à repartição da carga computacional entre os diferentes processadores e ao compartilhamento de dados entre os nodos.

Desta forma, o uso efetivo de aglomerados e de arquiteturas SMP para o PAD requer a realização do mapeamento da concorrência da aplicação sobre os recursos computacionais disponíveis. No entanto, cabe observar que, na maioria dos casos, este mapeamento não pode ser realizado de forma direta, pois a concorrência da aplicação normalmente é superior ao paralelismo suportado pela arquitetura. Portanto, utilizando recursos convencionais de programação concorrente, paralela ou distribuída, tais como multiprogramação leve (*threads*) [Cohen et al., 1998], MPI [Snir et al., 1996] ou RPC, o programador deve, além de programar sua aplicação de forma concorrente, determinar o número de tarefas concorrentes adequado para uma determinada arquitetura e distribuir essas tarefas e os respectivos dados entre os processadores e módulos de memória da arquitetura.

*Projeto Anahy – CNPq (55.2196/02-9)
†ITI/CNPq
‡ITI/CNPq

[1] A lista se encontra em www.top500.org e apresenta, ano a ano, um contínuo crescimento no número de aglomerados.

Transpor essas dificuldades, oferecendo tanto uma interface de programação de alto nível como mecanismos de gerência de recursos de hardware, implica em abordar questões ligadas à portabilidade de código e de desempenho dos programas [Alverson et al., 1998]. Cilk [Blumofe et al., 1995], Athapascan-1 [Galilée et al., 1998] e PM^2 [Denneulin et al., 1998] são ferramentas para o PAD inseridas nesse contexto. Estas ferramentas provêem tanto recursos de programação, para descrição da concorrência de uma aplicação, como introduzem núcleos executivos capazes de tirar proveito dos recursos da arquitetura visando desempenho na execução de programas.

Na próxima seção estes três ambientes são brevemente apresentados. O restante do artigo encontra-se assim organizado: a seção 3 apresenta o modelo de arquitetura de suporte à execução de programas Anahy; na seção 4 são discutidas características ligadas ao controle da correção de execução de programas concorrentes; na seção 5 são apresentadas as primitivas de descrição de concorrência da interface Anahy e na seção 6, o algoritmo empregado para escalonamento do programa concorrente. A seção 7 compara características de Anahy com as dos demais ambientes apresentados e, por fim, uma conclusão é apresentada.

2 Ambientes para Alto Desempenho

Esta seção descreve, brevemente, três ferramentas para exploração do processamento de alto desempenho: PM^2, Cilk e Athapascan-1. Embora não contemplando todas as classes de recursos de programação/execução disponíveis, é possível ilustrar abordagens adotadas para (*i*) representar o modelo de suporte de execução; (*ii*) descrever a concorrência de uma aplicação; e para (*iii*) incorporar mecanismos de escalonamento.

GTLB [Denneulin, 1998] é um núcleo de escalonamento de processos leves implementado sob a forma de biblioteca de funções. Essa biblioteca é utilizada em conjunto com PM^2, uma interface aplicativa para desenvolvimento de aplicações concorrentes [Denneulin et al., 1998]. A arquitetura considerada por GTLB é de uma máquina multiprocessada dotada de uma memória compartilhada (escrita e leitura livre pelos fluxos de execução criados). GTLB prevê um modelo de aplicação do tipo *branch-and-bound* (ou seja, tarefas independentes). Um protótipo foi implementação do protótipo sobre aglomerado de computadores.

O escalonador de GTLB explora a independência das tarefas: cada tarefa é disparada no momento em que é criada, podendo ser interrompidas e posteriormente reiniciadas. Não há garantia de coerência no acesso aos dados (ordenação) na memória compartilhada. É empregado um mecanismo de migração de tarefas para equilibrar a carga de trabalho entre os diferentes nodos da arquitetura.[2]

Cilk [Blumofe et al., 1995] possui protótipos implementados para arquiteturas SMP e aglomerados, sendo uma extensão à linguagem C. É previsto um modelo de programação sobre arquiteturas com memória compartilhada, na qual fluxos são executados de forma concorrente. A criação de atividades concorrentes é realizada de forma explícita e a comunicação por meio de leitura e escrita em memória compartilhada. O mecanismo de sincronização também é explícito: um fluxo é capaz de aguardar o término de todos fluxos de execução por ele criados. Assim sendo, criando e sincronizando atividades explicitamente, o programador é capaz de controlar a evolução do processo de troca de dados entre estas durante a execução do programa.

O núcleo executivo de Cilk implementa um algoritmo de escalonamento de lista [Graham, 1969], do tipo roubo-de-trabalho, no qual um processador ocioso "rouba trabalho"na lista de tarefas de um processador ativo. A estratégia determina que cada processador execute suas atividades priorizando sua profundidade no programa, profundidade que pode ser representada pela ordem de execução das chamadas de funções caso a execução fosse seqüencial. De forma prática, a criação de uma atividade não implica na geração de uma nova atividade de fato, e sim, no disparo imediato da execução da nova atividade, ficando a seqüência da atividade criadora em estado de espera do término da atividade criada para prosseguir sua execução; um mecanismo de continuação, auxiliado por um processo de compilação, garante a correta execução do programa em uma arquitetura multiprocessada. O objetivo do escalonador de Cilk é minimizar o tempo de execução de programas, no entanto, foram implementadas técnicas para para redução do consumo de memória na execução de programas [Blelloch et al., 1997].[3]

Athapascan-1 [Galilée et al., 1998] oferece a visão de uma arquitetura onde diversos processadores acessam uma memória compartilhada. As atividades concorrentes da aplicação são explicitamente codificadas através de tarefas. Uma tarefa é definida como uma seqüência de código, possuindo uma entrada de dados (parâmetros) e produzindo, ao seu término, um resultado (saída); a leitura dos dados de entrada e a escrita dos dados de saída são realizadas na memória compartilhada. A correção da execução é garantida disparando uma tarefa apenas quando os parâmetros necessários para sua execução estejam disponíveis na memória compartilhada e, uma vez iniciada, não é interrompida, voltando a acessar a memória compartilhada apenas para escrita dos resultados. O controle de tal ordem de precedência é realizado através de um grafo de fluxo de dados entre tarefas. Nesse ambiente, a função do escalonador [Cavalheiro et al., 1998] é de explorar o paralelismo da arquitetura, respeitando a ordem de precedência das tarefas explícitas nas trocas de dados.

A implementação de Athapascan-1 considerou

[2] Maiores informações em `www.pm2.org`.

[3] Maiores informações:`supertech.lcs.mit.edu/cilk`.

a possibilidade de utilizar, no núcleo executivo, diferentes algoritmos de escalonamento (dotados ou não de estratégias de balanceamento de carga) [Cavalheiro et al., 1998, Cavalheiro, 2001]. Esse ambiente opera em arquiteturas SMP e em aglomerados, sendo utilizado sob a forma de uma biblioteca em programas C++ (um pré-compilador auxilia no uso desta biblioteca).[4]

3 Arquitetura Alvo

A implementação de Anahy disponibiliza um ambiente para a exploração do processamento de alto desempenho sobre arquiteturas do tipo aglomerado de computadores, onde cada nó pode vir a ser um multiprocessador com memória compartilhada. No entanto, essa arquitetura é considerada apenas para a implementação do ambiente. O programador tem a visão de uma arquitetura *virtual* multiprocessada dotada de memória compartilhada. Essas duas visões da arquitetura são ilustradas na figura 1.

Como destaca a figura 1, a arquitetura real é composta por um conjunto de nodos de processamento, dotados de memória local e de unidades de processamento (CPUs). A arquitetura virtual é composta por um conjunto de processadores virtuais (PVs) alocados sobre os nodos e por uma memória compartilhada pelos PVs. O número de PVs e o tamanho da memória compartilhada são limitados em função da capacidade dos recursos da arquitetura real. No entanto, a capacidade de processamento e de armazenamento *virtuais* não alteram o modelo.

Cada um dos PVs possui a capacidade de executar seqüencialmente as atividades que a ele forem submetidas: enquanto um PV estiver executando uma atividade, nenhuma outra sinalização será tratada por ele. Quando ocioso, ou seja, não executando nenhuma atividade do usuário, o PV pode ser despertado ao existir uma nova atividade apta a ser executada. Além das instruções convencionais (aritméticas, lógicas, de controle de fluxo, etc.), foram introduzidas duas novas instruções para descrição da concorrência da aplicação, permitindo a criação de uma nova atividade e a sincronização de uma atividade com o final de outra, e instruções de alocação, deleção, leitura e escrita na memória compartilhada. Nenhum sinal é previsto para ser enviado entre PVs, estejam estes no mesmo nó ou não. Cada PV conta ainda com um espaço de memória próprio, utilizado para armazenar dados locais às atividades do usuário que serão executadas.

A comunicação entre os PVs se dá através da memória compartilhada, acessada pelas instruções introduzidas pela arquitetura virtual. Essa arquitetura não suporta nenhum mecanismo de sincronização: todo controle ao acesso aos dados compartilhados deve ser feito através das instruções de controle de concorrência.

[4]Maiores informações: `www-id.imag.fr`.

4 Princípio de Controle de Execução

A exemplo de Cilk e de Athapascan-1, entre outras interfaces de programação concorrente, Anahy oferece um mecanismo que garante a correção da execução do programa. O princípio considerado diz respeito ao controle de dependências de dados entre as diversas atividades concorrentes que serão criadas pelo programa em execução.

4.1 Comunicação e sincronização em programas seqüenciais

Um programa imperativo seqüencial pode ser visto como uma coleção de instruções elementares, executadas em uma ordem específica dado um conjunto de dados de entrada. Um programa correto produz o mesmo conjunto de dados de saída toda vez que for executado tendo o mesmo conjunto de dados de entrada. Isto porque a mesma seqüência de execução de instruções é reproduzida.

Neste contexto, o programador é consciente que seu programa consiste em uma série de transformações de dados em memória. Assim o esforço de codificação materializa as transformações que devem ser sofridas pelos dados em uma seqüência de instruções elementares. Em outras palavras, o resultado da instrução deve ser estocados em algum lugar da memória. Esse valor em memória servirá como dado de entrada para que uma instrução futura, na execução do programa, possa ela também ser executada. Desta forma, em programas imperativos seqüenciais, é resolvido o problema da comunicação de dados entre instruções.

No entanto, a comunicação não garante a correta execução do programa. As instruções devem ser sincronizadas de forma que uma instrução não seja executada antes que os dados que ela necessitar como entrada estejam disponíveis na memória. No paradigma de execução imperativo seqüencial, a sincronização entre instruções é garantido pelo próprio mecanismo de execução, no qual as instruções são executadas seqüencialmente, não sendo iniciada a execução de uma instrução S_i caso a instrução S_{i-1} não tenha sido completada. Essa dependência é representada por $S_{i-1} \prec S_i$.

Existe portanto, na execução de um programa seqüencial imperativo P, uma ordem específica para ativação das instruções dado um conjunto X de dados de entrada:

$$\mathcal{E}(\mathcal{P}, \mathcal{X}) = S_1 \prec S_2 \prec S_3 \prec \ldots \prec S_s$$

observe que a sincronização é realizada considerando a posição da instrução no fluxo de execução do programa. Uma execução errônea do programa poderia ser causada pela execução de uma instrução fora da ordem prevista ou em duplicata, a não execução de uma instrução ou a execução de uma instrução que não esteja nesta seqüência.

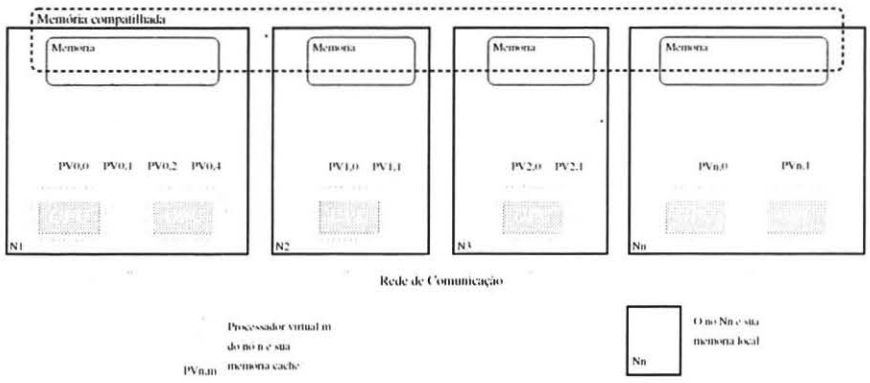

Figura 1. Modelo lógico e físico da arquitetura de suporte à Anahy.

4.2 Comunicação e sincronização entre tarefas

Assim como nos programas seqüenciais, programas concorrentes produzem resultados através de transformações de dados recebidos em entrada. No entanto, a programação concorrente (paralela ou distribuída) implica na divisão do trabalho total da aplicação em atividades concorrentes, denominadas *tarefas*. Inevitavelmente estas tarefas necessitam trocar dados entre si de forma a fazer com que o programa evolua.

Desse modo, as interfaces para programação concorrente introduzem mecanismos de comunicação de dados e de sincronização as tarefas ([Ghezzi and Jazayeri, 1998]). Os mecanismos de comunicação permitem que dados produzidos por alguma tarefa sejam, de alguma forma, colocados a disposição de uma outra tarefa. Os mecanismos de sincronização permitem a uma tarefa informar a outra que um dado encontra-se disponível ou verificar a disponibilidade de um determinado dado. Com os mecanismos de sincronização é possível controlar o avanço da execução do programa, não permitindo que tarefas sejam executadas antes que seus dados de entrada estejam disponíveis.

É importante observar que, no contexto deste trabalho, a função da sincronização é de conciliar as datas de execução das tarefas em relação à produção/consumo de dados. Muitas ferramentas de programação, no entanto, oferecem mecanismos de sincronização que não garantem uma ordem na execução das atividades, garantindo apenas que uma atividade tenha conhecimento do estado de uma outra; um exemplo clássico é o uso de mutex para controle de execução de sessões críticas. O uso deste tipo de recurso de sincronização, muito embora fundamental para diversas aplicações, introduz um nível de indeterminismo na execução que não permite que seja garantido um determinado resultado para todas execuções de um programa dado um determinado conjunto de dados de entrada. Como este tipo de sincronização não permite controle da comunicação de resultados de tarefas, ela não esta sendo considerada.

Desta forma, a execução $\mathcal{E}(\mathcal{P}_c, \mathcal{X})$ de um programa concorrente pode ser representada por uma coleção de tarefas $\mathcal{T} = (\tau_1, \tau_2, \ldots \tau_n)$ e um conjunto de dados $\mathcal{X} = (x_1, x_2, \ldots x_n)$, descritos por um grafo de dependências (como em [Galilée et al., 1998]) $\mathcal{G} = (V, E)$, onde os nodos $V = \mathcal{T} \cup \mathcal{X}$ são representados pelo conjunto tarefas e de dados manipulados pelo programa e os acessos (leitura/escrita) são representados pelas arestas: $E = (\mathcal{T} \times \mathcal{X}) \cup (\mathcal{X} \times \mathcal{T})$. Nesta representação, (τ_i, x_i) indica que o dado x_i é produzido pela tarefa τ_i e (x_i, τ_i) que o dado x_i é necessário para executar a tarefa τ_i.

Estas considerações permitem visualizar as partes componentes do modelo de execução de um programa concorrente em Anahy e definir de forma mais precisa uma tarefa. Um programa em execução consiste em um conjunto de tarefas, onde cada tarefa delimita uma seqüência de instruções elementares e define dois conjuntos de dados: (i) os dados necessários para iniciar sua execução e (ii) os dados produzidos como resultado de sua execução. A ordem de execução das tarefas é definida pela disponibilidade de seus dados de entrada. Ao terminar, uma tarefa produz um resultado que poderá viabilizar a execução de uma outra tarefa.

5 Interface de Programação Anahy

Um dos maiores problemas ligados ao desenvolvimento de programas concorrentes advém do alto grau de liberdade de ação que o programador passa a ter: decomposição da sua aplicação em atividades concorrentes e alocação destas atividades sobre as unidades de cálculo da arquitetura, entre outros. Isto sem contar o número de arquiteturas com características distintas e as inúmeras ferramentas de programação. O modelo proposto por Anahy foi projetado segundo diversos critérios considerados *úteis* em um modelo de programação concorrente [Skillicorn, 1994]. Entre estes critérios encontra-se a capacidade de minimizar as dificuldades de gerenciamento de um grande número de fluxos de execução concorrente e de comunicações entre eles.

5.1 Serviços oferecidos

Os serviços da interface de programação de Anahy oferecem ao programador mecanismos para explorar o paralelismo de uma arquitetura multiprocessada dotada de uma área de memória compartilhada, permitindo sincronizar as tarefas concorrentes da aplicação, realizando, implicitamente, troca de dados entre elas. Esses serviços podem ser representados através das operações *fork/join*, disponibilizando ao programador uma interface de programação bastante próxima ao modelo oferecido pela multiprogramação baseada em processos leves (no que diz respeito a criação e sincronização com o término de fluxos de execução). Essa abstração permite a descrição de atividades sem que o programador identifique explicitamente quais destas atividades são concorrentes na sua aplicação.

Uma operação *fork* consiste na criação lógica de um novo fluxo de execução, sendo o código a ser executado definido por uma função $\mathcal{F}$ definida no corpo do programa. Esse operador retorna um identificador ao novo fluxo criado. No momento da invocação da operação *fork*, a função a ser executada deve ser identificada e passados os parâmetros necessários a sua execução. O programador não possui nenhuma hipótese sobre o momento em que este fluxo será disparado. Sabe-se que após seu término, um resultado será produzido, ou seja, $\mathcal{Y} = \mathcal{F}(\mathcal{X})$.

A sincronização com o término da execução de um fluxo é realizada através da operação *join*, identificando o fluxo a ser sincronizado. Essa operação permite que um fluxo bloqueie, aguardando o término de outro fluxo, de forma a recuperar os resultados produzidos. Ou seja, recuperar $\mathcal{Y}$ produzido por $\mathcal{F}(\mathcal{X})$.

Desta forma, as operações de sincronização (*fork* e *join*) realizadas no interior de um fluxo de execução permitem definir novas tarefas que poderão vir a ser executadas de forma concorrente. Essas tarefas são definidas implicitamente:

- no momento do *fork*: o novo fluxo de execução inicia executando uma nova tarefa que possui como dados de entrada os argumentos da própria função;
- no momento de um *join*: o fluxo de execução termina a execução de uma tarefa e cria uma nova a partir da instrução que sucede (na ordem lexicográfica) o operador *join*. Essa nova tarefa tem como dados de entrada a memória local do fluxo de execução (atualizada até o momento que precedeu a realização do *join*) e os resultados retornados pela função executada pelo fluxo sincronizado;e,
- no fim da função executada por um fluxo de execução.

Observe que o acesso à memória compartilhada é realizado implicitamente pelos operadores *fork* e *join*.

Outro aspecto à observar é a capacidade de execução seqüencial do programa quando eliminados os operadores de sincronização. Em outras palavras: a execução concorrente da aplicação produz o mesmo resultado que ofereceria a execução seqüencial do mesmo programa, o que facilita o desenvolvimento do programa e sua depuração.

5.2 Sintaxe utilizada

Anahy está sendo desenvolvido de forma a permitir compatibilidade com o padrão POSIX para *threads* (IEEE P 1003.c). Desta forma, as primitivas e estruturas oferecidas são um subconjunto dos serviços oferecidos por este padrão. Os atuais esforços de implementação estão concentrados em uma interface de serviços para programas C/C++.

Definição do corpo de um fluxo de execução O corpo de um fluxo de execução é definido como uma função C convencional, como representado no seguinte exemplo:

```
void* func( void * in ) {
      /* código da função */
      return out;
}
```

Neste exemplo, a função `func` pode ser instanciada em um fluxo de execução próprio. O argumento `in` corresponde ao endereço de memória (na memória compartilhada) onde se encontram os dados de entrada da função. A operação de retorno (`return out`) foi colocada apenas para explicitar que, ao término da execução de `func`, o endereço de um dado na memória compartilhada deve ser retornado pela função, endereço este referente ao armazenamento do resultado produzido pela tarefa.

Sincronização de fluxos de execução As sintaxes das operações *fork* e *join* correspondem as operações de criação e de espera por término de *thread* em POSIX: `pthread_create` e `pthread_join`. As sintaxes destes operadores são exemplificadas por:

```
int athread_create( athread_t *th,
                    athread_attr_t *atrib,
                    void *(*func)(void *),
                    void *in );
int athread_join(athread_t th, void **res);
```

Nesta sintaxe, `athread_create` cria um novo fluxo de execução para a função `func`; a entrada desta função está presente no endereço de memória `in`. O novo fluxo criado poderá ser referenciado posteriormente através do valor `th`, o qual consiste em um identificador único. Os valores fornecidos por `atrib` definem atributos com quais o programador informa características do novo fluxo de execução no que diz respeito a sua execução (por exemplo, necessidades

de memória). Na operação `athread_join` é identificado o fluxo com o qual se quer realizar a sincronização e `res` identifica um endereço de memória (compartilhada) para os dados de retorno do fluxo. Ambos operadores retornam um código de erro.

6 Concepção do Núcleo Executivo

Anahy prevê a execução de programas concorrentes tanto sobre aglomerados de computadores como sobre arquiteturas SMP. O ambiente provê transparência no acesso aos recursos de processamento da máquina. Como resultado, o uso de Anahy como ambiente de programação/execução permite que o programador codifique apenas sua aplicação, livrando-o de especificar o mapeamento das tarefas nos processadores (ou dos dados nos módulos de memória). O núcleo executivo foi igualmente concebido de forma a suportar a introdução de mecanismos de balanceamento de carga.

6.1 Algoritmo de escalonamento

O algoritmo de escalonamento pressupõe a arquitetura descrita na seção 3 e utiliza, como unidade de manipulação, uma tarefa. Uma tarefa é uma unidade de trabalho, definida pelo programa em execução, composta por uma seqüência de instruções capaz de ser executada em tempo finito – uma tarefa não possui nenhuma dependência externa (tal uma sincronização), nem pode entrar em nenhuma situação de errônea, tal um laço sem fim. Dentre as instruções executadas por uma tarefa, podem existir operações de criação de novas tarefas. Como visto na seção 5.1, uma tarefa termina ao executar uma operação de sincronização com uma tarefa com o término de outras tarefas.

O algoritmo gerencia quatro listas de tarefas: a primeira contém as tarefas **prontas** (aptas a serem lançadas), a segunda, as tarefas **terminadas** cujos resultados ainda não foram solicitados (a operação de *join* sobre estas tarefas ainda não foi realizada). A terceira e a quarta lista contêm tarefas **bloqueadas** e **desbloquadas**, respectivamente.

Para compreender o algoritmo de escalonamento, considere uma arquitetura monoprocessada. O processador, inicialmente vazio, busca a primeira tarefa, τ_1, da lista de tarefas prontas (no caso prático, a função *main*) e inicia sua execução. As instruções elementares de τ_1 são computadas normalmente, já as que descrevem o comportamento concorrente da aplicação envolvem o processo de escalonamento. Caso seja executada uma operação *fork*, uma nova tarefa τ_2 é criada e armazenada na lista de tarefas prontas e τ_1 segue executando. Caso seja executada uma operação de *join*, por exemplo com τ_2, τ_1 termina e uma nova tarefa τ_3 é criada, sendo o início de seu código determinado pela instrução que sucede o *join*; o estado inicial de τ_3 é bloquado, pois τ_3 somente poderá executar quando τ_2 for terminada, tendo produzido os dados necessários a τ_3: a essa relação de dependência dá-se a notação $\tau_2 \prec \tau_3$. A tarefa τ_2 é então retirada da lista de tarefas prontas e executada. Ao término de τ_2, a relação $\tau_2 \prec \tau_3$ é satisfeita, sendo τ_3 desbloquada e iniciada, tendo como entrada os dados produzidos por τ_2. Esse procedimento pode ser recursivo.

No caso de uma arquitetura paralela, dois ou mais processadores executam o mesmo algoritmo descrito no parágrafo anterior, implicando que duas ou mais tarefas executem simultaneamente. Assim, no momento em que uma tarefa τ_i solicitar um *join* com τ_j, duas outras situações podem ocorrer: ou τ_j já terminou ou τ_j está sendo executada no momento. Caso τ_j já tenha terminado, o procedimento consiste em recuperar os dados produzidos por τ_j permitindo que o processador prossiga com a execução de τ_{i+1} (τ_j é retirada da lista de tarefas terminadas). Caso τ_j esteja sendo executada, τ_{i+1} permanece bloqueada e o processador busca uma nova tarefa na lista de tarefas prontas. τ_{i+1} será desbloqueada quando τ_j for terminada.

Esse algoritmo permite obter o tempo de execução de uma tarefa $t(\tau_i)$ e sua data de término máxima $T(\tau_i)$:

$$t(\tau_i) = I_i + m * \sigma_c$$

$$T(\tau_i) \leq t(\tau_i) + T(\tau_{i-1}) + \sum_{j=0}^{k} t(\tau_j)$$

Ou seja, o tempo necessário para executar uma tarefa τ_i é o custo da execução de suas instruções elementares, mais o custo associado à criação de m tarefas concorrentes, sendo o custo da criação de uma tarefa σ_c. A data de término de uma tarefa τ_i deve considerar o tempo de execução das tarefas que a precedem. Esses tempos colocam em evidência que uma tarefa não pode ser iniciada antes que todas as tarefas que produzam dados necessários à sua computação não estejam concluídas. Portanto, em um programa em execução, existem relações entre suas tarefas, representadas por $\tau_j \prec \tau_i$ (leia-se o início de τ_i depende do término de τ_j) de tal forma que é possível identificar caminhos de dependência de fluxos de dados entre as tarefas. Dentre estes, a seguinte seqüência com k tarefas:

$$\mathcal{E}(\mathcal{P}_c, \mathcal{X}) = \tau_1 \prec \ldots \prec \tau_{k-2} \prec \tau_{k-1} \prec \tau_k$$

define o maior caminho de transformação de dados no programa $\mathcal{P}_c$ tendo como entrada $\mathcal{X}$. Essa seqüência é denominada caminho crítico e representa a carga computacional do problema que não pode ser paralelizado ([Graham, 1969], [Konig and Roch, 1997]). Ou seja, para um $\mathcal{P}_c(\mathcal{X})$,

$$T_\infty(\mathcal{P}_c, \mathcal{X}) = T(t_k)$$

determina seu tempo mínimo de execução.

A existência de um caminho crítico norteia a implementação do escalonador de Anahy: todo custo adicional à execução de tarefas deve ser evitado e, durante toda execução do programa, ao menos um dos processadores deve estar ativo executando uma tarefa deste caminho. Partindo da análise de $T_\infty(\mathcal{P}_c, \mathcal{X})$ pode ser observado que a ocorrência destas situações resultam em atrasos no lançamento de $T(t_k)$, e no consequente aumento no tempo no processamento do caminho crítico.

6.2 Escalonamento multinível

Da implementação do núcleo executivo, destaca-se sua organização do escalonamento em três níveis. O primeiro é realizado pelo sistema operacional e consiste no mapeamento dos fluxos de execução associados aos PVs aos recursos físicos de processamento (de forma equivalente, os dados manipulados em um nó na memória local).

O escalonamento aplicativo, no qual se dá a distribuição da carga computacional e o controle da execução do programa, é realizado nos níveis seguintes. O primeiro deles refere-se a alocação das tarefas aos PVs. Nesta alocação é considerada a ordem de execução das tarefas (controle semântico) e o escalonamento gerencia as listas de tarefas (prontas, terminadas, etc.) de forma global aos PVs.

Finalmente, o terceiro nível de escalonamento é encarregado da distribuição da carga computacional gerada entre os nodos que compõem a arquitetura utilizada. Nesta distribuição podem ser considerados diversos fatores, entre eles o custo computacional das tarefas e a localidade física dos dados – essa última pode ser obtida através de uma análise da dependência dos dados de entrada e saída das tarefas do usuário.

7 Sumário

A tabela 1 sumariza algumas das características dos ambientes descritos na seção 2 e as compara com as encontradas em Anahy. Como característica comum, destaca-se o fato de todos os ambientes possuirem um núcleo executívo. Também é ressaltado, que o modelo inicial de Cilk não foi previsto para aglomerado, uma extensão proveu suporte à esta arquitetura e que Athapascan-1 possui um pré-compilador, em opção ao uso direto da biblioteca.

Em relação ao escalonamento, todos os ambientes incorporam alguma forma de distribuição de tarefas. Embora as diferentes técnicas de escalonamento possam afetar o desempenho na execução de programas, destaca-se que, à exceção de GTLB, os mecanismos empregados provêem suporte ao controle semântico dos programas. Enquanto em Cilk este controle está associado à sincronização entre atividades, em Athapascan-1 e em Anahy a semântica é controlada pela troca de dados entre tarefas. Os mecanismos empregados, no entanto, diferem: enquanto Anahy dispara a execução de uma tarefa quando o dado que ela produz se faz necessário a uma outra, Athapascan-1 habilita a execução de uma tarefa assim que os dados necessários a sua execução encontram-se disponíveis.

As estratégias adotadas por Athapascan-1 e Anahy no controle semântico refletem a forma como cada ambiente manipula a memória compartilhada. Athapascan-1 exige o uso de tipos de dados especiais para compartilhamento de informações: acessos a estes dados são controlados por um mecanismos à parte. Em Anahy, os dados trocados entre tarefas são obtidos implicitamente, na criação e na sincronização de tarefas. Resulta destas duas abordagens que em Anahy a manipulação do grafo de dependências é realizada sob demanda de sincronizações, de forma semelhante a Cilk, no controle do avanço da execução.

8 Conclusão

O presente trabalho apresentou o estado atual do desenvolvimento de Anahy, um ambiente para exploração do processamento de alto desempenho em aglomerados de computadores e em arquiteturas SMP. O enfoque principal foi dado à interface de programação proposta e aos princípios adotados para modelagem do ambiente, considerando tanto de sua interface de programação quanto ao núcleo executivo. Trabalho anteriores ([Garzão et al., 2001], [Villa Real et al., 2002]) abordam a questão da portabilidade do ambiente e dos programas escritos em Anahy – são discutidos a opção desenvolver o ambiente com uso de ferramentas de sofware livre e o uso de mecanismos de escalonamento a nível aplicativo.

Anahy encontra-se disponível em uma versão para arquiteturas SMP. O modelo adotado para a interface de programação tem sido validado através da implementação de aplicações em programas utilizando o princípio de programação de Anahy com ferramentas tradicionais de programação, como MPI e *threads* POSIX. Uma destas aplicações (busca de padrões de imagens) encontra-se descrita em [Moschetta et al., 2002].

As próximas etapas têm por objetivo uma versão para aglomerados e a introdução de mecanismos de escalonamento dotados de técnicas de balanceamento de carga. Por fim, deverão ser introduzidos os mecanismos definidos por POSIX para controle de sincronizações entre processos leves (sessões críticas e variáveis de condição). Muito embora a utilização destes mecanismos não seja recomendada em Anahy, devido a potencial perda de desempenho, eles deverão ser incorporados para facilitar a compatibilidade com códigos já existentes.

Tabela 1. Características de ambientes de programação concorrente

	PM^2/GTLB	Cilk	Athapascan-1	Anahy
Linguagem base	C	C	C++	C
Padrão POSIX *threads*	Não	Não	Não	Sim
Recursos de programação	Biblioteca	Extensão de C	Biblioteca	Biblioteca
Compilador	Não	Sim	Pré-compilador	Não
Núcleo executivo	Sim	Sim	Sim	Sim
Suporte à aglomerados	Sim	Não	Sim	Sim
Escalonamento	Balanceamento de carga dinâmico	Compartilhamento de carga dinâmico	Diversos, estáticos e dinâmicos	Algoritmos de lista dinâmicos
Controle semântico	Não	Fluxo de execução	Fluxo de dados	Dependência de dados
Sincronização	Explícita	Explícita	Implícita	Explícita
Descrição da concorrência	Explícita	Explícita	Explícita	Explícita
Compartilhamento de dados	Mensagens	Memória global	Memória compartilhada	Memória compartilhada
Tipos de dados especiais	Não	Não	Sim	Não

Referências

[Alverson et al., 1998] Alverson, G. A., Griswold, W. G., Lin, C., Notkin, D., and Snyder, L. (1998). Abstractions for portable, scalable parallel programming. *IEEE Trans. on Parallel and Distributed Systems*, 9(1):71–86.

[Blelloch et al., 1997] Blelloch, G. E., Gibbons, P. B., Matias, Y., and Narlikar, G. J. (1997). Space-efficient scheduling of parallelism with synchronization variables. In *Proc. of the 9th Annual ACM Symp. on Parallel Algorithms and Architectures*, Newport.

[Blumofe et al., 1995] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. C. E. (1995). Cilk: an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, 30(8):207–216.

[Cavalheiro, 2001] Cavalheiro, G. G. H. (2001). A general scheduling framework for parallel execution environments. In *Proceedings of SLAB'01*, Brisbane, Australia.

[Cavalheiro et al., 1998] Cavalheiro, G. G. H., Denneulin, Y., and Roch, J.-L. (1998). A general modular specification for distributed schedulers. In Verlag, S., editor, *Proc. of Europar'98*, Southampton.

[Cohen et al., 1998] Cohen, W. E. et al (1998). Exploitation of multithreading to improve program performance. In *Proc. of The Yale Multithreaded Programming Workshop*, New Haven.

[Denneulin, 1998] Denneulin, Y. (1998). *Conception et ordonnancement des applications hautement irrégulières dans un contexte de parallélisme à grain fin*. PhD thesis, Université des Sciences et Technologies de Lille, Lille.

[Denneulin et al., 1998] Denneulin, Y., Namyst, R., and Méhaut, J. F. (1998). Architecture virtualization with mobile threads. In *Proc. of ParCo 97*, volume 12 of *Advances in Parallel Computing*, Amsterdam. Elsevier.

[Galilée et al., 1998] Galilée, F., Roch, J.-L., Cavalheiro, G. G. H., and Doreille, M. (1998). Athapascan-1: online building data flow graph in a parallel language. In *Pact'98*, Paris, France.

[Garzão et al., 2001] Garzão, A. S., Villa Real, L. C., e Cavalheiro, G. G. H. (2001). Ferramentas para desenvolvimento de um ambiente de programação sobre agregados. In *Anais do Workshop em Software Livre*, Porto Alegre.

[Ghezzi and Jazayeri, 1998] Ghezzi, C. and Jazayeri, M. (1998). *Programming Language Concepts*. John Wilei & Sons, New York, 3 edition.

[Graham, 1969] Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429.

[Konig and Roch, 1997] Konig, J.-C. et Roch, J.-L. (1997). Machines virtuelles et techniques d'ordonnancement. In Barth, D. et al editor, *ICaRE'97: Conception et mise en oeuvre d'applications parallèles irrégulières de grande taille*, Aussois. CNRS.

[Moschetta et al., 2002] Moschetta, E., Osório, F. S., e Cavalheiro, G. G. H. (2002). Reconhecimento de imagens em aplicações críticas. In *III Workshop em Sistemas de Alto Desempenho*, Vitória. SBC.

[Skillicorn, 1994] Skillicorn, D. (1994). *Foundations of Parallel Programming*. Cambridge, Great Britain.

[Snir et al., 1996] Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. (1996). *MPI: the complete reference*. MIT Press, Cambridge, MA, USA.

[Villa Real et al., 2002] Villa Real, L. C., Dall'Agnol, E. C., e Cavalheiro, G. G. H. (2002). Construção de um ambiente de programação para o processamento de alto desempenho. In *ERAD 2002: Sessão de Pôsteres*, São Leopoldo. SBC.