Make Distribuído

Aredis Sebastião de Oliveira

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Make Distribuído

Aredis Sebastião de Oliveira

Dezembro de 1997

Banca Examinadora:

- Prof. Dr. Rogério Drummond Burnier Pessôa de Mello Filho[1] (Orientador)
- Prof. Dr. Francisco Vilar Brasileiro[2]
- Prof. Dr. Ricardo de Oliveira Anido[1]
- Prof. Dr. Hans Kurt Edmund Liesenberg[1] (Suplente)

[1] Instituto de Computação, Unicamp.
[2] Departamento de Sistemas e Computação, UFPB.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Oliveira, Aredis Sebastião de

OL4m Make distribuido / Aredis Sebastião de Oliveira -- Campinas, [S.P. :s.n.], 1997.

Orientador : Rogerio Drummond Pessôa de Mello Filho

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatistica e Computação Cientifica.

1. Processamento distribuido. 2. Programação paralela (Computação). 3. Algoritmos. I. Mello Filho, Rogerio Drummond B.P. de (Rogerio D.B.P. de). II. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Computação Científica. III. Título.

Make Distribuído

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Aredis Sebastião de Oliveira e aprovada pela Banca Examinadora.

Campinas, 15 de Dezembro de 1997.

Prof. Dr. Rogério Drummond Burnier Pessôa de Mello Filho[1] (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 15 de dezembro de 1997 pela Banca Examinadora composta pelos Professores Doutores

Prof. Dr. Francisco Vilar Brasileiro

Prof. Dr. Ricardo de Oliveira Anido

Prof. Dr. Rogério Drummond Burnier Pessoa de Mello Filho

Resumo

Este trabalho apresenta o projeto e a implementação de um Make Distribuído (MakeD) baseado na conhecida ferramenta *make*. A aplicação cliente/servidor MakeD permite usar uma rede de computadores como um recurso computacional único para minimizar o tempo gasto no processo de *make* durante o desenvolvimento de projetos de médio e grande porte.

Num ambiente de desenvolvimento típico uma estação cliente apresenta um padrão de uso de CPU com curtos períodos de intensa utilização e longos períodos de inatividade ou baixa utilização. Dessa forma, utilizar os clientes como servidores de CPU e usar a ferramenta MakeD para distribuir as tarefas de compilações entre eles melhora o uso da capacidade de processamento disponível na rede, contribuindo para balancear a utilização dos recursos.

A implementação do MakeD combina o mecanismo de RPC com recursos de *multithreading* para explorar a distribuição de tarefas na rede e a multi-tarefa em cada sistema. Os resultados de testes comparativos entre MakeD, GNU *Make* e outro *make* distribuído (Dmake) demonstram a eficiência dos métodos empregados.

O trabalho também discute alguns aspectos de transformação de aplicações centralizadas em distribuídas e propõe um servidor de *make* distribuído multi-usuário como extensão para o MakeD.

Abstract

This work presents the design and implementation of a Distributed Make (MakeD) utility. The MakeD client/server application allows using a network as a single computing resource to reduce compilation time in the development of medium and large projects.

In a typical development environment, client workstations presents relatively short periods of high CPU load followed by long periods of low or zero utilization. MakeD uses ordinary clients as CPU servers in order to distribute tasks, thus making a network's idle processor capacity available and balancing resource use.

MakeD uses RPC and multithreading to achieve distributed processing in the network and multitasking within each node. The results of comparative tests show performance gains relative to GNU Make and a similar distributed make (Dmake).

This work also discusses some aspects of transforming centralized applications into distributed ones and proposes a multi-user distributed Make Server as a future evolution.

Agradecimentos

A Deus, por mais uma oportunidade e pela força para aprender e fazer.

Aos meus pais, Armando e Dina, que mesmo distantes, sempre estiveram presente no meu pensamento. Este trabalho é mais um fruto da sua paciência, trabalho e dedicação ao longo de todos estes anos.

Aos meus irmãos, Arinaldo, Arilda, Arília e Areno pelos incentivos e por pertencermos à mesma família.

A querida Jeanne, pelas correções neste trabalho e pela sua paciência, amor e carinho.

Ao Edmar e Emerson, que juntos decidimos partir para mais um desafio: o mestrado. Foram companheiros de alegrias, decisões, trabalho e convívio agradável.

Ao meu orientador, Prof. Dr. Rogério Drummond pelo problema, as idéias e as perguntas certas e ao Prof. Dr. Paulo Lício que colaborou no início de outro trabalho.

Aos colegas e amigos do Laboratório A-HAND, pela ajuda e companhia em dias e noites num harmonioso ambiente de trabalho. Em especial, agradeço ao Fernando pela ajuda na confecção das figuras e ao Carlos Furuti e Alexandre Teles pelas correções e inúmeras sugestões neste trabalho e pelos "helps" no entendimento das intermináveis linhas de código do GNU *Make*.

Aos amigos de mestrado, Maurício, Ana Waleska, Raul, Neiva, Eustáquio, Paulo Drummond, Lúcio, Geraldo Magela, Eliane, Giovanna, Gladys, Márcio, Dinalva (in memorian), Edilmar e outros pelos momentos de alegria nesta cidade.

Aos colegas e amigos de trabalho da área de conectividade e do Centro de Computação da Unicamp pelo apoio e incentivo.

Aos professores do Departamento de Informática da Universidade Federal de Uberlândia e do Instituto de Computação da Unicamp pela transmissão do conhecimento e experiências e pelo incentivo à pesquisa.

Ao CNPq pelo apoio financeiro nos dois primeiros anos do trabalho.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

INTRODUÇÃO

Este trabalho visa transformar uma aplicação centralizada existente numa ferramenta distribuída que usa e explora os recursos oferecidos pelos elementos de um ambiente de rede distribuído.

O *Make* Distribuído (MakeD), uma extensão da conhecida ferramenta *make*, busca oferecer a facilidade de uso dos recursos disponíveis na rede durante o desenvolvimento de projetos de médio e grande porte. O principal objetivo é aproveitar a ociosidade dos processadores de maneira transparente ao usuário para minizar o tempo gasto no processo de *make* de um sistema. Com a experiência de migração de uma aplicação centralizada para o ambiente distribuído, busca-se também levantar aspectos práticos e importantes para o processo de transformação de aplicações centralizadas em distribuídas. O trabalho nessa aplicação levou a pesquisas nas áreas de sistemas distribuídos, desenvolvimento de aplicações distribuídas, redes de computadores, processamento paralelo e em *makes* paralelos e distribuídos.

Neste capítulo, é discutida a motivação por trás da criação do MakeD e quais os objetivos a serem alcançados no trabalho. Ainda é realizada uma descrição geral da ferramenta *make* e uma breve descrição do trabalho desenvolvido.

1.1 Motivação

Nas últimas décadas, a capacidade de processamento e de armazenamento das máquinas cresceu num ritmo acelerado de modo a suprir as demandas geradas pelo mercado. Os avanços tecnológicos proporcionam o aumento do poder computacional acompanhado de novas funcionalidades e redução no preço relativo.

Por outro lado, as aplicações são projetadas procurando aproveitar o potencial do *hardware* e oferecer um grande número de recursos e funcionalidades ao usuário final, automatizando muitos processos que antes requeriam tempo e custo adicional. O preço desta automação é refletido na complexidade e no aumento do código fonte dos programas. Para gerenciá-los, é indispensável o uso de técnicas de projeto, como a divisão em módulos, projeto *top-down* e a criação de um grande número de funções, cada qual implementando uma tarefa específica.

Durante a construção, manutenção e depuração de programas, o *make* [Fel79, Pal87] (Seção 1.2) é uma ferramenta imprescindível que automatiza o processo de compilação recompilando somente aquilo que foi alterado desde a última versão gerada. O ganho de tempo e o conseqüente aumento da produtividade do programador são consideráveis. Entretanto, o *make* concentra o esforço do processamento numa única máquina, e embora reduza o tempo de gerência, não altera o tempo de compilação dos módulos e geração do produto final, ainda bastante alto e com intenso uso de CPU para grandes projetos.

Uma possível solução para essas limitações do *make* motivou-nos a estender suas funcionalidades aliadas ao desafio de torná-lo uma ferramenta distribuída cliente/servidor prática e de uso geral. Assim, além de contribuir para o melhoramento do próprio processo de *make*, tornando-o mais eficiente com o uso do processamento distribuído na rede, é possível verificar o ganho de desempenho obtido e ainda discutir aspectos práticos sobre o trabalho de desenvolvimento da aplicação distribuída MakeD a partir da versão centralizada convencional.

A configuração física da rede num ambiente de trabalho convencional geralmente segue o modelo cliente/servidor composto por um servidor com alta capacidade de processamento e vários clientes com média/alta capacidade de processamento (figura 1.1). A utilização dos recursos da rede depende das atividades do grupo de usuários e dos *softwares* disponíveis.

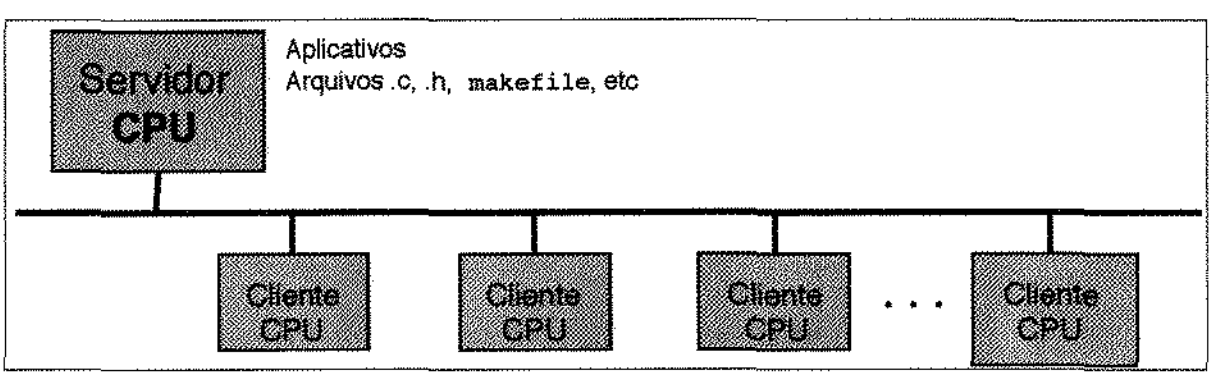


Figura 1.1: Ambiente de rede convencional

Observa-se que uma estação cliente tipicamente apresenta um padrão de uso de CPU com curtos períodos de intensa utilização (carga de aplicativos, desenho de interface gráfica, formatação de texto, manipulação de imagens) e longos períodos de inatividade ou baixa utilização (figura 1.2). No geral, a média de uso da CPU num período é muito pequena.

Os percentuais de uso de CPU dos gráficos seguintes são apenas suposições para dar noção da utilização média da capacidade de processamento das estações.

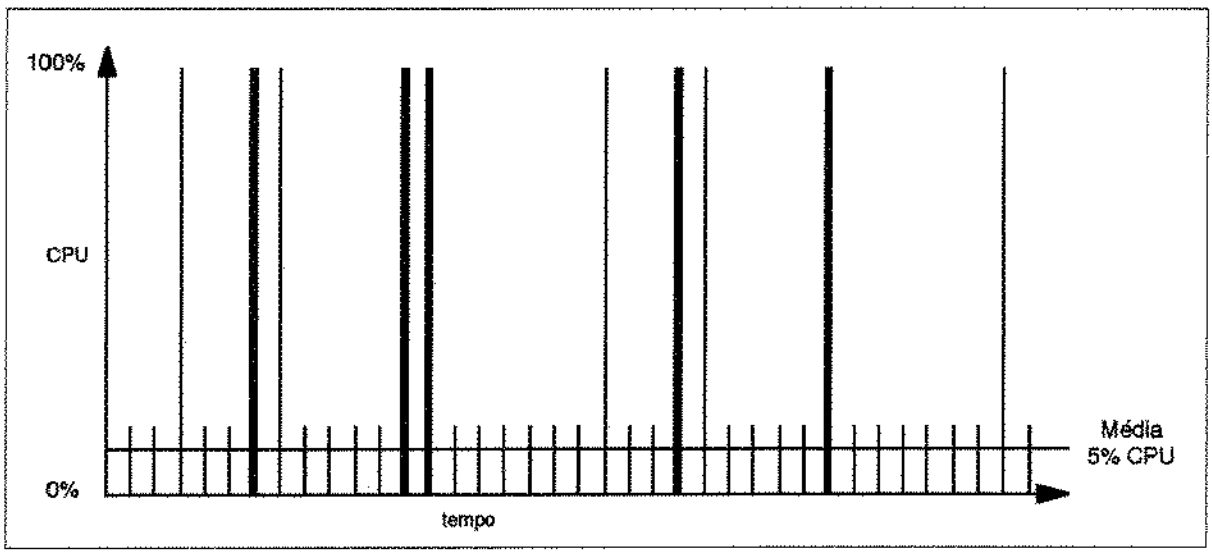


Figura 1.2: Uso de CPU para um cliente genérico

Por outro lado, o servidor de aplicações recebe freqüentes requisições de serviços de clientes (acesso a banco de dados, fornecimento de aplicativos, serviços Internet), com razoável carga média num dia de trabalho (figura 1.3).

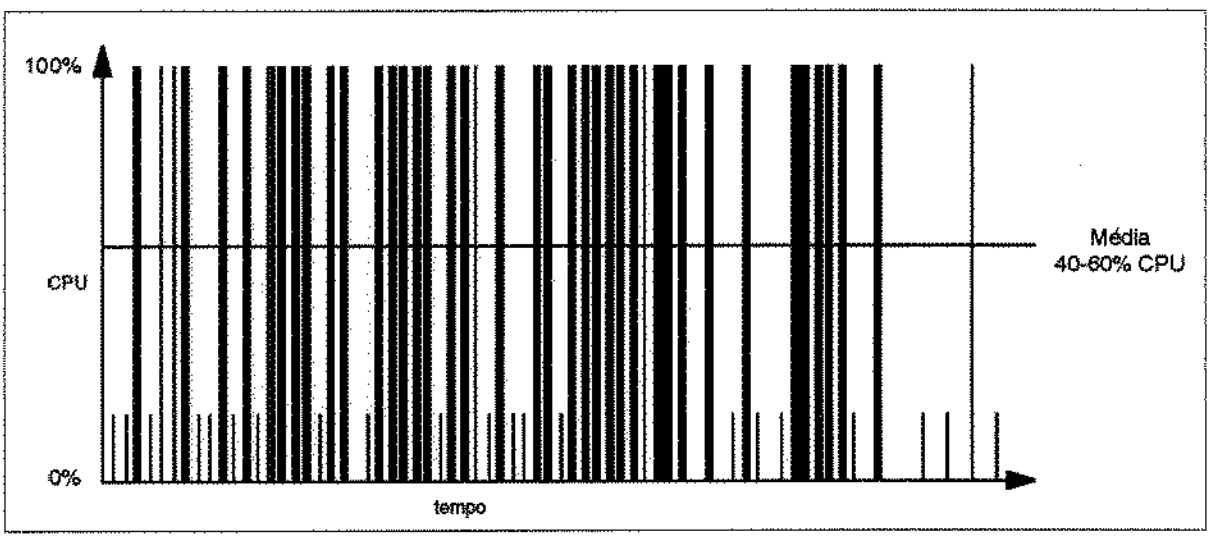


Figura 1.3: Uso de CPU de um servidor de aplicações

Num ambiente típico de desenvolvimento, cada programador possui geralmente uma cópia (no servidor) dos módulos que está alterando, a fim de evitar conflitos de versões e alterações simultâneas no mesmo arquivo. Boa parte de seu tempo é gasto em tarefas de

edição, depuração, testes, leitura e gravação de arquivos que consomem pouca CPU, com rápidos e variáveis picos de alta utilização. Entretanto, a tarefa de compilação e ligação de módulos, realizada no cliente e geralmente usando o *make*, exige longo tempo de intenso processamento e interfere no desempenho dos demais processos locais (figura 1.4). O uso da CPU num dia de trabalho é uma sucessão de períodos de baixa carga com períodos contínuos de elevada carga.

Quanto ao uso de CPU, uma estação cliente programador tem comportamento ora de cliente genérico ora de servidor. Como cliente genérico, a taxa média de uso de CPU é muito baixa, mas como servidor de CPU, executando *make* por exemplo, sua CPU é totalmente exigida, possivelmente com longo tempo para as compilações. Uma conseqüência imediata é o aumento da ociosidade do programador, enquanto ao seu redor os demais programadores estão tipicamente nos longos ciclos de edição de programas e as respectivas estações com alta ociosidade.

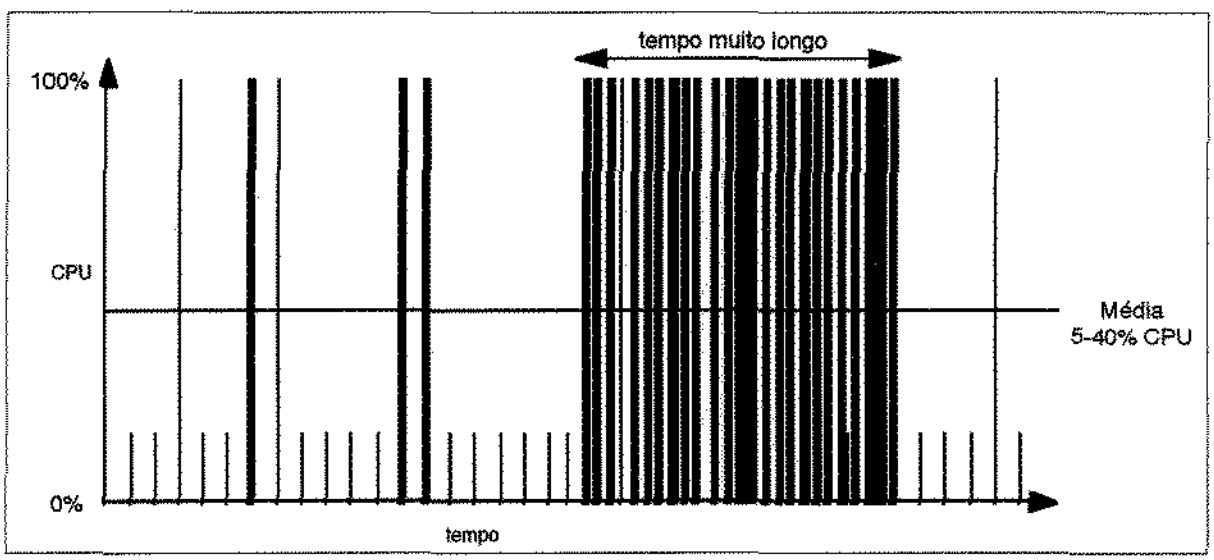


Figura 1.4: Uso de CPU de um cliente programador

Uma solução para diminuir o tempo do processamento local no cliente é utilizar os demais clientes como servidores de CPU e usar uma nova ferramenta (MakeD) para distribuir as tarefas de compilação entre eles.

Ao utilizar as estações da rede como servidores de CPU para executar tarefas, o MakeD viabiliza a possibilidade de se fazer bom uso da capacidade de processamento disponível, contribuindo para balancear a utilização dos recursos. A figura 1.5 apresenta uma utilização de CPU de um cliente que funciona também como servidor de CPU para outros clientes. Normalmente, além da carga do próprio cliente programador, vai existir uma carga devido ao uso do servidor de CPU pelos outros clientes. Dessa forma, a ociosidade dos clientes pode ser aproveitada devido a característica de trabalho do programador.

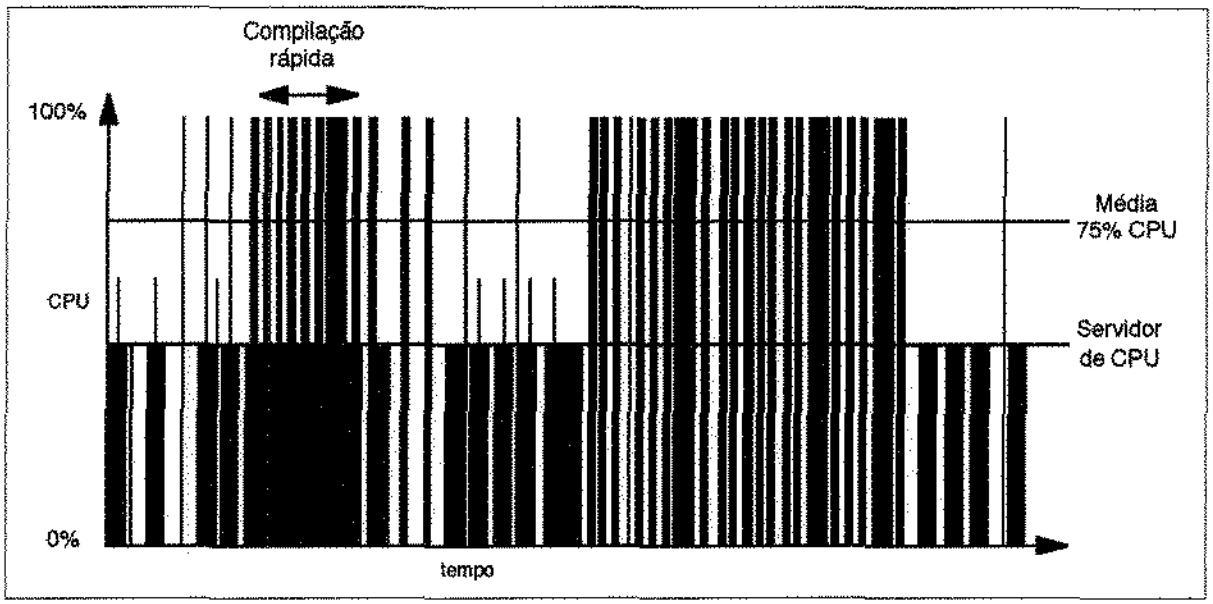


Figura 1.5: Uso de CPU para um cliente servidor de CPU

Num ambiente típico de desenvolvimento com vários clientes de média/alta capacidade de processamento, o MakeD pode aproveitar a capacidade ociosa dos clientes e usá-los como servidores de CPU de baixa prioridade para formar um ambiente ideal de compilações distribuídas com pouca interferência no desempenho das atividades dos usuários locais.

Para melhor entender a seção que descreve este trabalho (Seção 1.3), faz-se necessária uma abordagem geral sobre o *make*, já que ele é a base do trabalho desenvolvido.

1.2 Caracterização da ferramenta *make*

Diversas são as tarefas envolvidas no processo de viabilização de um projeto de médio ou grande porte: compilação e ligação, controle de versões, *backups*, testes, manutenção e distribuição de arquivos. A complexidade do trabalho exige o efetivo uso de ferramentas para sua automação, reduzindo sensivelmente o tempo despendido com tarefas repetitivas, aumentando a disponibilidade dos participantes para a especificação, análise e desenvolvimento.

Originalmente desenvolvido no sistema Unix, o programa *make*[1] facilita o desenvolvimento de projetos que consistem de vários arquivos, que podem ser gerados de diversas maneiras. Usos habituais do *make* incluem manutenção e atualização de programas e bibliotecas (com suporte a versões), execução de baterias de testes, instalação e distribuição de programas.

[1] *Make* é usado nos sistemas Unix desde 1975. Hoje pode ser considerado mais um padrão de ferramenta, tal o número de implementações.

O *make* requer que o programador descreva em um arquivo de regras (usualmente `makefile`) as relações de dependências entre os arquivos e as tarefas a realizar para recriar automaticamente um arquivo inexistente (ou desatualizado) a partir de outro. A partir do `makefile`, o utilitário determina automaticamente quais partes de um projeto precisam ser refeitas e executa os comandos necessários.

O *make* foi projetado para simplificar e automatizar o trabalho do desenvolvedor e mantenedor de projetos relativamente complexos, mas tem se mostrado ser uma ferramenta muito útil para diversos problemas diferentes dos previstos originariamente. É de uso geral, podendo gerenciar qualquer produto final cuja construção é definida por várias etapas intermediárias. Muitos programas e versões do próprio sistema Unix, são mantidas através do *make*.

Embora o MakeD continue sendo uma aplicação genérica, o enfoque neste trabalho, presente principalmente nas explicações e exemplos, será voltado para a maior aplicação do *make*, que é a compilação de programas com o objetivo de gerar o produto final.

Tendo apresentado uma visão geral do *make*, passa-se agora para a descrição do trabalho desenvolvido.

1.3 Descrição do trabalho

A implementação de aplicações distribuídas, que é uma tendência em plena ascenção, visa explorar os recursos computacionais da rede que estão interconectados por um meio físico de alta velocidade.

A construção destas aplicações exige um nível de controle e complexidade muito maior em relação às aplicações centralizadas. Para o desenvolvimento eficiente, rápido e confiável de aplicações distribuídas, é necessário que o ambiente de suporte a programação forneça um grau de abstração razoável. Para tal, seria desejável que o ambiente tivesse propriedades como heterogeneidade e interoperabilidade, portabilidade, facilidade de uso, orientação a objetos, segurança, tolerância a falhas, agrupamento de objetos e transparência quanto à localização. Estes requisitos são detalhados em [Cia94], e padrões como DCE (*Distributed Computing Environment*) [OSF90] proposto *pela Open Software Foundation* (OSF) e CORBA (*Common Object Request Broker Architecture*) [OMG97] proposto pelo *Object Management Group* (OMG), estão buscando suprir essas necessidades, com promessas no sentido de se estabelecerem como uma base para a efetiva integração de diferentes ambientes, em especial com IIOP (*Internet Inter ORB Protocol*) [OMG97].

Ambientes de suporte a programação distribuída têm se mostrado um meio eficaz, viável e economicamente atrativo para desenvolver aplicações dessa natureza. Dentre eles podemos citar os sistemas operacionais distribuídos como Chorus [Cho90], Mach [Acc86] e V [Che88], as linguagens orientadas a programação distribuída como Cm Distribuído [Gon94] e RMI [Sun97], e ainda os padrões DCE [OSF90] e CORBA [OMG97].

O atendimento aos requisitos relacionados tem forte impacto no desempenho, sempre citado como problema sério dos sistemas distribuídos [Mul90]. Além disso, geralmente deve-se abrir mão da portabilidade da aplicação desenvolvida nestes sistemas [Cia94]. Por outro lado, as linguagens para programação distribuída já estão se tornando realidade devido à atenção recebida nos últimos anos, e os padrões DCE e CORBA permitem o desenvolvimento de aplicações distribuídas com expectativas crescentes de interoperabilidade.

O MakeD, tratando-se de uma extensão da aplicação *make* existente, não depende de nenhum dos ambientes citados acima. Seu desenvolvimento é baseado nas ferramentas disponíveis nos sistemas Unix, como NFS (*Network File System*) [Sun87], RPC (*Remote Procedure Call*) [Bir84, Sun94], XDR (*External Data Representation*) [Sun87a], *threads* [Pow91, Eyk92] e o protocolo TCP/IP. NFS garante um sistema de arquivos comum entre as estações; RPC é responsável pelo padrão para clientes requisitarem serviços aos servidores; a linguagem XDR permite uma representação portável dos tipos de dados que são transmitidos entre diferentes arquiteturas, enquanto as *threads* permitem a execução simultânea de várias seqüências de instruções dentro de um programa.

Essas ferramentas usadas no desenvolvimento do MakeD suportam amplamente a programação distribuída cliente/servidor a baixo custo e estão presentes na maioria dos ambientes Unix, facilitando a portabilidade. Apesar de não possuírem o mesmo nível de abstração como os ambientes citados acima, várias propriedades como tolerância a falhas, podem ser implementadas no nível de aplicação ou mesmo a nível de sistema operacional.

O trabalho realizado pode ser dividido em três etapas principais. A primeira cobriu o estudo do código fonte do *make*. Foi necessário compreender seus complexos e extensos algoritmos, principalmente o de busca em grafos e os que fazem o controle e execução concorrente de tarefas. A segunda consistiu da elaboração de uma estratégia de modificação do *make* para torná-lo distribuído. A última fase foi marcada pela análise de desempenho da aplicação distribuída comparada à centralizada e a outro *make* distribuído pesquisado.

Ao utilizar os recursos da rede, o MakeD pode explorar não apenas o alto grau de paralelismo que o ambiente oferece, mas também a multi-tarefa em cada sistema, através da combinação dos recursos de RPC e *threads*.

A técnica de implementação usada é baseada no modelo cliente/servidor, compreendendo o MakeD duas partes independentes:

- **Controlador *make***: cliente da aplicação distribuída que executa na máquina local. É, na verdade, o *make* modificado que faz também requisições de processamento de tarefas (através de RPCs síncronas) aos servidores remotos. Decide em qual estação a tarefa será processada baseado na capacidade da estação e na divisão da carga de trabalho.

- **Servidor de Processamento**: servidor da aplicação distribuída. Faz as tarefas pedidas pelo controlador de *make* e devolve os resultados. É executado como um *daemon*[2] em todas as máquinas da rede que participam do processo de *make*; é *multithreaded*, recebendo e tratando pedidos de múltiplos controladores *make* ao mesmo tempo.

Todas as funcionalidades do *make* são mantidas no MakeD. Apesar da distribuição do processamento, o usuário tem a visão de um *make* centralizado, essencialmente quanto aos resultados e mensagens informativas do que ocorre durante o processamento. A diferença perceptível ao usuário é a agilidade no processamento global.

Nesta implementação, o MakeD tem como plataforma de *hardware* um conjunto de *workstations* da *Sun*[3] gerenciadas pelo sistema operacional Solaris 2.5[4] (*System V*).

Os detalhes sobre a implementação do MakeD são apresentados no Capítulo 4.

1.4 Um pouco de história

No início de 1995, estava sendo desenvolvido um projeto no Laboratório A-HAND (Instituto de Computação – Unicamp) que já contava com 100.000 linhas de código em Visual C++ e envolvia uma equipe variável entre oito e doze programadores. O Visual C++ gera o `makefile` automaticamente e para garantir consistência o `makefile`, contém, muitas vezes, mais interdependências do que o necessário. Para o projeto em questão, o `makefile` gerado possuia em torno de 100Kbytes de informações. Algumas sessões de *make* realizadas nas estações distintas demoravam duas horas. Outras implicavam em poucas recompilações de arquivos pequenos, caso em que a leitura e análise do `makefile` consumia parte considerável do tempo gasto.

As observações destes fatos diários motivou-nos a buscar alternativas mais eficientes. O MakeD acelera as compilações usando a ociosidade média nas estações clientes. A proposta do servidor de *make*, descrito no Capítulo 6, otimiza o processo de leitura e análise dos `makefiles` compartilhados por múltiplos usuários participantes do mesmo projeto.

1.5 Organização da tese

Por se tratar de um trabalho que envolve a aplicação *make* e a programação cliente/servidor em rede, é preciso descrever a aplicação base e a solução adotada, e analisar o desempenho comparado. Assim, os próximos capítulos são organizados da seguinte forma:

O Capítulo 2 descreve as características da ferramenta *make* e suas facilidades para escrever o arquivo de regras.

[2] Processo servidor que fica à espera de requisições de serviços.
[3] Sun Microsystems, Inc.
[4] SunOS 5.5 Operating System.

O Capítulo 3 é uma revisão bibliográfica relativa a pesquisas sobre programas *make* distribuídos e paralelos, e outras aplicações semelhantes.

O Capítulo 4 é dedicado à estrutura interna do GNU *Make*, a implementação que serviu de base para o trabalho. São descritos seus algoritmos, estruturas de dados e detalhes de implementação.

No Capítulo 5 é apresentado o trabalho desenvolvido sobre o *make* distribuído. Além da descrição detalhada da solução do problema, são feitas também comparações entre o MakeD e alguns entre os descritos no Capítulo 3.

O Capítulo 6 apresenta alguns aspectos práticos sobre a transformação de aplicações centralizadas em distribuídas enfocando um pouco das dificuldades, cuidados e vantagens envolvidos, baseados na experiência com o MakeD. Propõe também um servidor de *make*, uma extensão para o MakeD.

No Capítulo 7 são apresentados os testes e a análise de desempenho do MakeD comparada ao *make* original e a outro *make* distribuído descrito no Capítulo 3.

Por fim o Capítulo 8 é reservado às conclusões do trabalho e possíveis extensões futuras.

Capítulo 2

A FERRAMENTA *MAKE*

O *make* é uma ferramenta largamente usada para manter e atualizar programas e bibliotecas em um ambiente de desenvolvimento, além de instalar e atualizar versões de *softwares* na administração de sistemas. Pode ser facilmente utilizado para diversas aplicações, pois manipula relacionamentos gerais entre arquivos e comandos. Atualmente muitas aplicações e versões do próprio sistema Unix são mantidas pelo *make*.

Este capítulo fornece uma visão geral do *make* e apresenta algumas facilidades usadas na manutenção de projetos.

2.1 Definições

O *make* usa um arquivo de configuração (o nome comumente usado é `makefile`) onde o usuário especifica as relações de dependências entre os arquivos e os comandos a serem executados para atualizá-los. O formato para as regras no `makefile` é mostrado na figura 2.1, juntamente com um exemplo simples.

Um *target* é geralmente um arquivo a ser gerado. Pode ser também uma ação a ser executada, tal como `clean` no exemplo abaixo. As *dependências* se referem aos arquivos dos quais o *target* depende. É a partir das dependências que o *target* é construído com a execução dos *comandos* das linhas seguintes. As dependências por sua vez podem ser novos *targets*, de maneira que o *make* verifica recursivamente cada uma delas.

Uma *regra* pode ter mais do que um comando. Ela diz ao *make* duas coisas: quando os *targets* estão desatualizados, e como atualizá-los quando necessário através dos comandos. O critério para estar desatualizado é especificado em termos das dependências: um *target* está desatualizado se não existe ou se qualquer uma de suas dependências for mais recente

(por comparação da última data de modificação). Assim, sempre que uma ou mais dependências são alteradas, o conteúdo do respectivo *target* torna-se inválido.

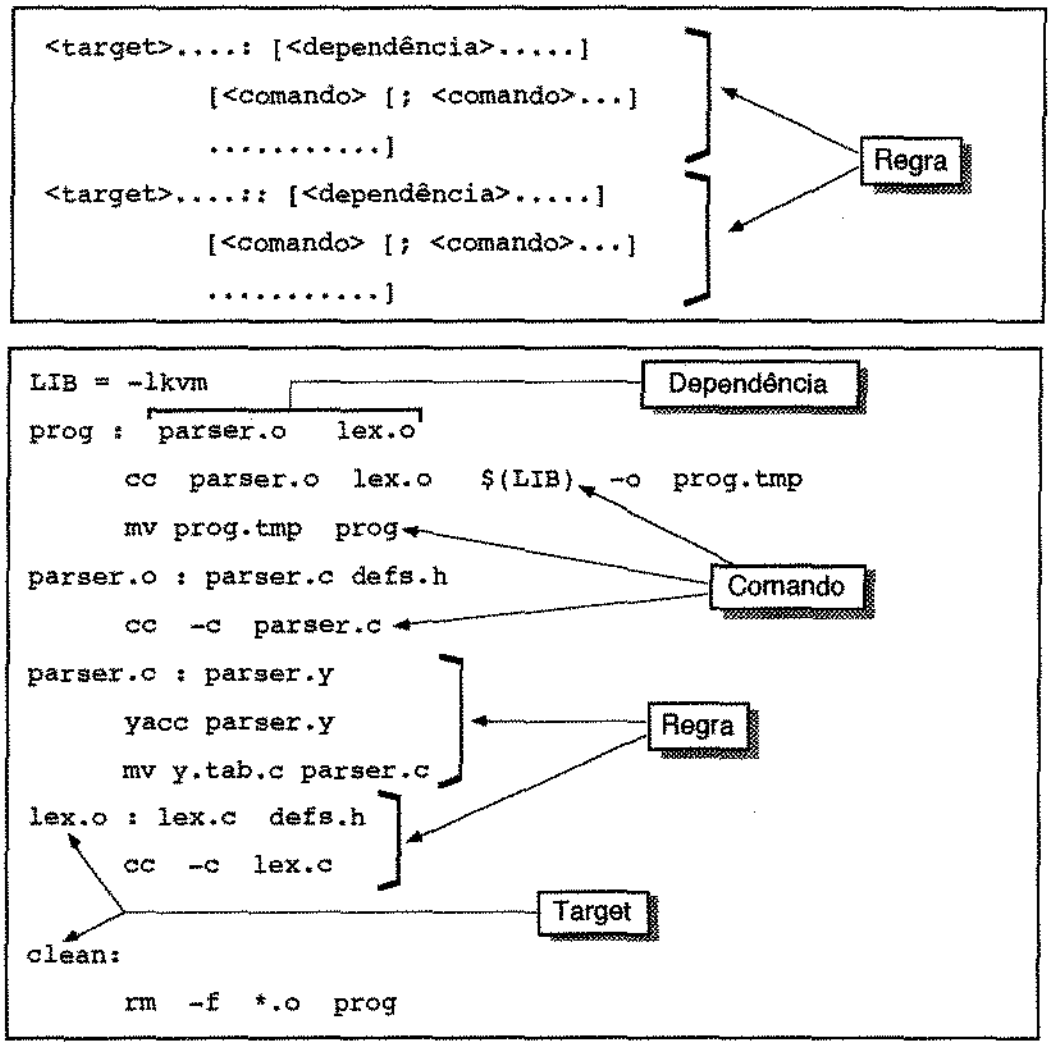


Figura 2.1: Formato das regras e um exemplo de `makefile`

Além das regras, o arquivo pode conter uma mistura de entradas de definição de variáveis (macros)[1], comentários e diretivas: `define`, condicional (`ifdef`, `ifndef`, `ifeq`, `ifneq`, `endif`, etc) e `include` (para inclusão de outros `makefiles`).

2.2 Visão Geral

A ferramenta *make* se baseia em algumas premissas:

[1] Variáveis são também chamadas de macros em várias versões de *make*.

- Muitos sistemas são compostos por vários arquivos (um programa em linguagem C possui módulos fonte e arquivos de cabeçalho; um documento Tex/nroff possui capítulos, tabelas, índices, macros);
- Um arquivo depende de outro e deve ser recriado, caso este seja alterado (se um capítulo for alterado, o índice deve ser reconstruído; se um arquivo fonte for editado, incluindo quaisquer arquivos de cabeçalho, o módulo objeto correspondente deve ser refeito);
- As dependências podem se estender por mais de um nível (fonte → objeto → biblioteca → programa ...);
- Com freqüência a reconstrução pode ser automatizada (no caso de programas, refazer um módulo objeto requer uma compilação sem intervenção/iteração do usuário).

Quando um arquivo é alterado, o responsável pela manutenção do sistema poderia:

- Identificar manualmente as atualizações necessárias;
- Usar um *script* que atualiza todos os componentes;
- Especificar uma única vez a relação de dependências, e usar uma ferramenta como *make* para refazer automaticamente apenas os componentes afetados.

O arquivo `makefile` é usado apenas para que o *make* construa em memória um grafo de dependências direcionado e hierárquico, sendo que cada nó está associado a um *target* e contém os comandos que constroem um *target* ou executam uma ação.

A figura 2.2 apresenta trechos de programa que são transformados e combinados para produzir outros arquivos, incluindo o produto final. O grafo de dependências é a visão do *make* sobre o problema. Através dele, o *make* determina os arquivos que foram alterados desde a última atualização do projeto e emite os comandos necessários para atualizá-los.

No processamento dos nós do grafo[2] é usado um algoritmo de busca em profundidade que ativa a atualização de um nó pai somente se algum de seus filhos for mais recente e após o término da eventual atualização de todos os filhos. Dependendo da plataforma, o *make* permite também a geração de vários *targets* concorrentemente, e nesse caso podem ser necessárias várias varreduras no grafo (ou sub-grafos).

Para *makes* com suporte a execução concorrente de tarefas, o algoritmo usado na busca em profundidade é complexo e eficiente no que diz respeito ao controle de concorrência utilizado. A complexidade vem da necessidade de varrer o grafo (ou sub-grafos) recursivamente várias vezes, garantindo e controlando a concorrência de processamento dos nós (geração de *targets*) através do uso de *flags* de controle em cada nó. O número máximo de *jobs* executando simultaneamente é determinado pelo usuário e pela carga média do sistema. Num dado instante, só existe um comando de cada nó em processamento sendo

[2] Processar um nó significa executar seus comandos em seqüência e gerar um *target* atualizado.

executado (até mesmo porque um comando pode depender do resultado do anterior) e, para cada nó é obedecida a seqüência na execução dos comandos.

A execução de comandos é realizada via processos filhos (em Unix, *fork()* seguido de *exec()*). Cada comando dá origem (diretamente) a um processo[3].

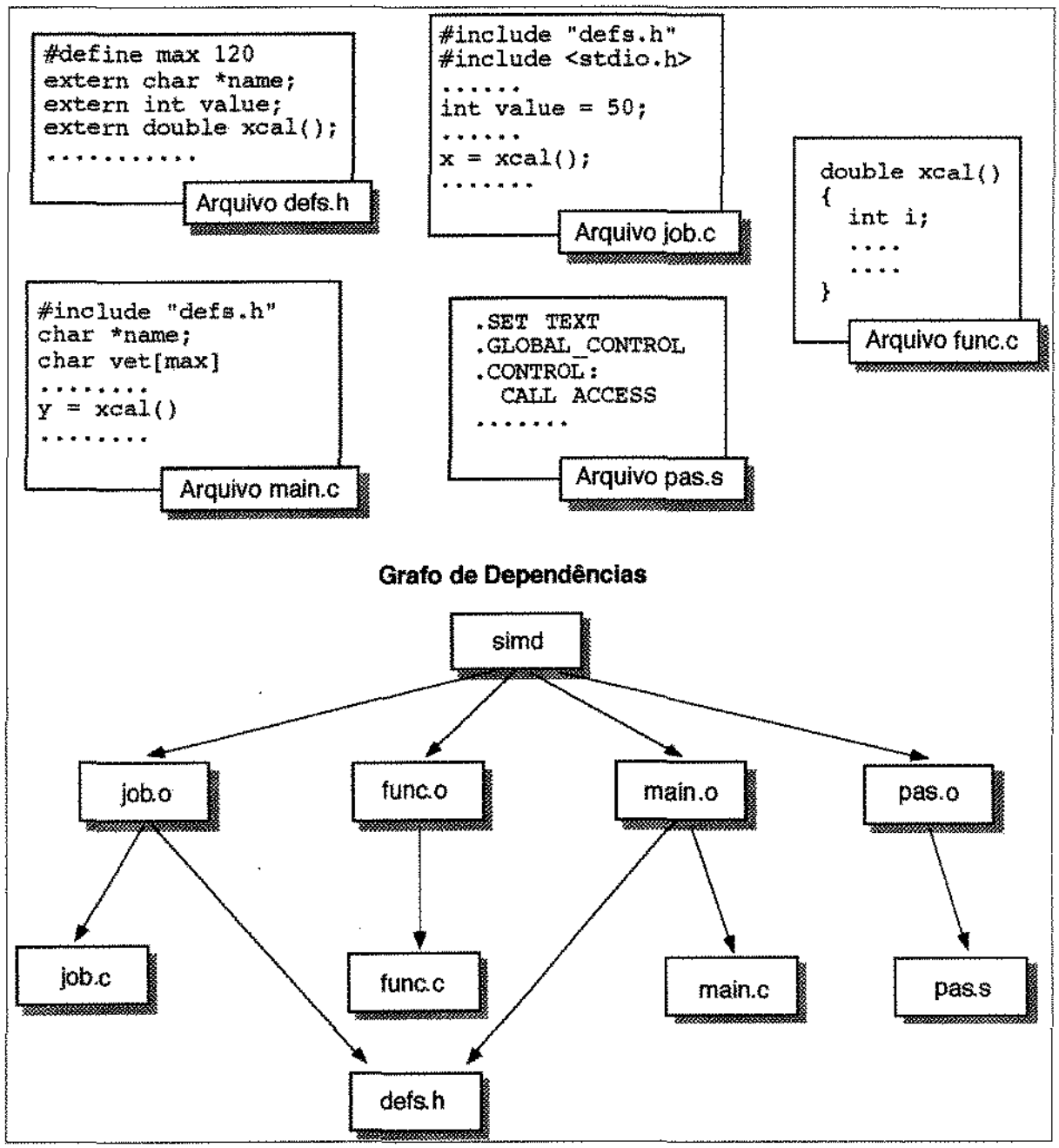


Figura 2.2: Exemplos de relação de dependências

[3] Como será visto, o processo pode criar outros.

2.3 Regras no `makefile`

Há dois tipos de regras que podem ser usadas:

- Regras explícitas: dizem quando e como atualizar arquivos dados por nomes exatos. O exemplo da figura 2.1 foi escrito usando regras explícitas.
- Regras implícitas: dizem quando e como atualizar uma classe de arquivos usando nomes genéricos. Podem ser definidas escrevendo regras de sufixo ou regras de padrão. Das regras abaixo, a primeira é escrita usando uma regra de sufixo e a segunda uma regra de padrão:

```
# diz como obter targets '.o' apartir de dependências '.c'.
.c.o :
        $(CC) -c $(CFLAGS) -o $@ $<

# realiza casamento de padrão; diz como obter o arquivo X.o a partir das
# dependências X.c e defs.h, para qualquer X
%.o : %.c defs.h
        $(CC) -c $(CFLAGS) -o $@ $<
```

Os comandos acima usam as variáveis automáticas (Seção 2.4) `$@` e `$<` para substituir os nomes de arquivos *target* e fonte para cada caso em que as regras se aplicam.

A figura 2.3 apresenta um `makefile` (usando regras implícitas) que descreve a relação de dependências para os arquivos da figura 2.2. O *target* (ou ação) `clean` não cria arquivo algum e realiza somente tarefas administrativas. Pode ser observado como as regras implícitas simplificam e reduzem o número de declarações de dependências no `makefile`.

Regras de sufixo são um método limitado e obsoleto para definição de regras implícitas, mas ainda são mantidas para compatibilidade com `makefiles` antigos. As regras de padrão são mais claras e gerais [Sta95].

A sintaxe de dependência usando "`::`" (figura 2.1) indica que um *target* depende de vários conjuntos de dependências, e uma seqüência diferente de comandos pode estar associada a cada conjunto. No exemplo abaixo, se quaisquer dos arquivos à direita de "`::`" estiver desatualizado com respeito ao *target* `lib`, a seqüência de comandos associada é executada.

```
lib :: src1.o src2.o
        ar r lib $?
lib :: src3.c
        cc -c src3.c ; ar r lib src3.o ; rc src3.o
```

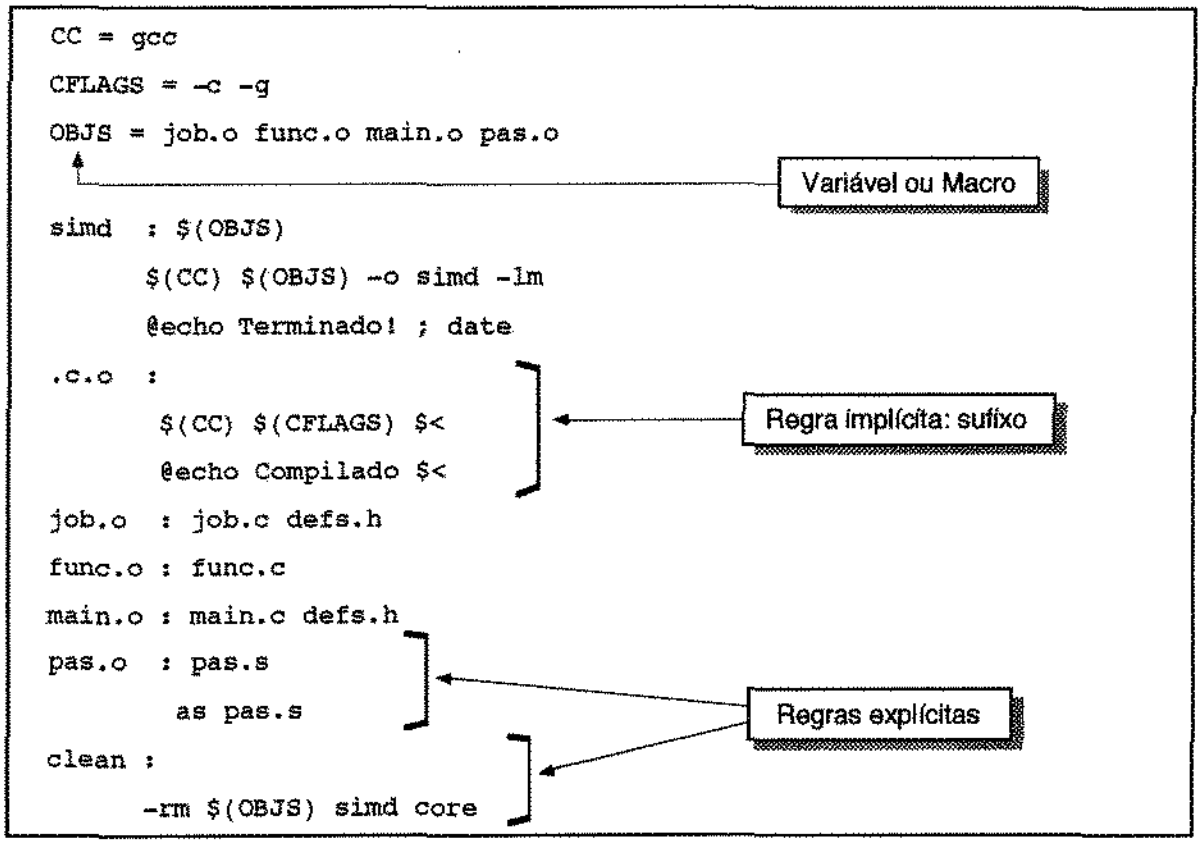


Figura 2.3: `Makefile` com regras implícitas

2.4 Variáveis

As variáveis podem ser usadas em qualquer ponto do `makefile` para abreviar listas de *targets* ou expressões, ou para substituir opções de compiladores, nomes de programas ou seqüências de texto que precisam ser repetidas. Podem ser usadas também para derivar listas de arquivos objeto a partir de arquivos fonte.

O *make* mantém um conjunto de variáveis predefinidas (variáveis gerais) para facilitar a construção de `makefiles`. Algumas são nomes de programas (como `CC`, `AS`, `LEX`) e outras argumentos de programas (como `CFLAGS`, `ASFLAGS`, `CPPFLAGS`). Também mantém um conjunto de variáveis dinâmicas (variáveis automáticas) por *target*, muito usadas em definições de regras implícitas, como no exemplo da figura 2.3. Os valores dessas variáveis são baseados no *target* e dependências, sendo recalculadas a cada aplicação da regra.

As estruturas de dados usadas para armazenar estas e outras variáveis são apresentadas no capítulo 4.

2.5 Manutenção de projetos usando o *make*

O *make* é util para manter um projeto segmentado em partes seguindo uma hierarquia de diretórios, por exemplo, bibliotecas, arquivos binários, fontes, objetos, de cabeçalho e documentação, residindo nos respectivos diretórios

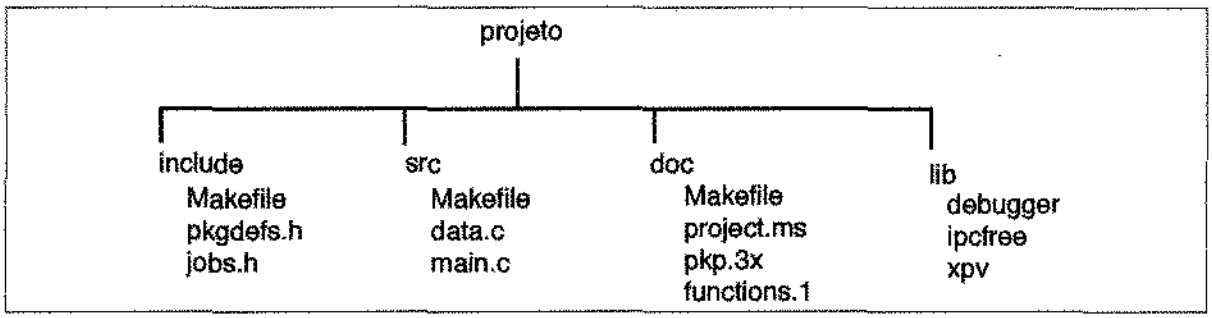


Figura 2.4: Hierarquia de diretórios de um projeto

Cada diretório pode conter um `makefile`, mas é necessário um `makefile` no diretório principal (*root* `makefile`) para fornecer entradas de *targets* e gerenciar o projeto como uma entidade única. A técnica de uso recursivo do *make* é util para facilitar a manutenção em projetos divididos em subsistemas seguindo a hierarquia de diretórios.

À medida que o projeto cresce, cresce também a necessidade de consistência e facilidade de uso dos `makefiles`. Assim, é importante adotar convenções para nomes de variáveis, *targets* e opções de compilação a serem usadas nos `makefiles`. Muitas variáveis podem ser definidas no *root* `makefile` e exportadas para os demais. Outra consideração é o tempo gasto para a leitura e análise dos `makefiles` a cada sessão de *make*, que é irrelevante somente se há muitas atualizações a serem feitas.

Além de construir o projeto completo, o *make* pode ser usado para instalar cópias dos módulos em outros sistemas de arquivos para integração e distribuição, realizar testes, *backups*, etc. Inclusive possui suporte a um sistema como o SCCS (*Source Code Control System*) com regras especiais para a recuperação automática de arquivos com várias versões.

Capítulo 3

REVISÃO BIBLIOGRÁFICA

O propósito deste capítulo é apresentar um *survey* da bibliografia estudada no que se refere principalmente a implementações paralelas e distribuídas de *make*. São ressaltados pontos importantes quanto a funcionalidades, características e implementação de alguns *makes* existentes e em desenvolvimento, sem entrar em muitos detalhes, para os quais pode-se consultar as referências.

Devido à precária literatura a respeito de *makes* com características distribuída e paralela, grande parte da bibliografia consultada é baseada em manuais de referência, relatórios técnicos, análise de partes do código fonte e documentação acompanhante.

Uma comparação mais detalhada deste trabalho com alguns *makes* aqui apresentados é realizada no Capítulo 5 (Seção 5.4). Neste mesmo capítulo, é apresentado também um quadro comparativo relacionando todos os *makes* pesquisados, inclusive o *make* distribuído desenvolvido neste trabalho.

3.1 Introdução

Os meios de comunicação de alta velocidade e a capacidade embutida nas estações de trabalho, em termos de CPU e memória, têm influenciado pesquisas nas áreas de processamento distribuído e paralelo e, conseqüentemente, implementações de *software* dessa natureza.

Os benefícios advindos do fato de se poder utilizar mais de um processador ou máquina com a aplicação *make* [Fel79] são facilmente perceptíveis. Até mesmo pela sua popularidade, uso e existência de ferramentas como NFS [Sun87], RPC [Bir84] e PVM [Beg94], é possível

integrar computadores para realizar as tarefas de compilação de um *software* com baixo custo de implementação e considerável aumento na velocidade de processamento global.

Nessa linha, existem alguns programas *make* que exploram o paralelismo ou recursos de rede, embora todos tenham um objetivo comum: agilizar e automatizar o processo de compilação dos módulos de um sistema de forma transparente ao usuário. A maioria dos programas estende uma versão simplificada do GNU *Make* [Sta95] para uma máquina paralela ou para um ambiente de rede distribuído. Alguns seguem o modelo de passagem de mensagens do PVM. Portanto, na Seção 3.2 é apresentada a arquitetura PVM, e nas próximas seções são dadas visões gerais dos programas *make* pesquisados.

3.2 PVM

O PVM (*Parallel Virtual Machine*) [Beg94, Gei93, Don93] é um conjunto de ferramentas e bibliotecas que possibilita a utilização de computadores seriais e paralelos num ambiente de rede heterogênea como um único recurso computacional para a execução de aplicações.

O PVM permite ao usuário escrever aplicações onde uma coleção de computadores executa partes da aplicação em paralelo. Este grupo de computadores, com diferentes arquiteturas e representação de dados, pode ser considerado como uma máquina virtual paralela.

Usando o PVM, o usuário pode configurar sua própria máquina virtual, que pode coincidir com máquinas virtuais de outros usuários. A configuração de uma máquina virtual significa simplesmente relacionar os nomes dos computadores num arquivo que é lido quando o PVM é iniciado. As aplicações, que podem ser escritas na linguagem C ou Fortran, usam rotinas fornecidas pelo PVM para criar sub-tarefas que cooperam para resolver o problema em paralelo. Uma sub-tarefa pode criar mais sub-tarefas. A comunicação e sincronização entre elas é realizada pelo paradigma de envio e recebimento de mensagens cliente/servidor. A figura 3.1, extraída de [Gei94], apresenta a arquitetura do sistema PVM de maneira simplificada.

O sistema PVM é composto de duas partes: a primeira é um *daemon* que reside nos computadores que fazem parte da máquina virtual, e a segunda é uma biblioteca de funções contendo as rotinas usadas para a passagem de mensagens, criação de processos, coordenação de tarefas e modificação da máquina virtual. Programas de aplicação devem ser ligados (*linked*) com essa biblioteca para usar o PVM.

Do ponto de vista do usuário, o PVM é um computador com memória distribuída, programado através da passagem de mensagens explícitas a nível de usuário, onde a conversão de dados entre processos é feita usando o XDR (*External Data Representation*) [Sun87a] [Gei93].

Finalizada a exposição do sistema PVM, passa-se à apresentação da revisão da literatura relativa às ferramentas *make* distribuído e paralelo.

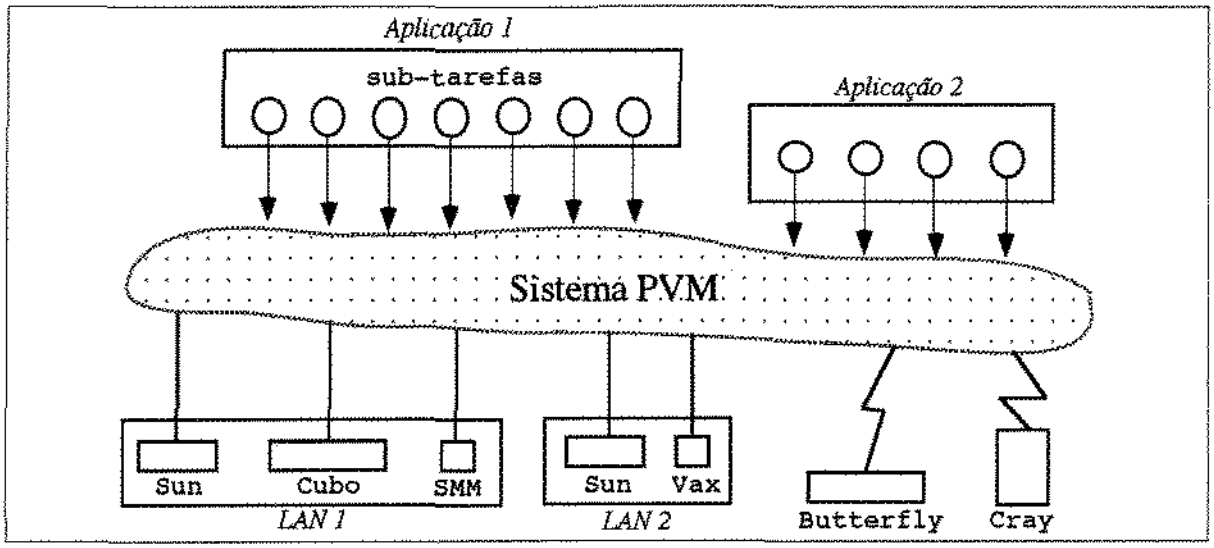


Figura 3.1: Arquitetura PVM

3.3 PVMmake

O *PVMmake* [Dev95] é uma aplicação implementada sobre o PADE (*Parallel Application Development Environment*) [Dev95], o qual usa os serviços do PVM. Em conjunto, o PADE e o PVMmake realizam a função de um *make* que executa compilações simultâneas em componentes heterogêneos de uma máquina virtual. O resultado pode ser uma mesma aplicação gerada para cada arquitetura. O PVMmake é chamado por [Dev95] de *make* paralelo, mas parte da função de *make* é controlada pelo PADE.

O PADE é um ambiente para desenvolvimento de aplicações paralelas que usa o mecanismo de passagem de mensagens do PVM. Ele fornece um ambiente integrado para todas as fases de desenvolvimento de uma aplicação PVM: edição, compilação e execução. A maior utilização do PADE é quando existe uma máquina virtual heterogênea com múltiplas arquiteturas e/ou múltiplos *file systems* e os componentes da aplicação sendo desenvolvida devem ser compilados em cada máquina, geralmente com diferentes diretivas e opções de compilação.

No processo de *make*, o PADE seleciona somente os arquivos que foram modificados desde a última atualização e invoca o PVMmake que os transfere (via passagem de mensagem PVM) para as máquinas remotas e emite, em cada uma delas, os comandos descritos no arquivo de configuração do PVMmake para que seja realizada a compilação local de todos arquivos. O PADE também centraliza os resultados das compilações nas diferentes arquiteturas para apresentação ao usuário.

O maior objetivo do PVMmake, em conjunto com o PADE, é fornecer ao programador a facilidade de desenvolver aplicações portáveis para vários sistemas, através da

automatização das compilações e testes em outras máquinas além daquela de desenvolvimento. Na verdade, o processo de atualização propriamente dito é realizado pelo *make* convencional de maneira centralizada em cada uma das máquinas remotas, gerando um *target* final para cada uma delas.

3.4 Lmake

O *Lmake* [Loi96] é um *make* paralelo[1], desenvolvido como trabalho de mestrado por Martin Loitz (*University Braunschweig* – Germany), que usa um *cluster de workstations* (máquina virtual PVM) para processar compilações simultâneas num sistema de arquivos comum fornecido via NFS. Fornece compatibilidade com *makes* padrão, salvo alguns problemas existentes com relação às opções.

O Lmake usa o LDS (*Lmake Daemon System*) para conseguir a execução remota de comandos e coletar informações de carga das estações. O LDS é um subsistema do Lmake formado por uma coleção de *daemons lmaked*, um em cada máquina, e o PVM é responsável pela troca de mensagens entre eles. O LDS também configura as estações do *cluster* e ativa os processos *lmaked* e *pvmd3*, sem necessidade de intervenção do usuário.

O esquema abaixo mostra a interação entre Lmake, LDS e PVM durante a execução de comandos. No caso de execução remota, o Lmake usa o PVM para entregar o comando (mensagem) ao *lmaked* apropriado, o qual ativa a execução, espera pelo seu término e devolve os resultados juntamente com a carga da máquina. Na execução local, o processo é idêntico, porém interno à estação.

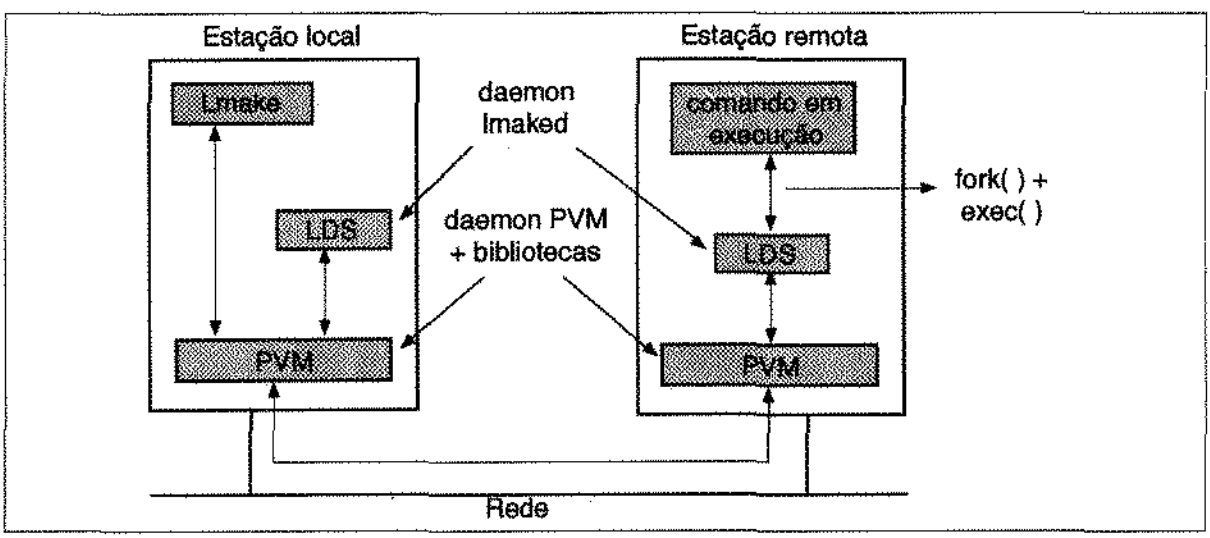


Figura 3.2: Interação entre lmake, LDS e PVM.

[1] Dito paralelo pelo fato de trabalhar num *cluster* de *workstations* (máquina virtual PVM).

O Lmake é baseado no GNU *Make* versão 3.67 e pode ser dividido numa parte serial e outra paralela. A parte serial é idêntica ao *make* tradicional, porém a execução dos comandos foi alterada para permitir seu processamento em paralelo. Ao invés dos comandos serem submetidos ao sistema operacional para execução, eles são escritos num arquivo temporário, juntamente com as dependências.

Na fase paralela, o grafo de dependências é construído através do arquivo temporário e o Lmake solicita às estações (usando mensagens PVM) o envio de suas cargas correntes. Baseado nelas, é feita a distribuição das tarefas, das quais somente uma pode estar ativa numa máquina num dado instante.

Cada *daemon lmaked* recebe a requisição e processa os comandos usando processos filhos (primitiva *fork()* seguida de *exec()*[2] [Ste92]). Ao término do serviço, é enviada uma mensagem de volta ao Lmake, incluindo a carga corrente da estação, para atualização no Lmake.

A escolha da máquina a executar um determinado comando é baseada no valor de sua última carga e em seu fator de velocidade, o qual é inversamente linear à velocidade real de uma estação ociosa e, definido por um número real pré-configurável pelo usuário. Por exemplo, se as máquinas A e B possuem os fatores de velocidade 1.0 e 2.0, respectivamente, então A é aproximadamente duas vezes mais rápida que B. O índice de disponibilidade das máquinas é obtido por:

```
índice(estação) = fator de velocidade * (1 + carga)
```

onde a carga é dada pelo número de *jobs* presentes na fila de execução no último minuto, obtida via *system call rstat()* do UNIX. Assim, a estação com menor índice é escolhida para executar um comando, pois o índice é inversamente proporcional à velocidade real da estação [Sch96].

3.5 Dmake

O Dmake (*Distributed Make*) [Dwy94, Duk94] é um *make* paralelo baseado no GNU *Make*, versão 3.69, que exige um sistema de arquivos compartilhado entre as estações e requer também pequenas alterações nos `makefiles` padrão. Pode trabalhar em conjunto com o DQS (*Distributed Queue System*) [Rev92] para usar o balanceamento de carga fornecido por este último.

As tarefas de compilações locais e remotas são feitas via um programa cliente/servidor adaptado ao `rsh` (*remote shell*) do UNIX. Para a compilação de um arquivo fonte, o Dmake passa o comando para o *spew* (cliente `rsh`) e este o envia (via `rsh`) para o *rspew*

[2] O processo criado pelo *fork()* executa outro programa usando uma das funções *exec()*. Os segmentos (texto, dados, etc) são substituídos pelo novo programa e o ID do processo não é alterado pelo fato de não haver a criação de um novo processo.

(servidor `rsh`), que por sua vez recebe os dados, processa a requisição e devolve os resultados. O *spew* retorna os resultados ao Dmake.

A figura 3.3 ilustra o procedimento usado pelo Dmake nas compilações remotas. Para as compilações locais o processo é idêntico, porém interno à estação.

A figura mostra que é usado um processo (*feeder*) para passar o comando e a configuração do ambiente (*environment*) ao `rsh`. Esta comunicação e a devolução dos resultados realizada pelo `rsh` são feitas usando o mecanismo de comunicação entre processos fornecido pelo *pipe* [Ste92] do UNIX.

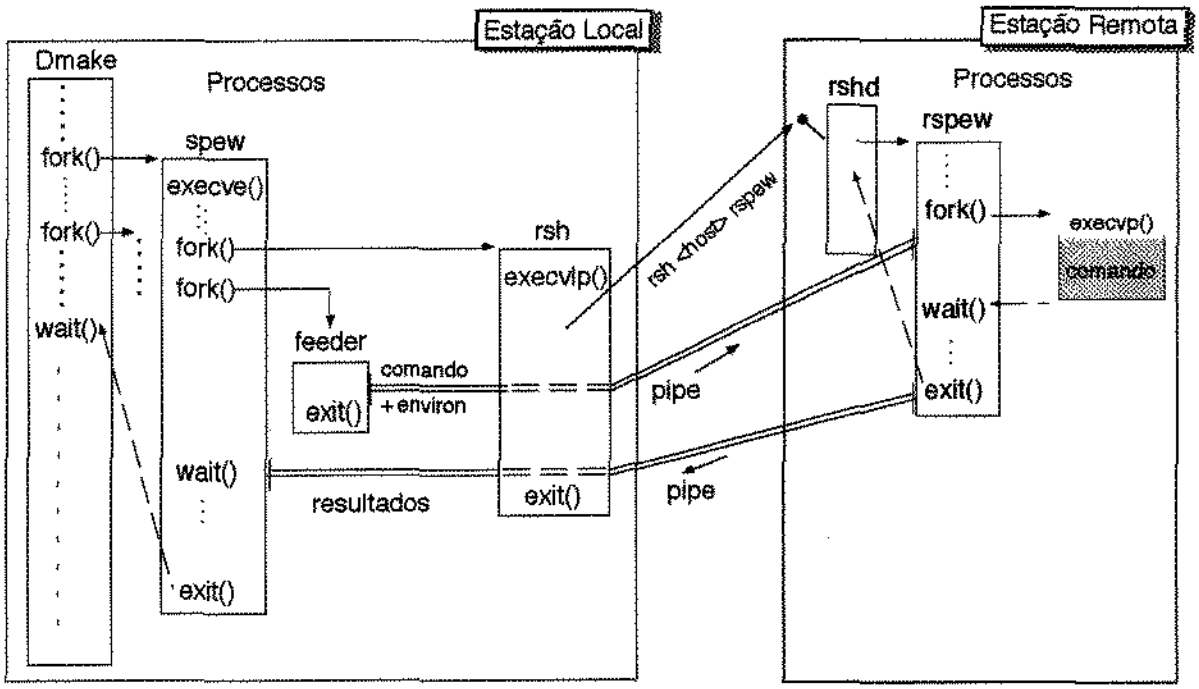


Figura 3.3: Processo de execução remota de um comando.

O conjunto das estações utilizadas é especificado pelo usuário, e sem o auxílio do DQS o Dmake as usará em ordem, tratando-as como uma lista circular (e.g., a quinta compilação é realizada na primeira de uma lista contendo quatro máquinas distintas). Se disponível, o DQS seleciona dentre as estações aquela que possui a menor carga para executar cada tarefa. Assim, o Dmake usa as vantagens oferecidas pelas características de balanceamento de carga do DQS.

O DQS [Rev92, Duk94], desenvolvido e mantido pelo SCRI (*Supercomputer Computation Research Institute at Florida State University*), é uma ferramenta que gerencia de maneira transparente a distribuição das tarefas computacionais em um *cluster* de máquinas. Desta maneira ele aumenta a produtividade de todas as máquinas, e também o número de *jobs* que podem ser finalizados num dado período de tempo [Duk94].

O DQS controla quando e em quais máquinas os processos serão executados. Tais processos são colocados em uma ou mais filas (com diferentes prioridades ou tempos para execução dos *jobs*) e distribuídos entre as estações seguindo o critério de menor carga. A estação que recebe todas as requisições e controla a distribuição de processos é o servidor DQS, e as demais são os clientes DQS. Um *job* é enviado a um cliente DQS e submetido a uma de suas filas para execução.

Na interface com o DQS, o Dmake fornece as estações que colaboram no processo de *make* e o DQS escalona aquelas com menor carga para serem utilizadas primeiro.

3.6 Pmake

O Pmake (*Parallel Make*) [Int94] é um *make* paralelo construído para o sistema Paragon (supercomputador paralelo) [Int94a] e baseado no GNU *Make*. Em adição às características do GNU *Make*, o Pmake controla a execução paralela feita nas chamadas *partições* do sistema Paragon e fornece outras características para melhorar a compatibilidade com outros utilitários *make* e facilitar o controle sobre o processamento paralelo [Int94].

O sistema Paragon consiste de um supercomputador Intel com até 2000 *nós*. Um nó é essencialmente um computador com um ou mais microprocessadores Intel i860 XP e no mínimo 16 Mbytes de memória. Os nós são interconectados por uma rede de barramentos de alta velocidade.

Os nós usam o sistema operacional OSF/1 Paragon, que é uma versão estendida do OSF/1 da *Open Software Foundation* com melhoramentos para suporte ao processamento paralelo. O OSF/1 Paragon fornece a imagem de um sistema único aos nós. Por exemplo, todos eles compartilham um único sistema de arquivos, acesso idêntico a todos os dispositivos de I/O, e identificadores de processo (pid) únicos. O comando `kill` elimina o processo especificado em qualquer nó, não importando em qual esteja ativo. O OSF/1 Paragon fornece ainda o mecanismo de passagem de mensagens através de *system calls* e acesso paralelo ao sistema de arquivos.

A execução do Pmake no Paragon é feita numa de suas partições. Uma partição é um grupo de nós associados a um conjunto de parâmetros, os quais controlam algumas características de execução da aplicação [Int94]. Partições podem ser criadas, modificadas ou removidas por comandos ou *system calls*.

Existem dois tipos de partições no Paragon (e a execução paralela do Pmake pode ser efetuada em ambas): *service* e *compute*. A primeira é usada para processos interativos e programas não paralelos, e a segunda para a execução de programas de uso intensivo dos processadores e aplicações paralelas.

Na partição *service*, o Pmake usa a *system call fork()* e migração de processos para iniciar a execução simultânea dos comandos em nós desta partição, assegurando a execução paralela.

Ele conta ainda com o balanceamento de carga para decidir o nó para o qual o processo irá migrar.

Na partição *compute*, o Pmake age como uma aplicação paralela através da passagem de mensagens entre processos. O próprio Pmake executa na partição *service*, agindo como um processo controlador e enviando comandos em paralelo para os nós disponíveis na partição (ou sub-partições) *compute*.

Para execução em qualquer das duas partições, o Pmake permite ao usuário estabelecer o número máximo de *jobs* que podem ser executados em paralelo. Também pode ser especificado um valor real de carga média do sistema, sobre o qual o Pmake limita a execução dos *jobs*.

3.7 Outros Trabalhos

SGPC (Sistema Gerenciador de Processamento Cooperativo)

O SGPC [Car93] explora o "processamento cooperativo" ao distribuir módulos (compilações de arquivos, execução de cálculos, etc) de uma aplicação entre estações de trabalho da rede, possibilitando a utilização de seu tempo ocioso e capacidade de processamento.

O SGPC é formado por duas partes:

- Um servidor de processamento cooperativo (oferece serviços), e
- Uma biblioteca de rotinas para processamento cooperativo (oferece operações).

O servidor de processamento (ServProc), um *daemon*, espera e atende as requisições de serviços nas estações da rede. A aplicação que faz uso do "processamento cooperativo" inclui a biblioteca de rotinas do SGPC. Estas rotinas fazem as solicitações de serviços aos ServProcs seguindo o modelo cliente/servidor implementado usando o mecanismo de RPCs.

A figura 3.4, extraída de [Car93], ilustra as interações entre os diferentes componentes do SGPC em operação. Elas incluem interações entre ServProcs e entre aplicações e ServProcs.

Cada aplicação numa estação tem a sua disposição um Servidor de Processamento (que poderá usar diretamente), e um conjunto de rotinas em uma biblioteca, através das quais tem acesso indireto aos Servidores de Processamento cooperativo.

O sistema SGPC faz uso intensivo de *system calls*, funções de bibliotecas, RPC (*Remote Procedure Call*), dos serviços de rede NIS (*Network Information Service*) e NFS (*Network File System*) [Sun87]. O serviço NIS permite o acesso às informações da configuração da rede e das estações interligadas, e o serviço NFS permite o compartilhamento do sistema de arquivos entre as estações [Car93].

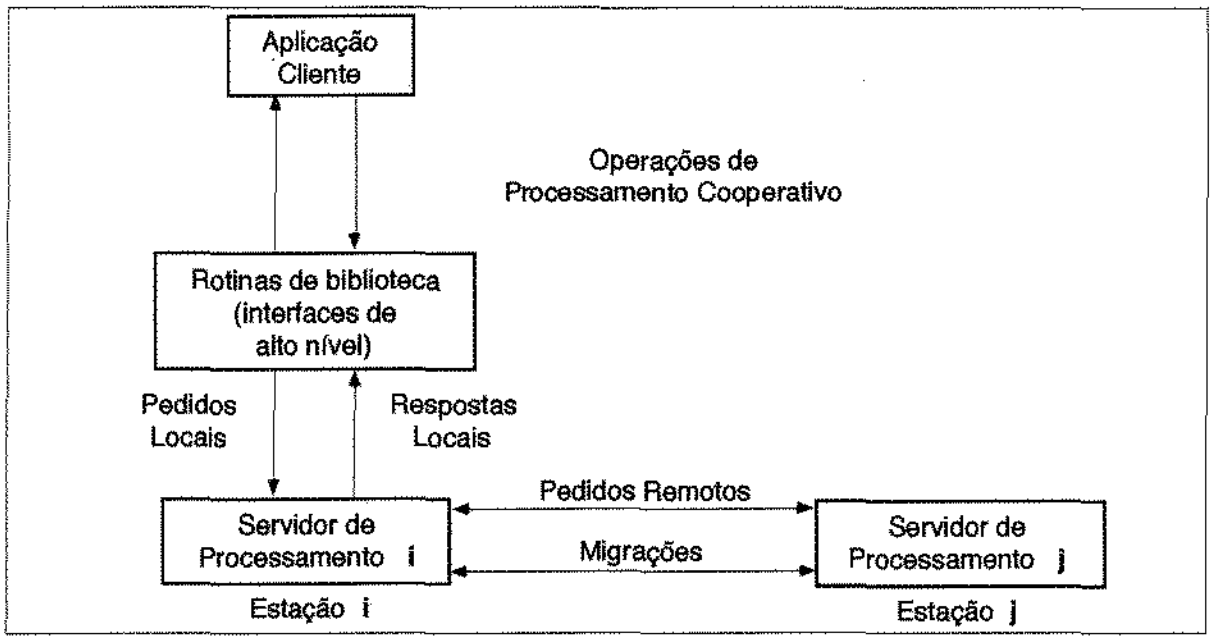


Figura 3.4: Interação entre Servidores de Processamento e Aplicações

[Car93] implementou dois exemplos de aplicação usando o SGPC. Num deles, trabalhou-se a partir do código fonte de um programa *make* bastante simples. Foram introduzidas modificações para que os pedidos de execução dos comandos fossem encaminhados através do SGPC. Assim, este *make* distribuído encaminha as compilações para diferentes estações, conseguindo que sejam realizadas em paralelo.

Periodicamente, cada ServProc local coleta informações sobre a porcentagem de tempo ocioso (*idle*) de sua CPU no último intervalo de tempo. Estas informações são passadas às demais estações. Assim, a seleção da estação que executará uma tarefa é feita baseando-se no nível de ociosidade.

No exemplo de *make* distribuído, ao invés dos comandos resultarem na execução de programas submetidos diretamente ao sistema operacional, são geradas chamadas ao SGPC para execução concorrente do mesmo programa.

O *make* utilizado no desenvolvimento do exemplo foi o *gymake*, desenvolvido por Greg Yachuk [Yac88] e disponibilizado em domínio público. Este *make* original foi modificado para suportar a execução concorrente dos comandos de uma regra, pois o *gymake* não suporta execução concorrente de regras como o GNU *Make*.

[Car93] deixa claro que as modificações feitas no *gymake* objetivam apenas sua utilização como um exemplo de distribuição das execuções para diferentes estações em paralelo. Uma versão definitiva de um *make* distribuído precisaria introduzir modificações na sintaxe do arquivo `makefile` para possibilitar a indicação de quais comandos podem ou não ser executados paralelamente.

Make Paralelo

[Fle89] descreve um protótipo de uma aplicação *make* paralela, que é um método de execução de compilações em paralelo em sistemas UNIX fracamente acoplados; os módulos objeto resultantes são ligados após todas as compilações terminarem. A aplicação foi construída usando o mecanismo de passagem de mensagens do Linda [Gei94], que é similar ao PVM. Todas as estações compartilham o mesmo sistema de arquivos distribuído, NFS.

Para realizar as compilações em paralelo são usados dados de um arquivo `makefile` como o exemplo abaixo, extraído de [Fle89]. O formato do arquivo é simples e contém somente as quatro variáveis definidas no exemplo. A divisão das compilações entre as máquinas da rede é feita estaticamente (de acordo com o tamanho dos arquivos em *bytes* e o número de estações disponíveis para uso), ou seja, quando as compilações se iniciam, já são conhecidas quais máquinas compilarão quais arquivos fonte. Cada estação recebe uma tarefa de compilação por vez, através de uma operação Linda. A próxima tarefa é recebida após a devolução dos resultados da anterior.

```
CFLAGS= -c -g ;
SOURCES= icons.c cmain.c draw.c menu.c locator.c keyboard.c file.c init.c\
         initbrushes.c initpatternes.c initcommands.c tablet.c\
         activebrusches.c globvar.c ;
LIBS= -lcore -lsunwindow -lpixrect -lm ;
EXECNAME= gwpaint ;
```

Figura 3.5: Formato e exemplo do `makefile` para o *make* paralelo

O propósito do protótipo era experimentar e demonstrar a utilidade dos recursos disponíveis na computação distribuída; a dependência entre os arquivos não é levada em consideração, ou seja, os arquivos a serem compilados são considerados um conjunto linear como descrito no exemplo. Portanto, a utilização prática de tal protótipo fica comprometida pelo fato de ser necessário considerar a relação de dependências num pacote de *software* a ser compilado. Os resultados também não foram satisfatórios; para melhorá-los, os próprios autores sugerem possibilidades mais eficientes de organização do protótipo. Dentre elas, pode-se destacar a mudança da repartição das compilações considerando a ociosidade das estações.

Capítulo 4

ESTRUTURAS INTERNAS DO GNU *MAKE*

O *make* é a base deste trabalho de mestrado. Para estendê-lo para um ambiente distribuído foi necessário o entendimento do código fonte, alteração de algoritmos e estruturas de dados do GNU *Make* e o projeto de um servidor multi-tarefa para o processamento de pedidos de clientes.

Este capítulo descreve as principais estruturas de dados e algoritmos utilizados pelo utilitário GNU *Make* para reprocessar apenas as etapas realmente necessárias num projeto e mantê-lo atualizado. São apresentadas ainda algumas considerações sobre sua implementação.

4.1 Estruturas internas

O *make* tornou-se uma ferramenta poderosa, inteligente e de grande utilidade tanto no ambiente de programação quanto na instalação e atualização de *softwares* pelos programadores e administradores de sistemas, havendo implementações e versões disponíveis para várias plataformas e sistemas operacionais.

O GNU *Make* é um dos programas *make* mais completos e flexíveis em termos de características e funcionalidades, as quais foram sendo incorporadas a cada nova versão. Para oferecer suas facilidades, armazena na memória estruturas de dados com definições de variáveis de ambiente (sessão Unix em uso), argumentos e opções da linha de comando e as entradas do `makefile`. As variáveis do ambiente incluem opções aceitáveis pelo *make*.

As principais estruturas internas[1] são as seguintes:

[1] As estruturas de dados referentes a estas estruturas internas são apresentadas no Apêndice A.

- Tabela de opções de comando
- Tabela de *Hashing* de Arquivos – THA
- Conjunto de variáveis por arquivo
- Conjunto de váriáveis gerais
- Lista de Processos em Execução – LPE

Conhecer e entender essas cinco estruturas e as relações existente entre elas é essencial para compreender o funcionamento interno do GNU *Make*, que pode ser dividido logicamente em quatro grandes fases dependentes e cooperantes:

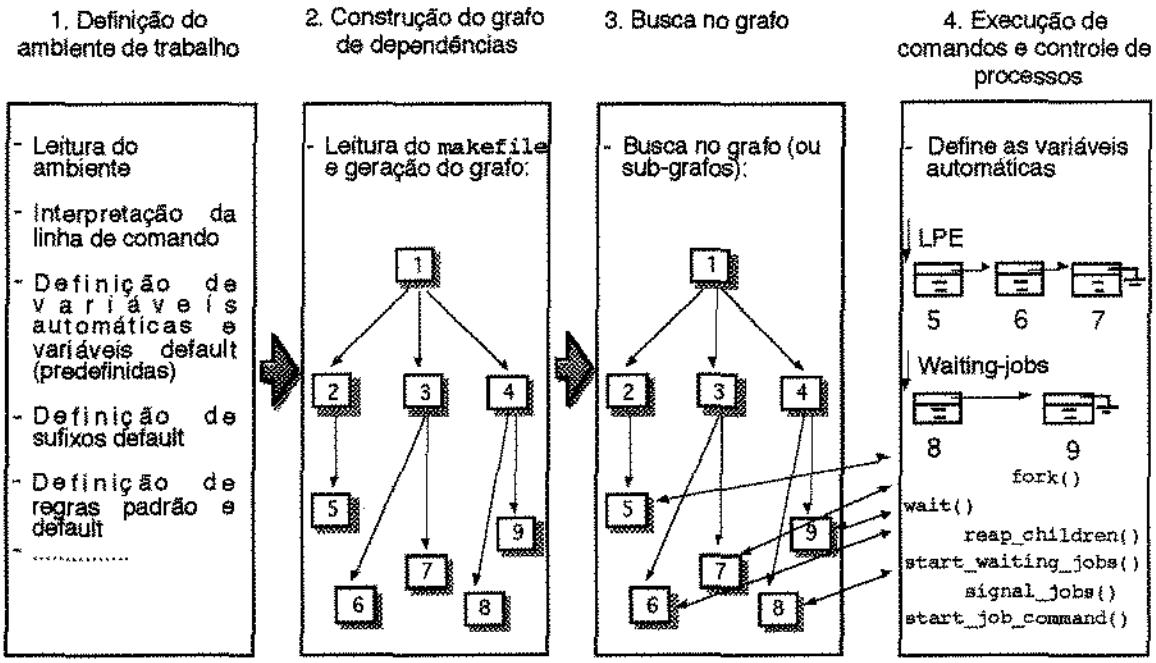


Figura 4.1: Fases do processo de *make*.

As fases 3 e 4 são bastante integradas. Quando a busca escalona um nó para processamento, a execução prepara os comandos e variáveis locais ao nó e decide se inicia ou não o seu processamento, ao término do qual a própria fase de execução efetua a atualização do grafo.

As tarefas realizadas por cada uma dessas fases serão comentadas ao longo deste capítulo, assim como a manipulação das estruturas mencionadas acima. A seguir, é descrita cada uma das cinco estruturas.

4.1.1 Tabela de opções de comando

Tabela que possui as possíveis opções de linha de comando aceitas pelo *make*. Existem cerca de 30 entradas, com informações como tipo, valor *default*, descrição e apontador para a

variável global correspondente à opção que habilita ou não o uso de uma funcionalidade existente. Parte desta tabela é inicializada estaticamente, e sua função principal é auxiliar na decodificação da linha de comando e variáveis de ambiente, além de fornecer mensagens em caso de erro.

O uso desta tabela proporciona uma solução flexível em caso de mudanças ou acréscimo de novas funcionalidades, além de eficiência quanto ao controle da variedade de opções oferecidas pelo *make*.

4.1.2 Tabela de *Hashing* de Arquivos

A Tabela de *Hashing* de Arquivos (THA) é a maior e mais importante estrutura interna do *make*, onde são acumuladas todas as informações de qualquer arquivo (*target* ou não). É necessário enfatizar que uma ação presente numa entrada do `makefile` é também vista como um *target*. Ainda, nesta tabela é construído o grafo de dependências, através do encadeamento de apontadores entre os elementos (estruturas `file`). Cada nó do grafo corresponde a um elemento da tabela.

Cada entrada da THA é uma lista de estruturas `file` alocada dinamicamente. A aplicação da função de *hashing* dá como resultado o endereço de inserção de um elemento na THA. A lista é a solução usada para a resolução de colisões, ou seja, os elementos de uma lista são aqueles que coincidem no resultado da aplicação da função de *hashing*.

A figura 4.2 mostra um esquema simples e geral da THA após a inserção de alguns *files*.

A função de *hashing* utilizada é bastante simples, tendo como chave o nome do arquivo.

```
int Hash(char *filename)
{
    int hashval = 0;
    char *ch = filename;
    while ( *ch )
        {
            hashval += *ch++;
            hashval  =  (hashval << 7) + (hashval >> 20);
        }
    return (hashval % FILE_BUCKETS);        /* FILE_BUCKETS = 1007*/
}
```

A função `Hash()` acima mostra que, para cada caractere do nome do arquivo (chave para acesso à tabela), adiciona-o a um acumulador e, soma os deslocamentos de 7 *bits* para a esquerda com 20 *bits* para a direita. O resultado final é dividido pelo tamanho da tabela, e o endereço resultante é o resto da divisão, que é o índice na THA.

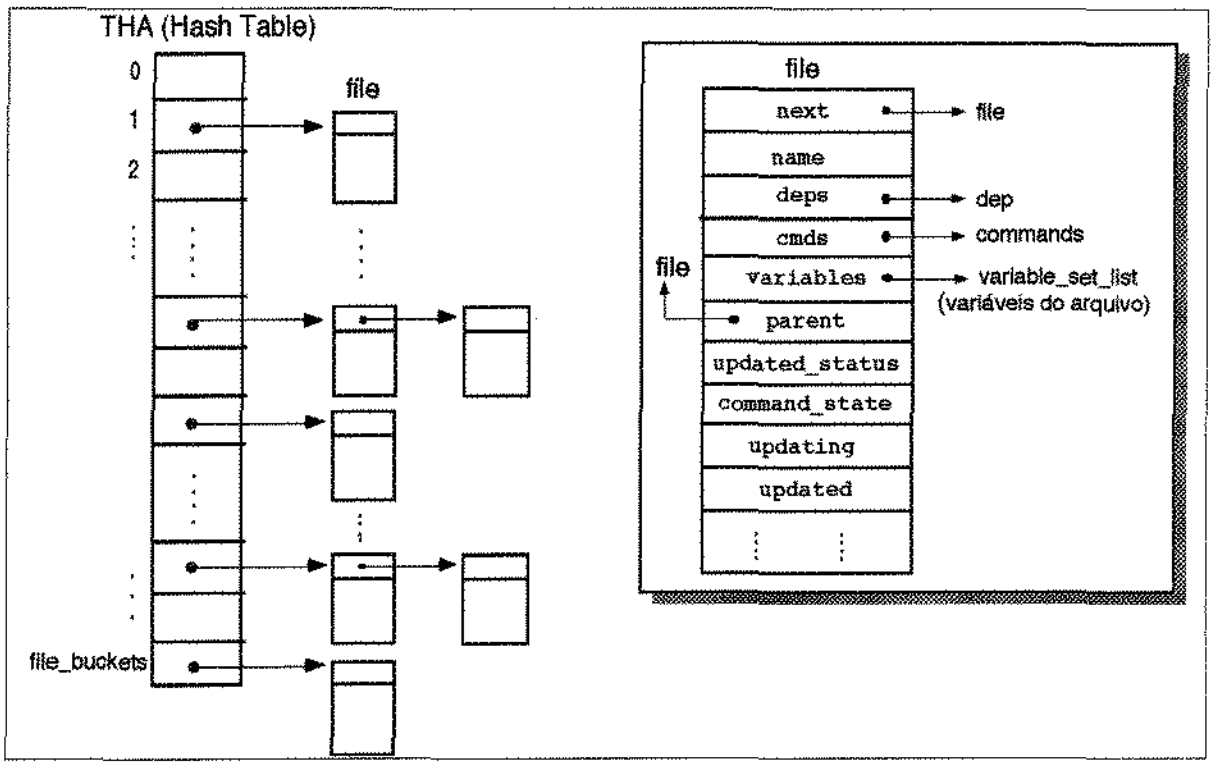


Figura 4.2: Esquema da tabela de *hashing* THA.

Cada elemento da THA possui uma série de campos incluindo:

- `name`: nome do arquivo;
- `deps`: lista das dependências de `file`. Cada elemento desta lista referencia um nó filho (dependente) no grafo de dependências;
- `cmds`: apontador para os comandos da regra presente no `makefile`;
- `variables`: apontador para o conjunto de variáveis do arquivo (veja Seção 4.1.3);
- `parent`: apontador para o nó pai (`file` que depende do nó) no grafo;
- `updated_status`: *status* da última tentativa de atualização do arquivo;
- `command_state`: *status* da execução dos comandos: não iniciada, comandos executando, filhos executando ou terminado;
- `updating`: indica se nós filhos estão executando;
- `updated`: indica se o arquivo já foi ou não atualizado.

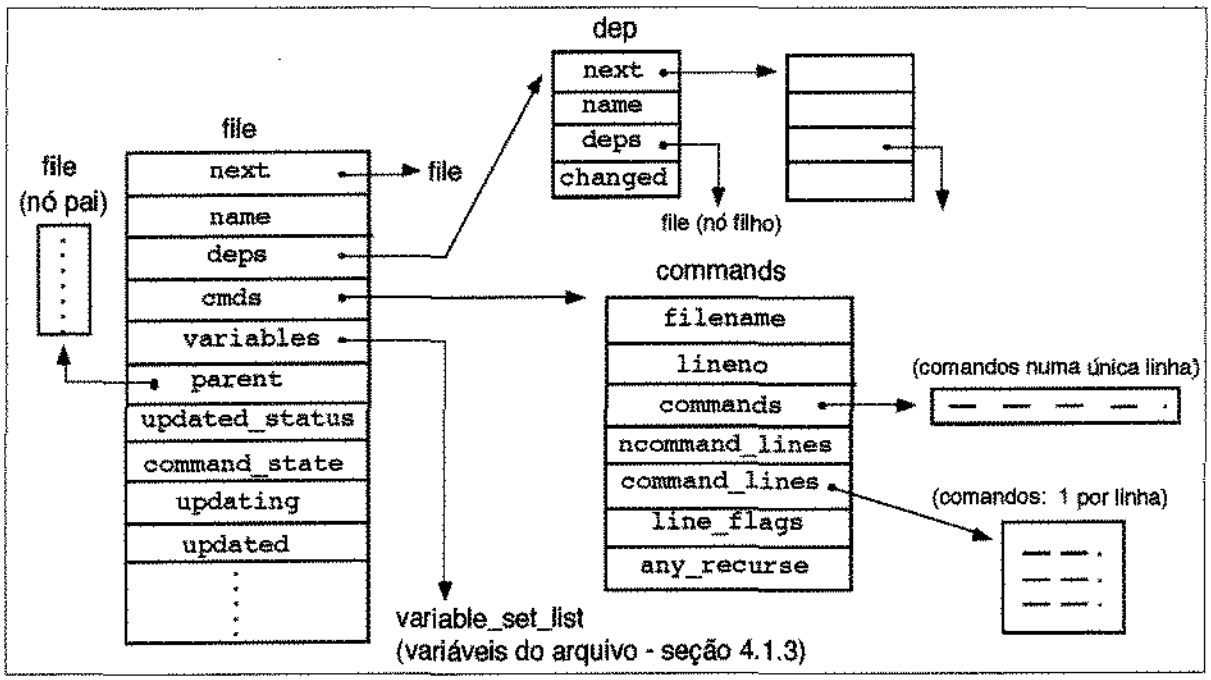


Figura 4.3: Estrutura de um elemento da THA.

4.1.3 Conjunto de variáveis de arquivo

Cada *target* possui seu conjunto de variáveis, chamadas variáveis automáticas de arquivo. Essas variáveis são visíveis somente dentro da regra que gera um *target* e podem ter como conteúdo o nome do *target*, nome da primeira dependência, nomes de todas as dependências, nomes das dependências mais recentes que o *target*, etc.

São armazenadas numa tabela (`table`) com mesma função de *hashing* da THA e referenciada através do campo `variables` da estrutura `file` presente na THA, esquematizado na figura 4.4

O conteúdo destas variáveis é baseado no *target* e em suas dependências, sendo computado no momento em que este é escalonado para ser gerado.

Variáveis automáticas de arquivo são muito usadas, especialmente em definições de regras implícitas. Por exemplo, quando se escreve uma regra implícita que diz como compilar qualquer arquivo do tipo `x.c` para obter um objeto do tipo `x.o`, o nome do arquivo muda a cada aplicação da regra:

```
%.c : %.o
        $(CC) -c $@ -o dirobjetos/$<
        echo Compilado o arquivo: $@
```

A cada aplicação da regra, as variáveis `$@` e `$<` são substituídas pelos arquivos `.c` e `.o`, respectivamente.

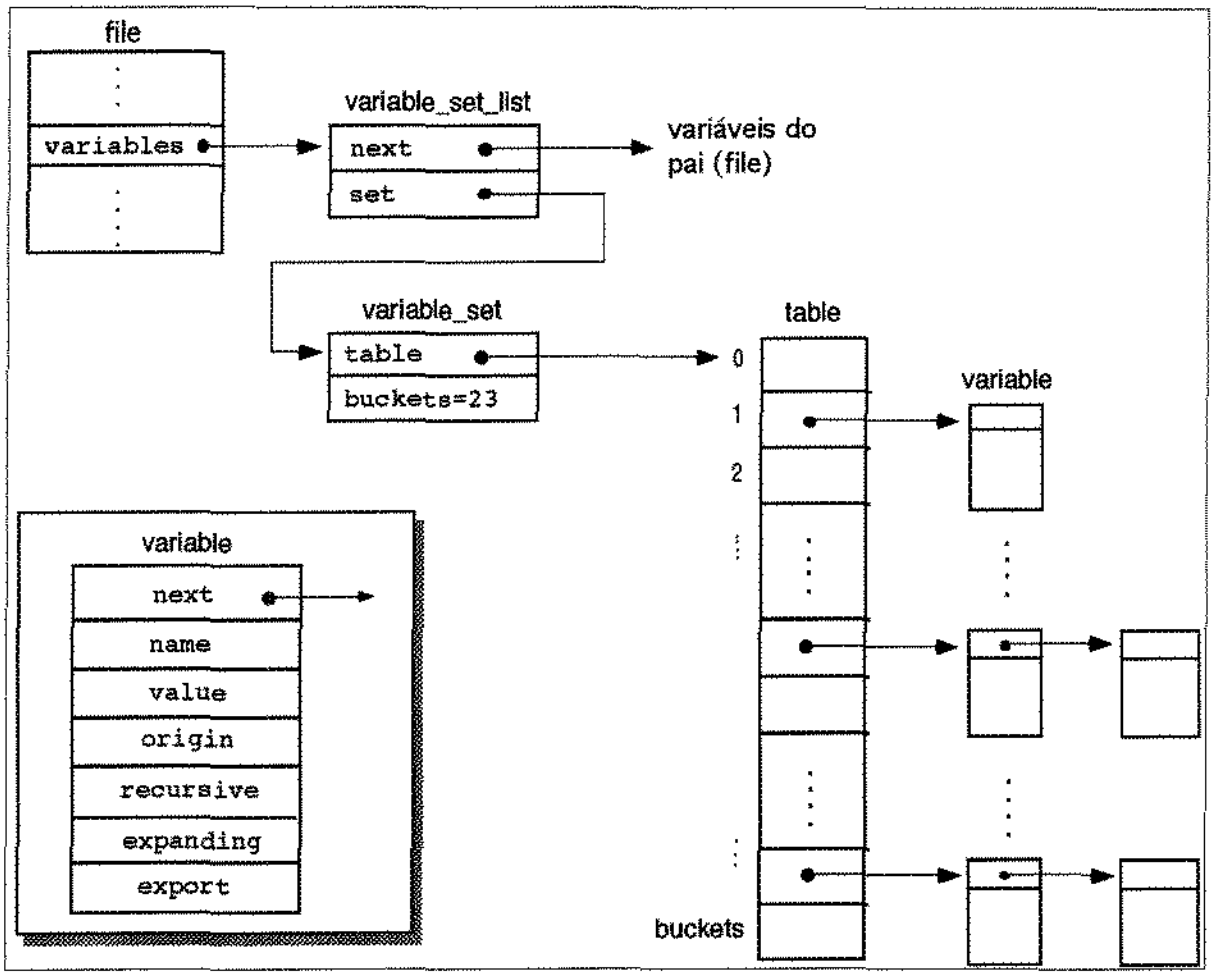


Figura 4.4: Esquema lógico do conjunto de variáveis de arquivo.

4.1.4 Conjunto de variáveis gerais

Além das variáveis automáticas definidas em cada *target*, o *make* mantém um grande conjunto de variáveis gerais composto pelas variáveis de ambiente (passadas pela *shell*), de `makefiles` (definidas nos `makefiles`), da linha de comando (definidas na linha de comando que invoca o *make*), nomes de programas muito usados e nomes de argumentos (ambos predefinidos internamente). São usadas principalmente para auxiliar e adicionar considerável flexibilidade na construção de regras e na execução de comandos.

A estrutura empregada para armazená-las (figura 4.5) segue o mesmo estilo das variáveis de arquivo, porém com um número muito maior de entradas.

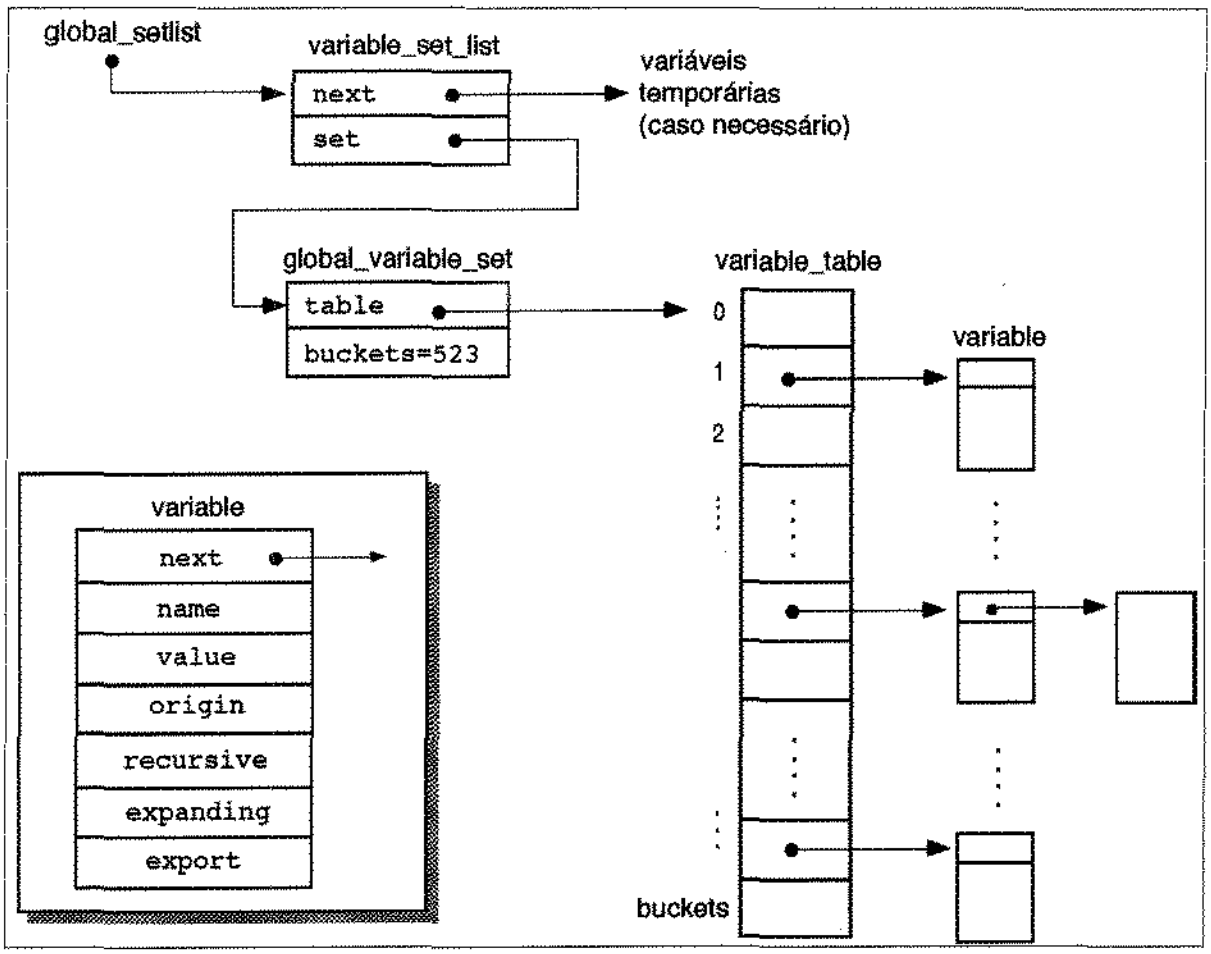


Figura 4.5: Estrutura do conjunto de variáveis gerais.

As variáveis gerais são classificadas de acordo com sua origem (local de definição). As principais "classes" (identificadas no campo `origin` de `variable`) são:

- `default`: variáveis predefinidas internamente, muito usadas em regras implícitas. São nomes de programas (como CC, TEX, YACC) e nomes de argumentos (como CFLAGS, LDFLAGS);
- `environment`: variáveis do ambiente;
- `file`: variáveis definidas em `makefiles`;
- `env_override`: variáveis definidas como variáveis de ambiente e com prioridade sobre as variáveis de `makefile` (`file`);
- `command_line`: variáveis definidas na linha de comando;
- `override`: variáveis superpostas pela diretiva `override` no `makefile`;

- `automatic`: variáveis que são variações das automáticas da seção anterior. No GNU *Make*, são consideradas semi-obsoletas, já que há funções que realizam efeito similar [Sta95].

Mesmo que uma variável já esteja definida, o *make* pode sobrerpor outro valor, se sua classe permitir. A prioridade para superposição de variáveis é crescente de acordo com as classes relacionadas. Por exemplo, se uma variável for definida na linha de comando (`command`) e num `makefile` (`file`), a primeira definição prevalece desde que a segunda não use a diretiva `override`. Existe também uma opção no *make* que inverte as prioridades das classes `environment` e `file`, formando a classe `env_override`.

As variáveis gerais definidas pelo programador simplificam a construção e manutenção do `makefile` e reduzem a possibilidade de inconsistências.

4.1.5 Lista de Processos em Execução (LPE)

Quando um *target* é escalonado para geração, após a expansão das variáveis dos comandos envolvidos, estes são inseridos na Lista de Processos em Execução (LPE) concorrentes. A lista é atualizada na execução (seqüencial) de cada comando de um *target*; após a conclusão do último comando, o nó (`file` na THA) é atualizado, e a entrada na LPE é eliminada. Assim, num dado instante, a relação entre o número de comandos sendo executados e *targets* sendo processados é um para um.

A estrutura que descreve um elemento na LPE (figura 4.6) tem como principais entradas:

- `file`: apontador para a estrutura `file` na THA. Identifica o nó sendo processado;
- `environment`: variáveis de ambiente para execução dos comandos. São extraídas do conjunto de variáveis gerais;
- `command_lines`: conjunto de comandos após a expansão das variáveis;
- `command_line`: Identificador numérico do comando sendo executado;
- `command_ptr`: apontador para a linha de comando em execução;
- `pid`: identificador do processo em execução.

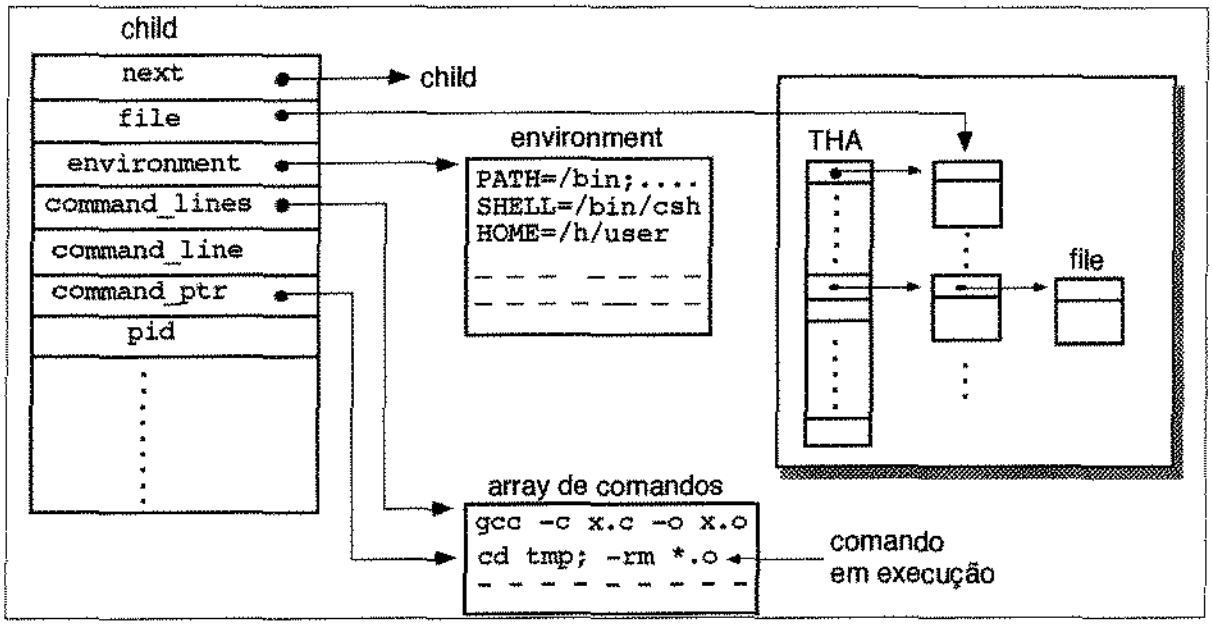


Figura 4.6: Representação de um elemento da LPE.

4.2 Grafo de dependências

A partir das regras existentes no `makefile`, o *make* constrói em memória o grafo de dependências. O grafo modela a relação existente entre os *targets* e suas dependências. Cada nó corresponde a um *target*, com comandos, variáveis e lista de nós filhos, ocupando o espaço de um elemento (estrutura `file`) na THA.

A montagem do grafo consiste em duas etapas. Na primeira, realizada durante a leitura do `makefile`, cada regra dá origem a um ou mais elementos na THA (figura 4.2), cada um correspondendo a um nó que vem acompanhado de suas dependências diretas, variáveis e comandos para construir o *target* (figura 4.3). O produto desta fase é um conjunto de nós sem arestas (apontadores). A segunda fase consiste na criação das arestas interligando os nós para formar uma estrutura hierárquica. Para cada nó que possui indicação de dependentes, cria-se uma aresta que conecta o indicador do dependente a sua respectiva localização física na THA. O produto final é o grafo de dependências hierárquico direcionado (figura 4.7), que no caso geral é desconexo, ou seja, formado por um conjunto de componentes conectados, onde cada grafo corresponde a uma ação presente no `makefile`.

A seguir é apresentado um exemplo simples do grafo de dependências já montado. O exemplo é composto do arquivo `makefile` (com poucas regras, e estas com poucas dependências), do grafo lógico e sua representação física na THA.

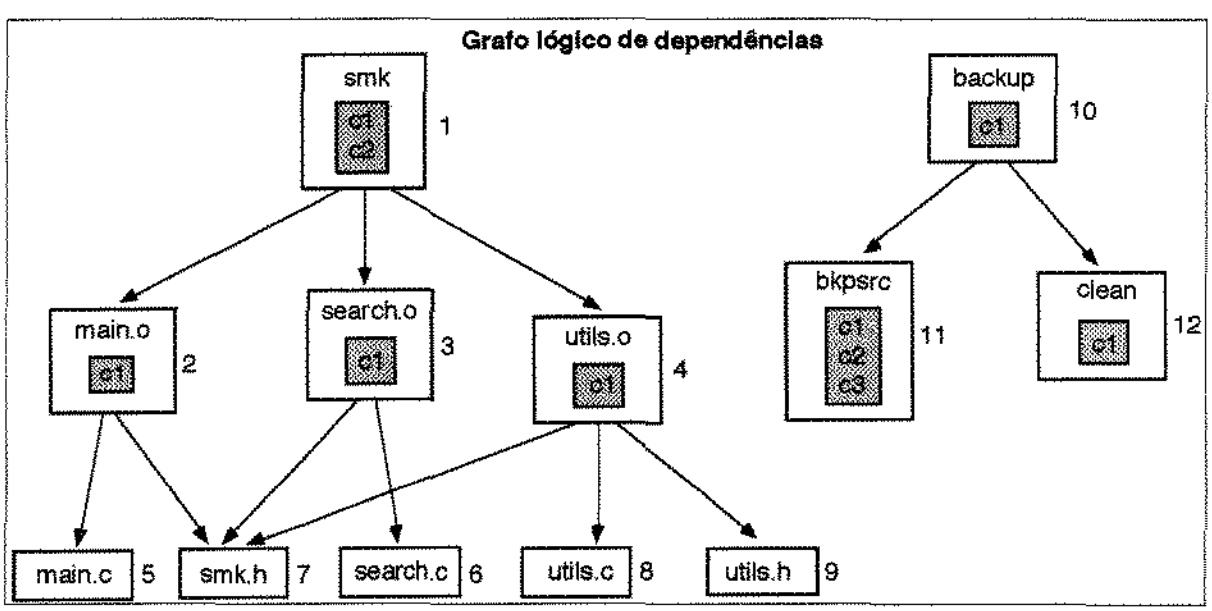
```
#
# Makefile para gerar smk
#
CC = gcc
CFLAGS = -g
LDLIBS = -L/usr/local/lib -lkvm -lelf
OBJS = main.o search.o utils.o

smk : $(OBJS)
      $(CC) $(CFLAGS) $(OBJS) $(LDLIBS) -o smk
      @echo Gerado smk ; date

#Regra de sufixo
.c.o :
      $(CC) -c $(CFLAGS) $<

main.o : main.c smk.h
search.o : search.c smk.h
utils.o : utils.c utils.h smk.h

#Backup do sistema, com remoção dos arquivos objetos
backup : bkpsrc clean
      @echo Backup Terminado.
bkpsrc :
      @echo Compactando os arquivos fontes...
      tar -cvf smk.tar *.[ch]
      mv -f smk.tar bkpsmk/smk.tar
clean :
      -rm -f $(OBJS) smk core
```



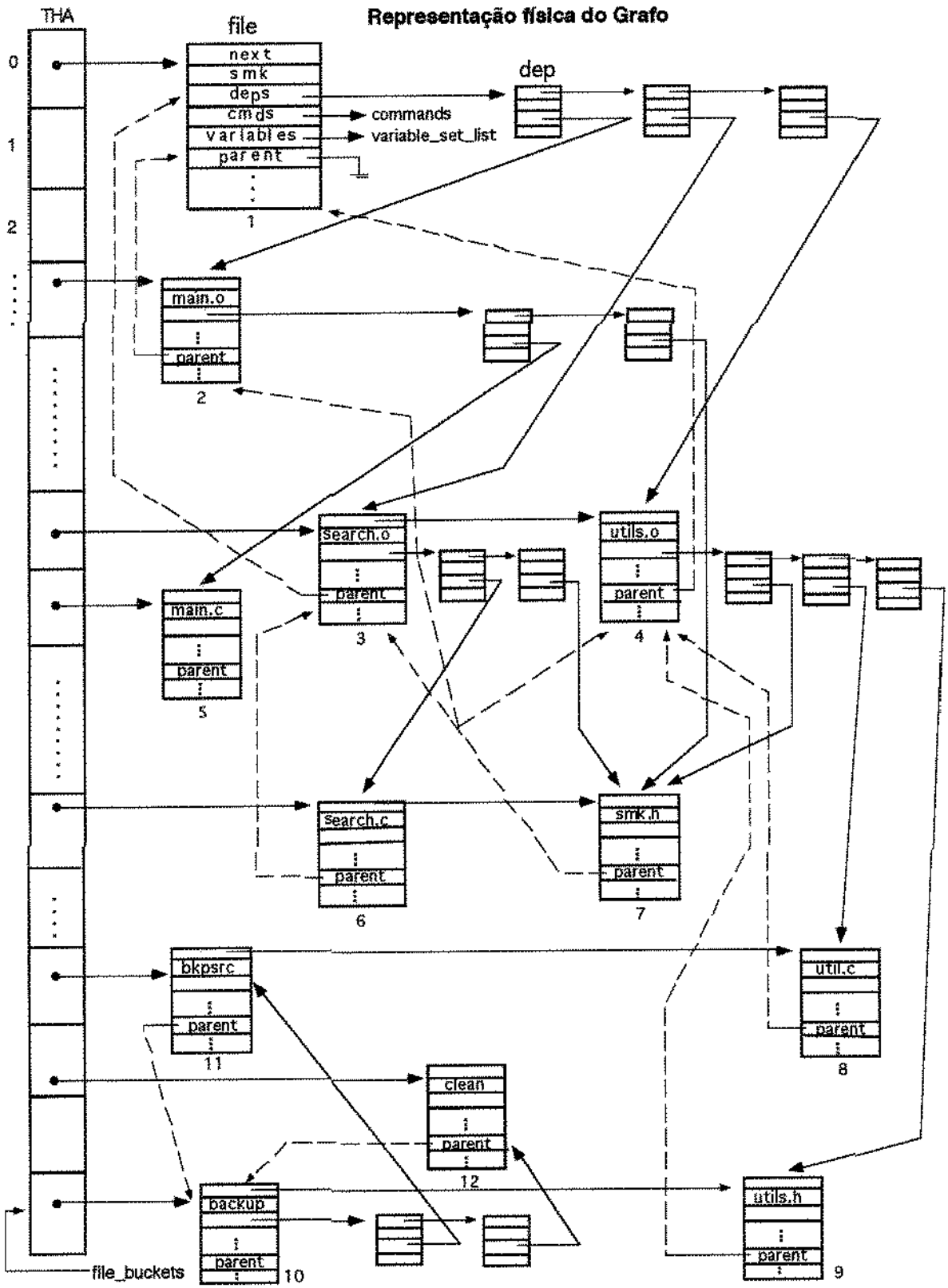


Figura 4.7: Makefile e grafos lógico e físico de um exemplo.

Na figura 4.7 são apresentados somente os campos básicos da estrutura `file` para formar o grafo de dependências na THA. Como vários *targets* podem depender de um mesmo arquivo (por exemplo, `smk.h` na figura 4.7), o campo `parent` de cada nó é atualizado de acordo com a busca no grafo.

4.3 Algoritmo de busca no grafo

A busca no grafo é a etapa mais complexa do processo de *make*. O algoritmo que realiza tal tarefa é recursivo e extremamente extenso, oferecendo o recurso de ativação de diversos nós para processamento simultâneo. Entretanto, um nó pode ser processado somente quando seus filhos estiverem atualizados. A execução de comandos de um nó é seqüencial, ou seja, num dado instante, o número de comandos sendo executados é igual ao número de nós sendo processados.

Uma grande vantagem do algoritmo é que nem sempre é necessário realizar a busca em todo o grafo. Seu início se dá a partir dos *targets* (denominados *goals*) especificados como argumentos do *make*. Neste caso são percorridos e atualizados somente os nós dos sub-grafos com origem nos *goals*. Se nenhum *goal* é especificado como argumento, a busca se inicia no primeiro *target* (ou numa ação) do `makefile`, que no exemplo abaixo é T1 *(default_goal)*. No mesmo exemplo, se T2, T7 e T8 fossem especificados como argumentos, a busca se restringiria aos sub-grafos indicados pelos elementos correspondentes da lista *goals*.

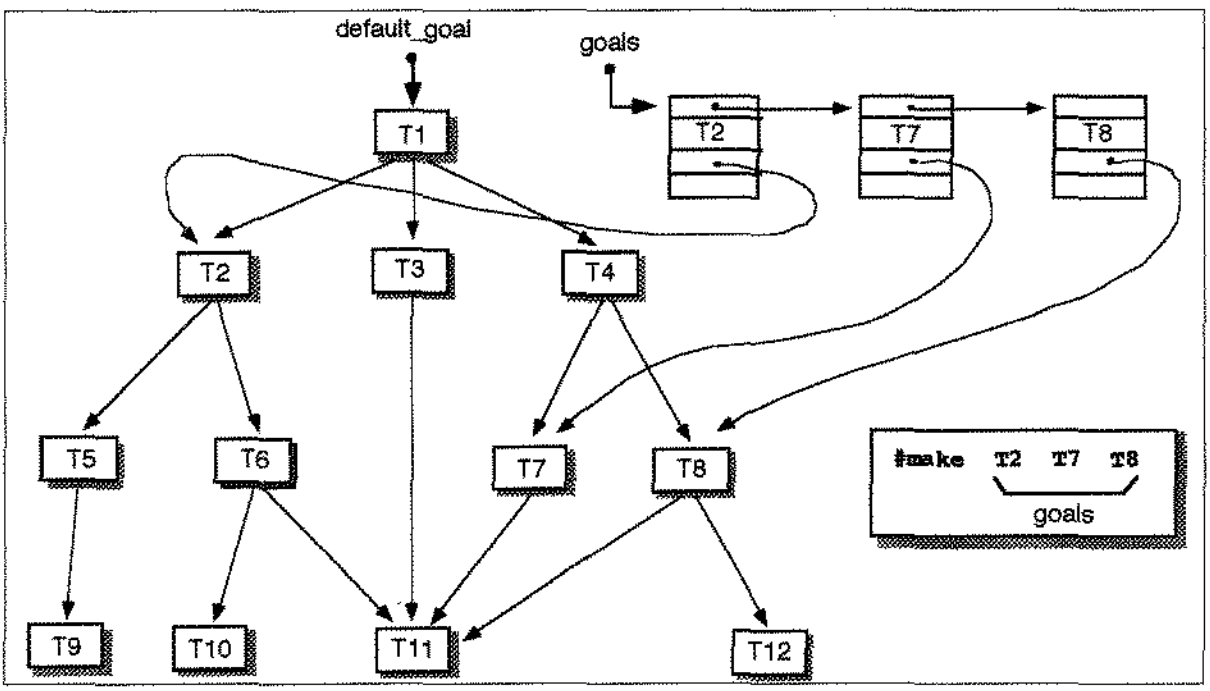


Figura 4.8: Exemplo de buscas em sub-grafos.

Usando a funcionalidade de execução concorrente, geralmente não é possível construir todos os *targets* com apenas uma varredura (em profundidade) em cada sub-grafo determinado pelos argumentos. A estratégia adotada é percorrer a lista *goals* (veja exemplo acima) várias vezes (cada *goal* determina a raiz de um sub-grafo) e, em cada iteração, colocar o número máximo de nós possíveis dos sub-grafos em processamento. Quando um sub-grafo é totalmente atualizado, o *goal* correspondente da lista é eliminado.

O procedimento `Update_Goal()` apresentado na figura 4.9, descreve de maneira resumida a implementação da estratégia de percorrer os sub-grafos várias vezes, até atingir o objetivo final. O procedimento `Search()` (figura 4.10) realiza a busca em profundidade em cada sub-grafo e ativa a *flag* `updated` da raiz de um sub-grafo quando este estiver atualizado, o que provoca a remoção do respectivo *goal* da lista.

```
Update_Goal(struct dep *goals)
{
   while (goals != 0)
      {
         struct dep *g, *lastgoal;
         start_waiting_jobs();       /* Inicia jobs que estão esperando devido a carga do sistema */
         reap_children(1,0); /* Espera p/ término de um nó e dispara comandos intermediários. */
         lastgoal = 0;
         g = goals;
         while (g != 0)
            {
               struct file *file = g->file;
               Search(file);          /* realiza a busca a partir de file */
               if (file->updated)
                  {            /* sub-grafo atualizado e */
                     if (lastgoal == 0)     /* é o primeiro da lista? */
                        goals = g->next;
                     else lastgoal->next = g->next;
                     free ((char *) g);     /* libera espaço de g */
                     if (lastgoal == 0)
                        g = goals;
                     else g = lastgoal->next;
                  }
               else {
                     lastgoal = g;
                     g = g->next;  /* próximo sub-grafo a percorrer */
                    }
            }
      }
}
```

Figura 4.9: Implementação de várias buscas em sub-grafos.

O procedimento `Update_Goal()` mostra que antes de iniciar cada iteração os nós em espera, devido a alta carga média do sistema, são colocados em processamento (`start_waiting_jobs()`) e seus comandos inseridos na LPE. Depois, espera-se até que o processamento de um nó termine, se houver algum em andamento. Durante a espera (realizada por `reap_children()`), para cada nó em processamento cujo processo em

execução termine, o próximo comando do respectivo nó (presente na LPE) é colocado para executar. Para cada processo finalizado, faz-se a verificação de erros e o conseqüente tratamento, que pode implicar até na parada do *make*.

Mesmo que o número de nós sendo processados seja menor que o máximo especificado, ocorre a espera pelo término de um deles antes do início de uma nova varredura na lista *goals*. Isto poderia ser melhorado através da ativação de espera somente quando a quantidade de nós em processamento atingir o valor máximo.

A busca em profundidade é ilustrada pelo procedimento `Search()` (figura 4.10), escrito como um pseudo-código resumido para torná-lo mais simples e facilitar o entendimento. São apresentadas somente as principais *flags* para controle de acesso e execução de nós, e nenhum tratamento de erros.

```
Search(node)
{
    atualize_node = false
    Se (node->updated OU node->command_state == cs_running OU node->command_state ==
        cs_finished OU node->command_state == cs_deps_running))
       return
    Se (target(node) não existe)
       atualize_node = true
    Para cada filho(node)
       Search(filho(node))
       Se (filho(node)->command_state == cs_running OU
           (filho(node))->command_state == cs_deps_running)
           node->command_state = cs_deps_running          /* dependências executando. */
           return
       Se (filho(node) mais recente que node)
           atualize_node = true
    Se (atualize_node == false)
       node->updated = 1
       return
    Update_No(node)        /* tenta colocar o node em processamento */
    Se (node->command_state != cs_finished)
       return              /* processamento do node finalizado */
    node->updated = 1      /* node atualizado. */
}
```

Figura 4.10: Pseudo-código da busca no grafo.

Através de uma análise do algoritmo, observa-se que:

- A busca desce pelo sub-grafo até atingir um nó que está atualizado ou em execução, sendo que na primeira iteração todos estão desatualizados;
- Após visitar os filhos de um nó, este não é processado se algum filho ainda não estiver pronto. Neste caso, o processamento do nó só poderá ser feito na próxima iteração (busca);
- Um *target* é construído se não existe ou se algum filho for mais recente;

O processamento do nó é iniciado em `Update_No()`. Porém, se o número de nós ativos for igual ao valor máximo permitido, o processamento iniciará somente após uma espera idêntica àquela implementada no início de cada varredura seqüencial na lista *goals* (uso de `reap_children()`). Caso a carga média do sistema esteja alta, independente da quantidade de nós ativos, o nó é inserido na lista de espera. Em ambos os casos faz-se: `node->command_state = cs_running`[2].

4.4 Execução de Comandos

Colocar um nó em processamento significa inserir seus comandos na LPE, iniciar a execução do primeiro deles, e mais tarde a dos demais. A cada término de um processo (comando em execução) é verificado seu estado; eventuais erros são tratados, podendo interromper a execução do *make*.

A verificação do término de um processo (incluindo o tratamento de erros e disparo do próximo) é feita em dois pontos durante a busca no grafo, antes de iniciar uma nova busca na lista *goals* (figura 4.9) e antes de iniciar o processamento do nó escalonado, em `Update_No()`. O mesmo se aplica aos nós que estão na lista de espera, ou seja, tenta-se colocá-los em processamento nestes dois pontos (com `start_waiting_jobs()`).

Para cada nó a ser processado, é formado um ambiente (*environment*) usado na execução de todos os seus comandos, e armazenado na LPE (figura 4.6). As variáveis (cadeias da forma "nome = valor") que o formam são extraídas do conjunto de variáveis gerais, obedecendo a regras que dependem de fatores como origem da variável e prioridades das classes; por exemplo, toda variável de ambiente presente na sessão de chamada do *make* faz parte do ambiente de execução de seus processos filhos, mesmo com o valor alterado.

Para executar um comando, o *make* cria um processo filho usando a *system call vfork()* seguida de *exec()*. Com otimização, se o comando não possui metacaracteres como ponto-e-vírgula (`;`), símbolos de redireção (< , > , >> , |), símbolos de substituição (* , ? , $, =), escapes ou comentários (" , ` , \ , # , etc), sua execução é feita diretamente pelo processo filho, sem necessidade de uso da *shell*, evitando assim a criação de um processo. Em caso contrário, o processo filho se transforma numa *shell* e esta efetivamente invoca o comando.

A relação entre os comandos em execução (num dado instante), nós na lista de espera e a quantidade máxima permitida de *jobs* em execução concorrente, é dada pela inequação:

```
E + W ≤ J

onde:  E = comandos em execução (ou nós em processamento);
       W = nós na lista de espera devido a carga do sistema;
       J = número máximo de jobs em execução.
```

[2] O campo `command` contém o estado de execução dos comandos: `cs_not_started` (não iniciado), `cs_deps_running` (nós filhos executando), `cs_running` (executando) e `cs_finished` (terminado).

4.5 Considerações de implementação

O GNU *Make* foi implementado conforme o padrão POSIX.2 (IEEE 1003.2-1992), o qual especifica o *make*. Diferentes implementações de *make* podem incluir extensões que não são previstas no POSIX.2, e na maioria dos casos dificultam a construção de `makefiles` portáveis [Sta95].

O código fonte usado (cerca de 20.000 linhas em linguagem C) pertence à versão 3.74, a mais recente disponível no início do trabalho e liberada para alterações sob condições da GNU *General Public License*, publicada pela *Free Software Foundation*. Uma inovação presente na última versão (GNU *Make* 3.75), disponibilizada recentemente, é a portabilidade do GNU *Make* para as plataformas Windows NT, Open VMS e Amiga DOS, o que tornou o código relativo a controle e manipulação de processos mais obscuro devido às diferenças entre as plataformas. A versão usada é portada para quase todos os sistemas Unix, e até mesmo para MS-DOS, bastando apenas compilar o código fonte sem qualquer alteração.

De maneira geral, o código fonte é organizado, com módulos bem definidos, o que facilita a inserção de novas funcionalidades, sendo necessário alterar um número reduzido de arquivos. Vale destacar também a exploração dos recursos de linguagem, o que mostra o conhecimento e a experiência dos programadores. Porém, a compreensão do código não é uma tarefa fácil. O tamanho das funções, em número de páginas, é variado. Muitas estão na faixa de três, algumas chegam a seis e um número bem pequeno pode alcançar até dez páginas.

Foi observado também que existe uma grande preocupação quanto à definição das estruturas de dados, tentando mantê-las reutilizáveis e de rápido acesso à informação armazenada. Nota-se o forte acoplamento entre elas, o que torna o todo complexo e de difícil entendimento.

A eficiência quanto ao acesso aos dados parece ser a grande preocupação no projeto do *make*. Certamente isso é um fator decisivo para uso de tabelas *hashing*. Em alguns testes (incluindo a compilação do próprio GNU *Make*) observou-se o número de elementos que são inseridos na THA e tabela *hashing* do conjunto de variáveis gerais. Verificaram-se poucas colisões em ambas as tabelas, indicativo de um bom desempenho da função de *hashing* utilizada.

Dentre as documentações encontradas sobre *make*, nenhuma descreve a implementação detalhando as estruturas internas e algoritmos empregados pela aplicação. A maior parte deste capítulo é fruto de estudos do código do GNU *Make*, fundamentais para adicionar funcionalidades à aplicação no sentido de torná-la distribuída.

Capítulo 5

MAKE DISTRIBUÍDO

O *make* distribuído é uma evolução natural do *make* convencional devido a necessidades da aplicação dos recursos disponíveis e benefícios alcançados. Outras aplicações também tendem a seguir caminho semelhante, mesmo havendo restrições e dificuldades de migração.

Este capítulo apresenta a solução dada ao problema de transformar o *make* numa aplicação distribuída. São detalhadas a implementação da parte cliente e servidora da aplicação e os controles de acesso à memória compartilhada. Além disso, é realizada uma comparação deste trabalho com outros *makes* semelhantes e apresentado um quadro comparativo entre todos os *makes* descritos no capítulo 3.

5.1 Aplicações Distribuídas

Os sistemas de *software* que executam pedaços de código em paralelo num ambiente de máquinas autônomas e compartilham recursos recebem a denominação de *aplicações distribuídas*. A construção dessas aplicações, chamada de *programação distribuída*, representa uma tarefa de complexidade bem maior que a requerida pelas aplicações não-distribuídas. Em uma rede de computadores a comunicação entre as máquinas geralmente consome muito tempo comparado ao gasto por uma máquina num acesso à memória; além do mais, esse tempo depende de fatores como a topologia da rede, o volume de tráfego de informações, a distância entre as máquinas comunicantes, etc. O tempo de processamento também é variável, tanto em função da capacidade da máquina quanto pela sua carga de trabalho num dado instante.

Todos esses fatores criam incertezas e problemas durante a execução, principalmente quando a aplicação exige alto desempenho. Se uma aplicação está sendo ampliada para suportar a execução num ambiente distribuído é indispensável uma análise detalhada no

sentido de verificar a viabilidade de tal transformação. Fatores como segmentação do problema em tarefas que podem ser executadas em paralelo e o nível de interação entre elas podem ser decisivos na busca do funcionamento eficiente da aplicação. Alem disso, a extensão de aplicações para um ambiente de rede exige soluções particulares, o que torna muito difícil (ou impossível) qualquer tentativa de automatização completa deste processo.

O ambiente distribuído oferece muitos benefícios, incluindo compartilhamento de recursos, heterogeneidade para que a aplicação possa selecionar os recursos computacionais mais convenientes para cada computação, fácil escalabilidade, replicação e possibilidade de migração de processos. Ferramentas de programação, ambientes de execução e mecanismos de comunicação são necessários para explorar estes benefícios e facilitar o tratamento dos problemas relacionados. No Capítulo 6 são discutidos outros aspectos inerentes à construção de aplicações distribuídas.

Em [Bal89], são apresentados três requisitos básicos que distinguem a programação distribuída da seqüencial e que devem ser considerados no desenvolvimento de aplicações distribuídas:

- Uso de múltiplos processadores: diferentes partes de um programa devem ser executadas em vários processadores através de uma política de escolha objetivando fazer uso ótimo dos disponíveis.
- Cooperação entre processos: processos devem ser capazes de se comunicar para possíveis trocas de resultados intermediários e sincronizar suas ações.
- Tolerância a falhas parciais: a aplicação deve sobreviver às falhas em alguns dos processadores, podendo até mesmo se recuperar parcialmente de algumas.

Além desses requisitos, são necessárias preocupações com outros fatores, tais como: desempenho, portabilidade, evolução da aplicação em tamanho e complexidade, interface com o usuário e manutenção de características originais da aplicação centralizada.

5.2 Como tornar o *make* uma aplicação distribuída?

Uma aplicação baseada no modelo de execução seqüencial é dividida em tarefas que obedecem a uma ordem de ativação devido a limitações impostas pelo modelo. Na passagem para o modelo de execução paralelo, várias tarefas podem estar ativas e muitas mensagens podem ser enviadas num dado instante. O desempenho das aplicações é melhorado através do paralelismo na execução das diversas tarefas, que ficam bloqueadas apenas por necessidades de sincronização próprias da aplicação, e não por imposição do modelo de execução.

Num modelo de execução distribuído, o conceito de transparência passa a ser importante [Gon94], possibilitando que tarefas sejam particionadas em sub-tarefas, e estas realizadas sem que os usuários estejam conscientes da troca de mensagens e dos recursos utilizados.

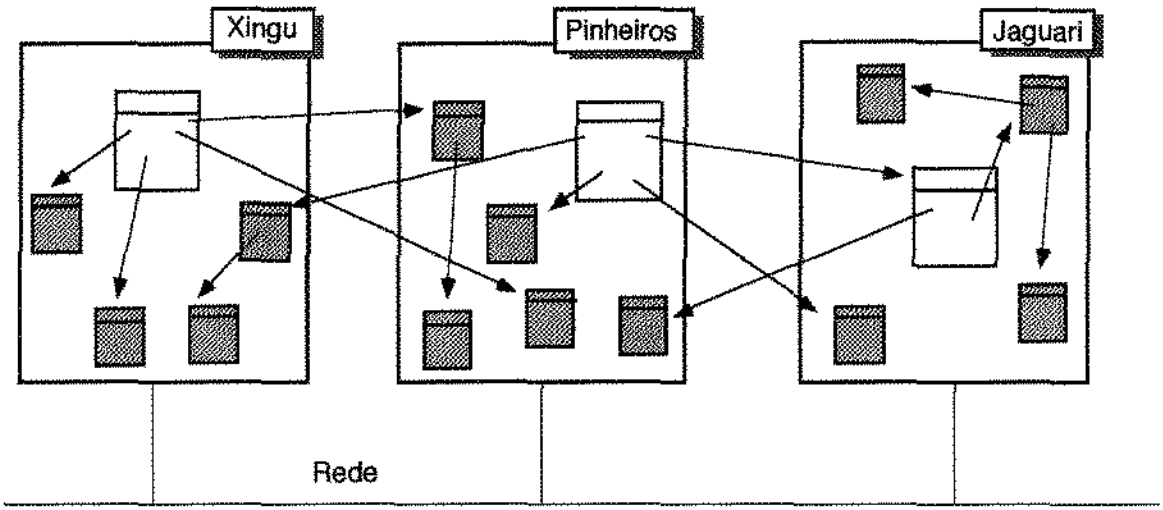


Figura 5.1: Tarefas sendo executadas de forma distribuída.

A adoção do ambiente distribuído como novo ambiente de execução para uma aplicação existente leva a uma difícil pergunta: como modificar uma aplicação de grande porte e baseada no modelo de execução seqüencial para permitir o uso transparente dos recursos com ganho global em eficiência e desempenho? Um grande desafio é conseguir reutilizar o código na nova aplicação sem significativas alterações.

Este trabalho utiliza como base a ferramenta GNU *Make*, baseada na execução seqüencial com recursos de ativação de tarefas concorrentes e não-comunicantes. O ponto central é estendê-la de maneira a explorar o paralelismo oferecido pelos recursos disponíveis no ambiente de rede local, preservando as características originais da aplicação, a portabilidade e a uniformidade da interface com o usuário.

Os comandos do `makefile` são candidatos naturais para a execução distribuída, pelo tempo gasto num processo de *make* e por não haver comunicação entre os processos em execução. O desempenho da aplicação pode ser maximizado se o tempo de execução dos comandos for considerado na escolha do processador. Entretanto, o *make* não conhece a sua complexidade, sendo difícil realizar estimativas baseadas em algum parâmetro, por exemplo, tempo de compilação proporcional ao tamanho do código fonte de um módulo. Uma solução é armazenar as estatísticas de tempos de execução de cada comando para usos futuro do *make*. Mesmo assim, o problema não é completamente resolvido por haver dependências instantâneas do nível de utilização da rede, carga de trabalho das estações e da estação em que foi executado anteriormente.

A solução escolhida na implementação do *make* distribuído é o assunto da próxima seção.

5.3 Implementação do *make* distribuído

Como o *make* é uma aplicação centralizada e *singlethreaded*, os comandos presentes nos nós do grafo são executados localmente usando processos filhos. Utilizando a facilidade de *jobs* concorrentes, em princípio haveria melhor aproveitamento da máquina usada. Entretanto, o usuário deve judiciosamente limitar o número máximo de *jobs* sob risco de piorar o desempenho devido a uma sobrecarga do sistema.

A figura 5.2 ilustra um cenário de execução com um limite de três *jobs* simultâneos. Os nós T5, T6 e T7 estão em processamento e seus respectivos comandos c1, c2 e c1 estão em execução. Ao término de c1 ou c2[1], o próximo comando do respectivo nó será iniciado. No caso da conclusão de c1 do nó T7, este *target* será dado como gerado e um novo nó será escalonado.

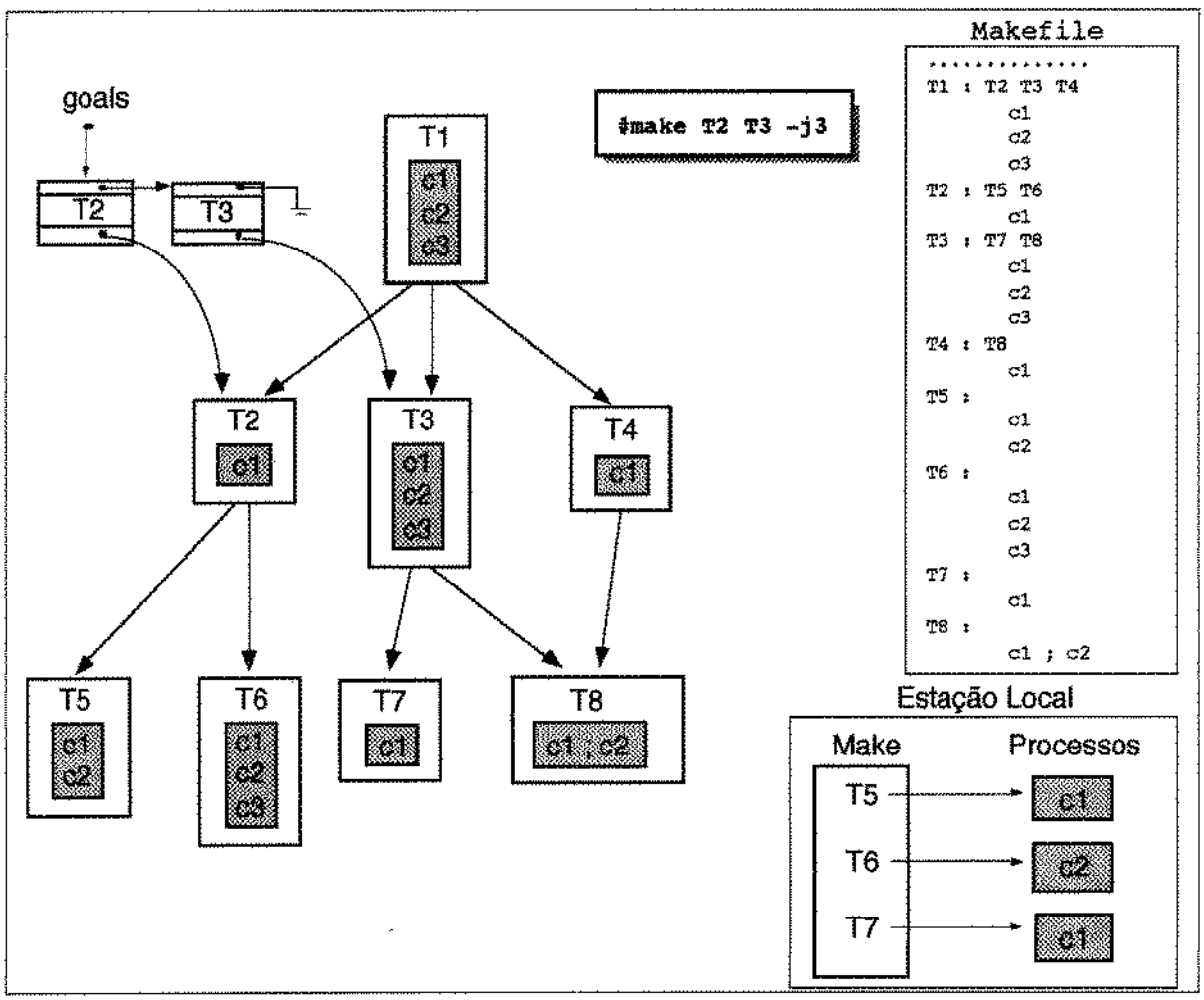


Figura 5.2: `Makefile`, grafo, e execução de *jobs* num dado instante.

[1] Veja a seção 4.4 para maiores detalhes sobre a detecção de término de processos.

Na implementação do MakeD, as tarefas de geração de *targets* são distribuídas entre as estações, pois são elas as grandes consumidoras de tempo e processamento em relação ao trabalho global do *make*. A hierarquia de dependências continua sendo obedecida e, na distribuição do processamento, a escolha do servidor a processar um nó é feita seguindo o critério da Seção 5.3.7.

Devido a preocupações quanto a granularidade, a divisão do processamento é feita por nó e não por comando, ou seja, uma estação recebe um pedido para processar um nó por completo, o que implica em executar todos os seus comandos para construir um *target*. Esta decisão, juntamente com a troca de processos filhos por *threads*[2], causou grandes alterações no código do *make*, inclusive a re-implementação do controle de execução de processos, antes baseado na criação e manipulação de processos filhos.

A implementação do MakeD exigiu o projeto de um servidor para atender a requisições nas estações da rede. Basicamente, há dois módulos independentes na aplicação:

- **Controlador *Make***: cliente da aplicação distribuída que executa na máquina local. É um *make* modificado que executa *jobs* locais e também requisita serviços (através de RPCs síncronas) aos servidores remotos. Decide em qual estação a tarefa será processada com base na capacidade de processamento das estações e no balanceamento da distribuição dos serviços (Seção 5.3.7).
- **Servidor de Processamento (SP)**: servidor multi-tarefa (*multithreaded*) da aplicação distribuída que efetivamente processa as tarefas requisitadas remotamente pelo controlador *make* e devolve os resultados. É executado como um *daemon* em todas as máquinas da rede que participam do processo de *make*, e consegue receber e tratar pedidos de múltiplos controladores *make* ao mesmo tempo.

A figura 5.3 ilustra um possível diagrama de tempo para a comunicação cliente/servidor no MakeD:

[2] Os chamados fluxos de execução são denominados, em inglês, de *lightweight processes* ou *threads* (termo mais usado ao longo do trabalho).

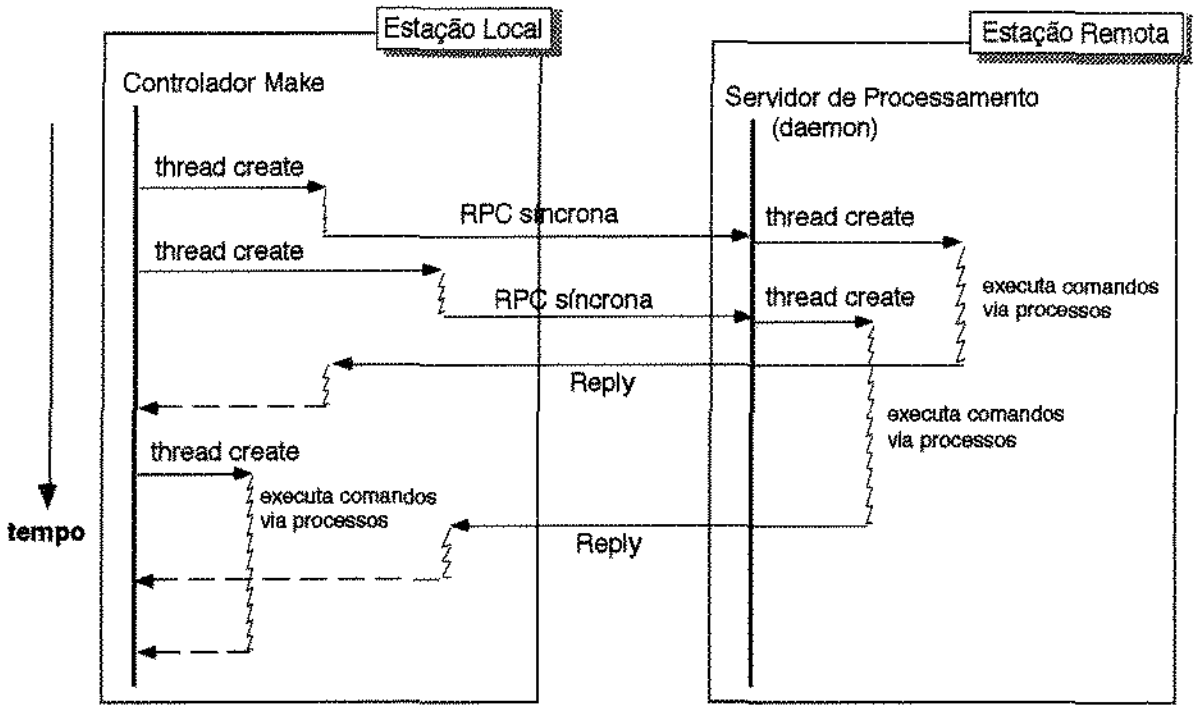


Figura 5.3: Diagrama de tempo para cliente/servidor

A figura 5.4 esquematiza um exemplo do funcionamento e a interação entre as partes do MakeD envolvendo quatro estações. O controlador *make* foi ativado na estação A e configurado para usar os servidores de processamento (SP) disponíveis nas três outras estações. A figura mostra que duas *threads* ativas na estação A estão executando tarefas locais (processamento de nós) e as demais estão requisitando aos servidores remotos (via RPC) para processar nós. No servidor, cada requisição é tratada por uma *thread*.

No caso de uma das requisições para a estação D, um comando a ser executado é a ativação recursiva de uma nova instância do controlador *make* nesta estação, dado pela macro `$(MAKE)` (Secão 5.4.2). Dessa forma, similar ao cliente da estação A, o controlador *make* da estação D passa a utilizar os demais servidores no seu processo de *make*.

5.3.1 Controlador *make*

O controlador *make* interpreta o `makefile`, constrói o grafo de dependências e o percorre em profundidade, disparando o processamento local de nós e pedidos de processamento remoto. Para cada pedido é feita uma conexão com o SP da estação remota através de uma chamada RPC. O controlador *make* também coleta os resultados das execuções locais e remotas, atualiza os nós correspondentes no grafo e faz a interface com o usuário.

Na fase de varredura do grafo, os comandos de cada nó escalonado são empacotados e passados para uma *thread*. No caso de processamento local do nó, a *thread* é criada para iniciar a execução de cada um desses comandos de maneira seqüencial, sem necessidade de

conexão com o servidor, conforme ilustrado pela figura 5.5. Se o processamento for remoto, é escolhida uma estação para efetuá-lo, de acordo com o método descrito na Seção 5.3.7; a *thread* criada insere os parâmetros em estruturas XDR (linguagem de representação de tipos de dados), e faz a chamada RPC, a qual solicita ao SP da máquina escolhida para processar o nó (figura 5.6).

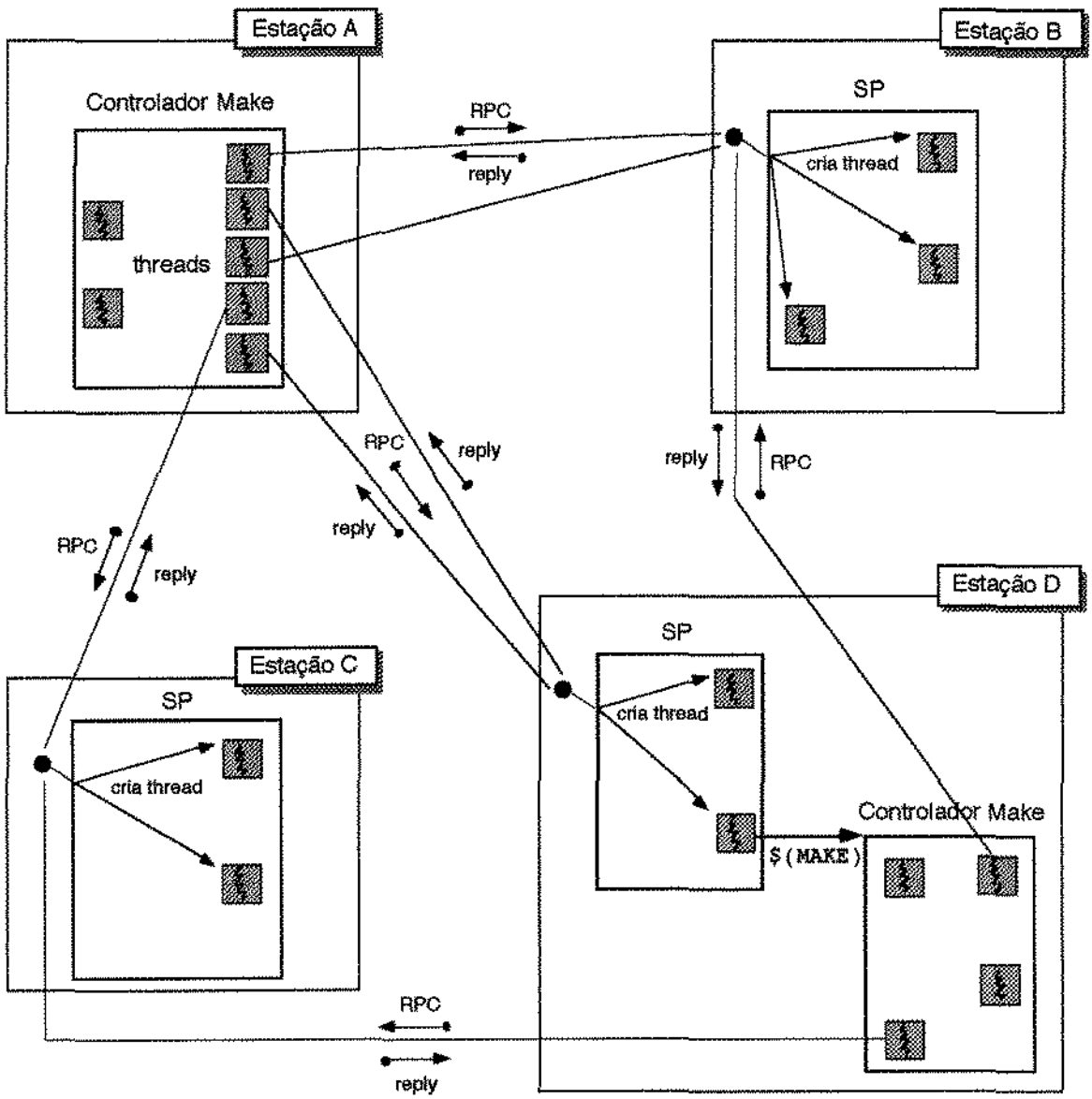


Figura 5.4: Execução do MakeD usando quatro estações.

A RPC é síncrona, ou seja, a *thread* bloqueia à espera dos resultados ou até que seja detectado um *timeout*[3]. Entretanto, para o controlador *make*, o processamento remoto é uma operação assíncrona, pois a RPC é iniciada por uma *thread*. O reconhecimento do término da *thread* é feito verificando o seu *status* e pela interrupção por *software* em alguns pontos do algoritmo de busca.

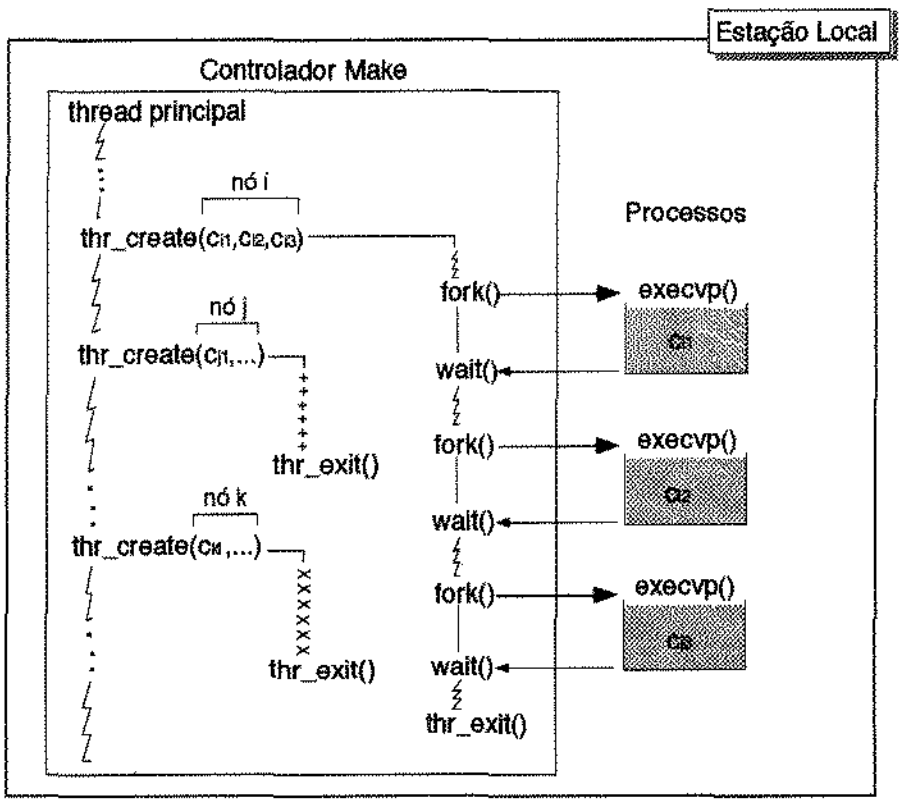


Figura 5.5: Execução local dos comandos de um nó.

A lista de processos em execução (LPE) é substituída por uma lista de *threads* em execução (LTE), onde cada elemento possui o conjunto de tarefas (comandos) do nó, além de um apontador para localizá-lo no grafo (THA)[4]. Uma *thread* ativa possui uma cópia do conjunto de comandos e à medida que um deles finaliza o próximo é disparado, sendo a lista atualizada somente quando todos terminarem (*target* gerado). No *make* original essa lista é atualizada a todo término de processo filho, e o disparo de um novo comando se dá em pontos determinados da busca, o que geralmente implica em perda de tempo.

A LTE é formada pelas *threads* locais e pelas que fazem chamadas RPC remotas – bloqueadas até a devolução dos resultados.

[3] Intervalo de tempo configurável para espera da resposta. Pode ser usado para detectar falhas na estação remota. No controlador *make*, o *timeout* possui um valor *default* para todas as requisições remotas, podendo ser redefinido pelo usuário via variável de ambiente.

[4] As estruturas LPE e THA são descritas no capítulo 4.

5.3.2 Servidor de Processamento

O servidor de processamento (SP) trabalha como um *daemon* à espera de requisições e cria uma nova *thread* para atender a cada uma delas. A *thread* que faz a chamada, no controlador *make* (cliente), fica bloqueada à espera do resultado do processamento do nó. Conceitualmente, as duas compõem uma única "*thread virtual*", abstraindo a separação em diferentes espaços de endereçamento; essa relação é importante para manter a uniformidade da execução de chamadas remotas de processamento de nós.

Cada *thread* atualiza um *target* criando um processo filho para cada comando do nó associado (figura 5.6). Eventuais mensagens (*stdout/stderr*) dos comandos são redirecionadas para um arquivo temporário cujo nome inclui o número da *thread* para que não ocorram conflitos. Finalmente a *thread* devolve ao cliente o conteúdo desse arquivo, juntamente com informações internas como o *status* do processamento do nó.

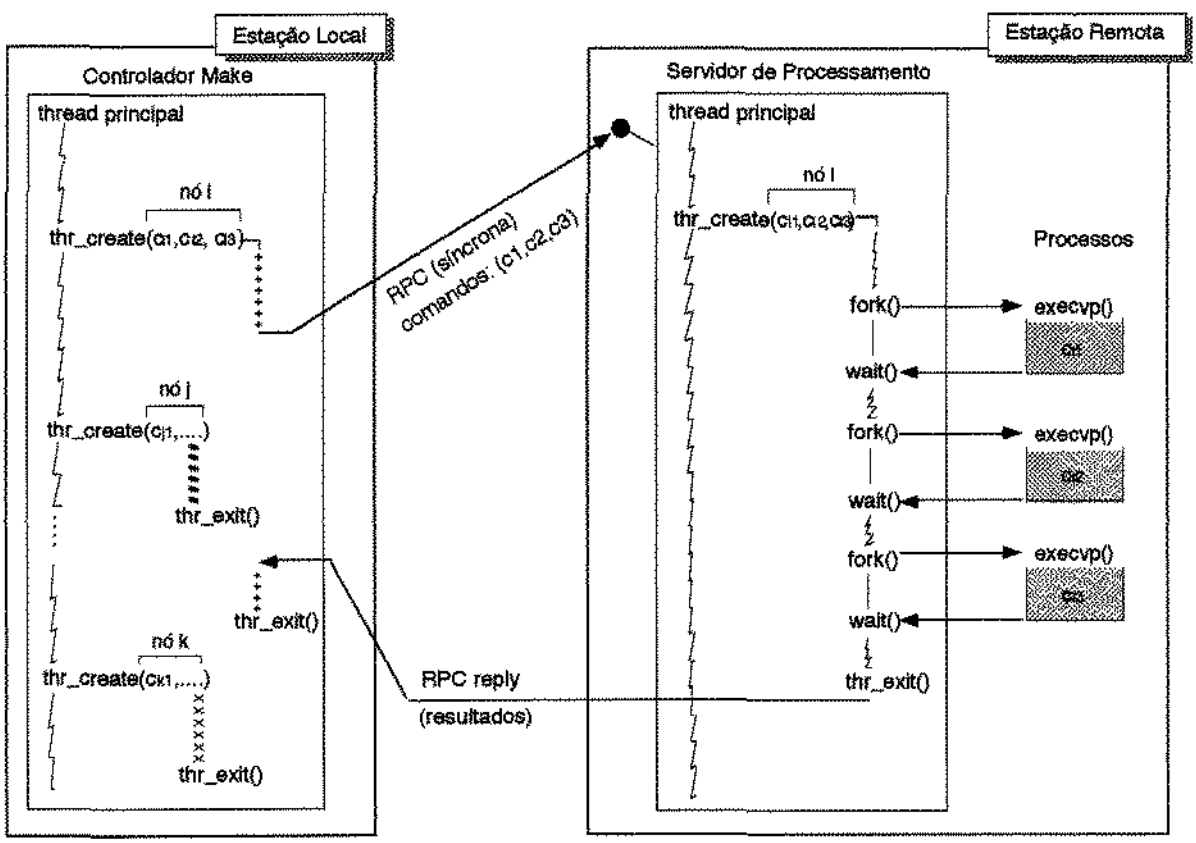


Figura 5.6: Execução remota dos comandos de um nó.

A quantidade máxima[5] de tarefas simultâneas é determinada na configuração do SP e limitada pelos mecanismos *mutex* e variável condicional; variáveis compartilhadas são protegidas de

[5] Discutida em detalhes na Seção 5.3.4 ("Limite de Tarefas Concorrentes no SP").

acessos indevidos pelo *mutex*.

Além do potencial para o atendimento a um grande número de clientes, todos executando de forma paralela, o SP também pode ser utilizado por diversos usuários distribuídos pela rede simultaneamente. Assim, enquanto um usuário está compilando uma aplicação e fazendo requisições de serviços a um SP, outro pode estar atualizando uma biblioteca e também fazendo requisições de serviços ao mesmo servidor.

O uso recursivo do *make* é preservado no controlador *make* e incluído no SP. Este recurso, disponível através da macro `$(MAKE)` no `makefile`, é bastante útil quando existem `makefiles` separados para vários sub-sistemas componentes de um grande projeto. A figura 5.4 ilustra uma chamada recursiva do MakeD na estação D; nesse caso é necessário cuidar da definição do *timeout*, pois, se muito pequeno pode não aguardar pelo retorno da chamada remota que originou o *make* recursivo.

5.3.3 Acesso à memória compartilhada

Processos (ou *threads*) paralelos cooperando através de memória compartilhada utilizam dados comuns para trocar resultados de computação ou esperar por alguma condição necessária para o prosseguimento das computações. Dados dois processos executando sem fazer nenhuma suposição quanto suas velocidades relativas, eles podem fazer acesso a dados comuns em seqüência difícil ou impossível de prever a *priori*. Além do mais, o acesso concorrente a tais dados pode ser feito de maneira totalmente aleatória, podendo o resultado final não refletir corretamente as operações realizadas.

Para sincronizar *threads* que usam variáveis compartilhadas na comunicação o MakeD usa regiões críticas. Uma *região crítica* é uma seqüência de comandos que deve ser executada de maneira indivisível. A execução de regiões de forma livre de interferência de outras regiões críticas é denominada *exclusão mútua* [And83]. A execução mutuamente exclusiva de regiões críticas é a garantia de que os dados compartilhados pelos processos podem ser alterados sem que haja interferência indevida durante essas operações.

O *mutex*, mecanismo utilizado na sincronização entre as *threads* para acesso aos dados compartilhados, limita a execução numa seção crítica que é delimitada pelas chamadas de funções *lock* e *unlock* do *mutex*. A *thread* que adquire o *lock* de uma região de código possui acesso exclusivo a esta até que ela própria libere a região através do *unlock*, permitindo que outra possa fazer o mesmo. Por exemplo, a variável global `environ`[6] do *Make*D possui a variável de sincronização[7] `mlockenv` associada e para alterá-la é feito:

```
(void) mutex_lock(&mlockenv);
environ = parent_environ;
(void) mutex_unlock(&mlockenv);
```

[6] Variável que identifica um vetor de cadeias que constitui o *environment* para um novo processo.
[7] Estrutura de dados em memória pertencente ao mecanismo de sincronização e usada pelas suas primitivas.

O *lock* na variável de sincronização em todas as atualizações de `environ`, compartilhadas pelas *threads*, garante que tal operação seja efetuada de maneira indivisível. As *threads* que tentarem acessar regiões críticas protegidas pela variável acima ficarão bloqueadas até que seja executado o *unlock*.

As alocações e liberações de memória e diversas variáveis globais e estruturas de dados possuem acesso compartilhado pelas *threads* e por isso precisam de proteção do *mutex* durante as modificações.

Na alocação e liberação de memória é preciso considerar também a exclusão mútua. Pelo fato das funções *malloc()* e *free()* serem empregadas ao longo do código do *make* e não serem MT-*Safe* (*Multithreaded Safe*) (Seção 6.3), a solução é a redefinição dessas funções com o uso de uma única variável de sincronização para ambas. Esta mesma variável é também usada para a proteção da liberação de memória alocada pelas rotinas XDR.

```
/* Multithreaded Safe malloc() */
char *MTmalloc( unsigned int size)
{
        char *result;
        (void) mutex_lock(&mlockmem);  /* lock para usar malloc() */
        result = (char *) malloc(size);
        (void) mutex_unlock(&mlockmem); /* libera região de código */
        if (result == 0)
                fatal("Virtual memory exhausted");
        return (result);
}

/* Multithreaded Safe free() */
char *MTfree( char *ptr)
{
        (void) mutex_lock(&mlockmem);   /* lock para usar free() */
        free(ptr);
        (void) mutex_unlock(&mlockmem); /* libera região de código */
}
```

Apesar da LTE ser usada pela *thread* principal (controlador *make*) e pelas *threads* descendentes, o método usado (detalhado na Seção 5.3.6) não requer nenhum mecanismo de exclusão mútua durante as atualizações.

5.3.4 Limite de tarefas concorrentes no SP

Normalmente, em uma estação com SP ativo e fornecendo serviços aos clientes, existem outras aplicações e usuários consumindo os recursos da máquina e que não podem ser perturbados pelas tarefas do SP. Um SP não deve usar os recursos interativos nem sobrecarregar a estação. Para tal, é introduzido um recurso para limitar o número de tarefas simultâneas no SP. Atingido o valor máximo de tarefas concorrentes, as *threads* das próximas requisições são criadas e bloqueadas, e à medida que algumas vão terminando outras vão sendo escalonadas para execução.

A figura 5.7 ilustra um cenário com vários clientes e usuários usando os mesmos SPs.

O controle da quantidade máxima de tarefas simultâneas é implementado com a combinação das primitivas de sincronização *mutex* e variável condicional. A variável condicional é usada para bloquear *threads* atomicamente até que uma condição específica seja satisfeita. O teste da condição é realizado sob a proteção do *lock* de exclusão mútua (*mutex*). Deste modo, a variável condicional é sempre usada em conjunção com o *mutex*.

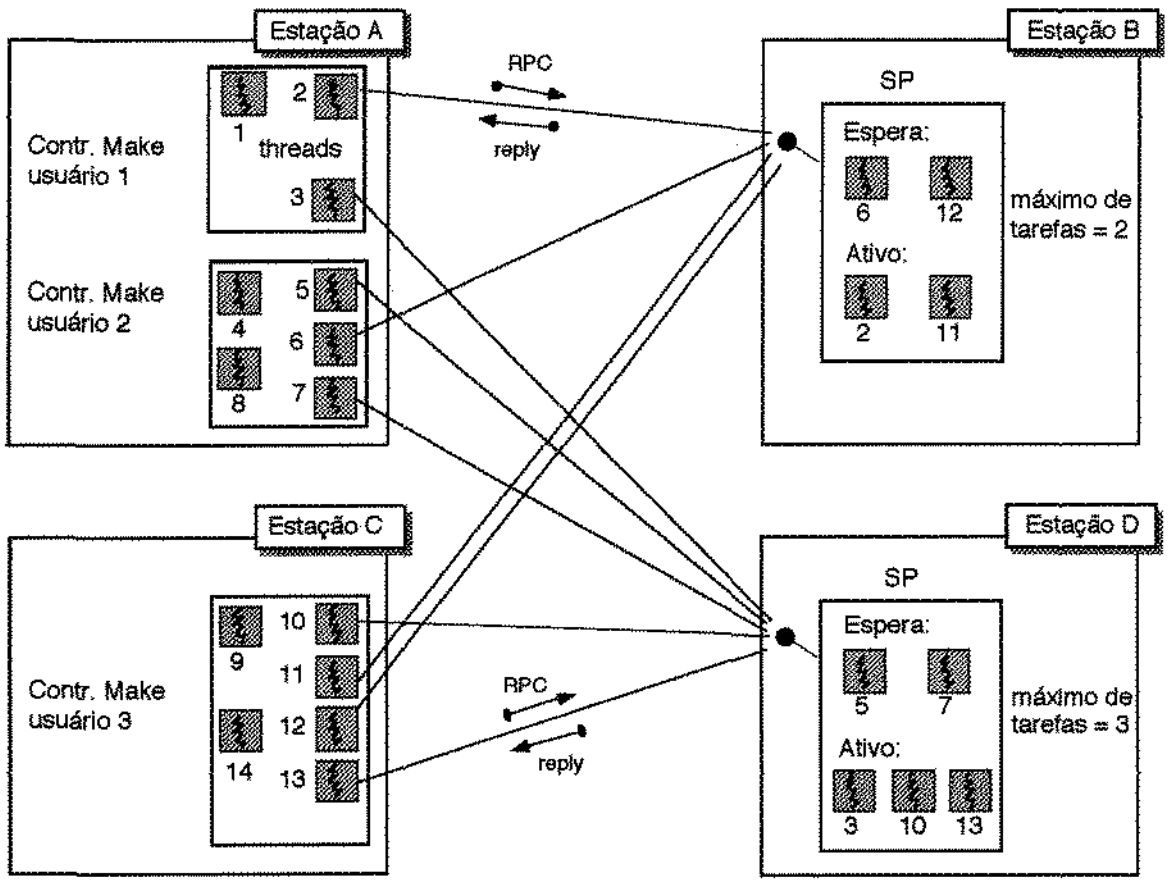


Figura 5.7: SPs atendendo vários clientes/usuários.

As duas funções apresentadas a seguir descrevem o mecanismo adotado na limitação das tarefas, sendo que a primeira verifica a quantidade de tarefas que ainda podem ser disparadas simultaneamente (`jobslots`) e bloqueia a *thread* se necessário (`jobslots <= 0`); a segunda é executada no final de cada *thread* e permite que uma *thread* bloqueada pela primeira função se torne ativa. Em ambas as funções a variável condicional é protegida pelo *mutex*.

Em `BlockThread()`, executada no início de cada *thread*, a função `cond_wait()` libera atomicamente o *mutex* apontado por `mlockcond` e faz com que a *thread* seja bloqueada na variável condicional `condition` se o limite de tarefas já atingiu o máximo (`jobslots =`

0). A *thread* bloqueada pode ser reativada por `cond_signal()` (em `UnblockThread()`).

A função `UnBlockThread()`, executada no fim de cada *thread*, libera um espaço (`jobslots++`). `cond_signal()` ativa uma *thread* que está bloqueada na variável condicional `condition`. Quando não há *threads* bloqueadas nessa variável, `cond_signal()` não tem efeito.

```
/* A thread é bloqueada se jobslots <= 0 */
static void BlockThread(int *jobslots)
{
        (void) mutex_lock(&mlockcond);
        while (*jobslots <= 0)
                cond_wait(&condition, &mlockcond);
        (*jobslots)--;
        (void) mutex_unlock(&mlockcond);
}

/* Ativa uma thread que está bloqueada e incrementa a capacidade de
   tarefas simultâneas */
static void UnblockThread(int *jobslots)
{
        (void) mutex_lock(&mlockcond);
        (*jobslots)++;
        cond_signal(&condition);
        (void) mutex_unlock(&mlockcond);
}
```

O valor limite de tarefas simultâneas é configurado no momento em que o *daemon* do SP é carregado na memória. Desta forma pode ser definido um valor diferente para cada SP, dependendo da capacidade e disponibilidade de uso da estação.

Numa primeira versão do MakeD foi utilizado o mecanismo de semáforo para implementar o controle do número de tarefas concorrentes. Entretanto, a combinação do *mutex* e variável condicional é mais flexível caso seja necessário tornar dinâmica a reconfiguração do limite de tarefas no SP. O semáforo (contador inteiro não-negativo) é tipicamente usado para coordenar acesso a recursos, com o contador inicializado com o número de recursos livres. As *threads* incrementam atomicamente o controlador quando recursos são adicionados (operação V) e o decrementam atomicamente quando recursos são removidos (operação P). Quando o contador chega a zero (nenhum recurso disponível), as *threads* tentando decrementar o semáforo são bloqueadas até que haja recursos (contador maior que zero).

5.3.5 Variáveis de ambiente

Na execução de cada programa, o sistema operacional lhe fornece uma configuração de ambiente (*environment*) composta por um conjunto de variáveis definidas da forma "nome = valor".

No *make*, o ambiente é o mesmo para a execução de todos os comandos de um nó, sendo formado basicamente do ambiente de execução do próprio *make*, das definições e redefinições na linha de comando e do `makefile`. No escalonamento de um nó seu ambiente é gerado e armazenado na sua entrada na LPE, para ser usado na execução de todos os comandos do nó (figura 4.6).

O MakeD aproveita essa formação do ambiente, e através de uma única conexão na rede, envia o nó por completo para processamento remoto num servidor. Além de economizar com o número de conexões (uma por nó, ao invés de uma por comando), o ambiente é enviado uma única vez para cada nó, sendo geralmente o maior parâmetro da chamada RPC.

Uma conexão por comando elevaria muito o número de chamadas RPC e um mesmo ambiente seria enviado várias vezes. Muitos comandos presentes no `makefile` são triviais e não justificam o *overhead* de comunicação; além disso, comumente os comandos de um nó não são paralelizáveis.

5.3.6 Verificação do término de tarefas no controlador *make*

No *make* original, quando um processo filho termina, um sinal `SIGCHLD`[8] [Ste92] é enviado ao processo pai. Este último usa uma das funções *wait()* para capturar o *status* de término e o *pid* (*process identification*) do processo filho. Tal procedimento é realizado nos dois pontos de verificação de término de processos: antes de uma nova busca na lista *goals* (função `Update_Goal()`) e antes de iniciar o processamento do nó escalonado, ambos descritos na Seção 4.3.

Com a substituição de processos filhos por *threads* no controlador *make*, a captura de sinais foi deslocada para dentro da *thread*, ou seja, esta executa a *system call fork1()*[9] e espera pelo término do processo na função *wait()*. Nesse caso, o controlador *make* passa a verificar pelo término de *threads* sem se preocupar com processos filhos (são internos às *threads*). Comparado ao *make* original, embora haja um custo adicional para a criação da *thread* (uma por nó) no caso de processamento local, a ativação seqüencial dos comandos de um nó é realizada automaticamente pela *thread*, sem intervenção do controlador *make*.

O tratamento de término de *threads* é bem mais simples e claro em relação a processos. Quando cada uma delas conclui sua tarefa, o *status* do processamento do nó, de término da própria *thread* e seu ID são atualizados na LTE sem necessidade de proteção do *mutex*, como mostra a função da figura 5.8.

A *thread* principal faz acesso aos dados de um elemento da LTE somente quando a respectiva *thread* ativar seu *status* de término através da operação indivisível `thnode->finished == 1` e em seguida terminar. Durante a execução de *threads* filhas,

[8] Sinal enviado ao processo pai quando um processo muda de estado, em especial quando pára ou termina a execução.
[9] As funções *fork()* e *fork1()* duplicam o espaço de endereçamento do processo pai no filho. Entretanto, *fork1()* duplica somente a *thread* que cria o processo filho, enquanto *fork()* duplica todas *threads* do processo pai.

a *thread* principal pode consultar, inserir e remover elementos da LTE sem prejudicá-las, pois cada *thread* filha possui um apontador (`thnode`) para o respectivo nó na LTE, sem requerer acesso seqüencial para atualização e sem necessidade de exclusão mútua.

```
void FinishThread(struct child *thnode, int result)
{
        thnode->pid = thr_self();       /* ID da thread */
        thnode->exitstatus = result;    /* status de término do nó */
        thnode->finished = 1;           /* status de término da thread */
        (void) mutex_lock(&mlockvar);
        ++dead_thread;                  /* número de threads finalizadas */
        (void) mutex_unlock(&mlockvar);
        thr_exit();                     /* fim da thread */
}
```

Figura 5.8: Atualizações efetuadas antes do término da *thread*

A captura do término da *thread* é realizada nos mesmos pontos dos processos filhos do *make*. Primeiro a LTE é analisada para verificar se alguma *thread* terminou recentemente (`thnode->thfinished == 1`). Se for necessário esperar por alguma conclusão executa-se:

```
while ( thr_join(0, &threadid, 0) == 0)
        ;
```

A função *thr_join()* bloqueia a *thread* principal (controlador *make*) até que seja detectado o término de alguma *thread*, quando seus resultados já estarão armazenados na LTE.

5.3.7 Escolha do SP (distribuição de carga)

O balanceamento de carga para aplicações distribuídas visa otimizar o uso dos recursos computacionais através da distribuição das tarefas entre as estações disponíveis na rede. Ao mesmo tempo que evita a sobrecarga de um sistema, aumenta a utilização média das CPUs e diminui os intervalos ociosos. A implementação a nível de aplicação é menos transparente que a nível de sistema, pois é implementado em cada programa, podendo conduzir a decisões contraditórias pelo fato de diferentes aplicações não conhecerem nada sobre as demais [Sch96].

O MakeD não considera a carga das estações servidoras para distribuir as tarefas. O critério usado para escolher o SP que processará um nó é baseado nas capacidades relativas das estações, dadas em número de tarefas possíveis de serem executadas simultaneamente e na quantidade de tarefas (do MakeD), em processamento, em cada uma das estações. As capacidades são configuradas pelo usuário para controle de distribuição no controlador *make*. Assim, se um usuário especificar que durante o processo de *make* de seu programa a estação A pode executar dois *jobs* concorrentes e a estação B quatro, então a capacidade de B é o dobro de A. Enquanto isso, outro usuário pode estar usando estas e outras estações

para processar requisições de seu controlador *make* e definir capacidades distintas (como 4 e 8) para A e B.

Para dividir a carga de trabalho, o algoritmo considera a capacidade da estação em dois estágios: inicialmente leva em conta apenas a sua metade; posteriormente considera a capacidade total. O esquema é simples: a estação a processar um nó é aquela que possui maior capacidade disponível no momento, calculada através da diferença entre a metade da capacidade da estação e o número de nós que ela está processando (provenientes de um controlador *make*). A estação livre (sem nós processando) tem prioridade para receber tarefas. Mas, se todas as estações estiverem com a metade da capacidade completa (1, 2 e 4 no exemplo da figura 5.9), então é usada a capacidade total das estações para decidir qual delas será escalonada para a próxima tarefa. Neste caso, o algoritmo calcula a diferença entre a capacidade total e o número de tarefas em processamento (provenientes de um controlador *make*), escolhendo aquela que possui maior capacidade disponível.

A figura 5.9 ilustra um cenário com três SPs e as respectivas capacidades configuradas para uso do cliente (controlador *make*). As capacidades configuradas para uso do cliente são independentes dos limites de tarefas configurados na inicialização de cada SP (Seção 5.3.4). Na figura, é apresentada uma seqüência de distribuição de tarefas para as estações A, B e C, as quais estão com 1, 3 e 6 *jobs* executando, respectivamente. Os números representam a quantidade de nós processando na estação e a seqüência (vertical) em que aparecem indica a ordem de distribuição. Caso nenhum *job* termine, os dois próximos serão ativados em C e A, respectivamente. Para facilitar, a lista é ordenada de acordo com as capacidades (crescentes) das estações e aquelas com menor capacidade têem prioridade no caso de empate.

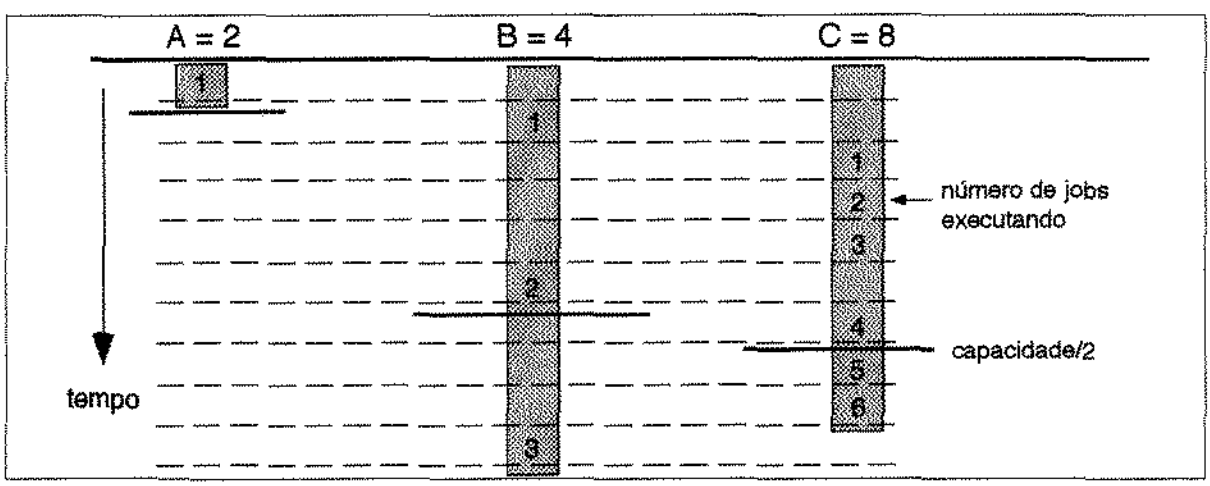


Figura 5.9: Cenário de distribuição de tarefas para SPs.

O critério garante que todas as estações serão utilizadas com no mínimo um *job* num dado instante, independente de sua capacidade. A divisão de trabalho é realizada de forma

balanceada de acordo com a configuração do usuário, ideal para um ambiente homogêneo (capacidades idênticas) em períodos de pouca utilização dos sistemas.

A grande desvantagem do algoritmo é não considerar a carga dos SPs. Na figura acima, apesar da estação C ser bastante rápida comparada a A, ela pode estar sobrecarregada com muitos processos/usuários ativos e até mesmo processando pedidos de outros controladores *make*. Além disso, o usuário precisa conhecer as estações utilizadas para definir suas capacidades antes de iniciar o processo de *make*.

5.3.8 Considerações sobre a implementação do MakeD

O *make* distribuído mantém toda a funcionalidade de seu equivalente centralizado. A distribuição de tarefas entre estações é decidida e realizada de forma transparente ao usuário, sendo que a única diferença perceptível fica por conta da agilidade no desempenho global da aplicação.

A plataforma de *hardware* da implementação corrente é constituída por um conjunto de *workstations Sun*[10] interligadas em rede. O sistema operacional usado é o Solaris 2.5[11] (*System V*), e foi seguido o seu padrão de *threads*; porém a mudança necessária para tornar a aplicação compatível com *threads* POSIX é mínima. A portabilidade do GNU *Make* foi mantida; a única restrição é que o sistema operacional deve suportar *multithreading*.

A manutenção de processos filhos no MakeD tornaria mais complexo o controle da memória compartilhada para comunicação entre processos pai e filhos. Seria necessário ainda gerenciar a alocação e liberação de memória. As *threads*, além de exigirem menos recursos, permitem o uso de primitivas de sincronização para acesso à memória compartilhada. O código é mais compacto, elegante e fácil de entender se comparado à alternativa com processos convencionais. Além disso, com a mudança de implementação, o código foi bastante reduzido e mais fácil de ser mantido. Por outro lado, a programação *multithreaded* requer mais cuidado e disciplina e é dificultada pela falta de ferramentas de depuração adequadas.

Com relação às primitivas de sincronização usadas (seguindo o padrão Solaris), o *mutex* é o mecanismo mais básico e eficiente tanto no uso de memória quanto em tempo de execução. É usado para serializar o acesso a um recurso; a variável condicional vem em seguida, e seu uso básico é bloquear numa troca de estado, sempre em conjunção com o mutex; O semáforo usa mais memória que a variável condicional, sendo de uso mais fácil em algumas circunstâncias por simplificar o código [Sun94a].

As dúvidas iniciais deste trabalho eram principalmente descobrir quais as partes do código precisariam ser alteradas. Desta forma, foi implementada uma versão inicial do MakeD em que não foram usados recursos de multi-tarefa, nem mesmo no SP. Esta versão foi o passo

[10] Sun Microsystems, Inc.
[11] SunOS 5.5 Operating System.

inicial para verificar que o objetivo era factível. A partir disso, tornou-se o SP multi-tarefa e, em seguida, o controlador *make*.

5.4 Análise comparativa deste trabalho

Ao implementar a extensão do *make* para suportar a distribuição e execução de tarefas num ambiente de rede local, é necessário situar este trabalho junto a outros *makes* distribuídos já implementados. Para isso, será feita uma análise comparativa, entre o MakeD e dois outros, considerando aspectos de implementação e algumas características inerentes de cada um. Os *makes* considerados na análise são o Lmake (Seção 3.4) e o Dmake (Seção 3.5), pois em termos de funcionalidades e implementação, são os que possuem mais semelhanças com o MakeD. O Pmake (Seção 3.6) é específico para o ambiente Paragon e o PVMMake (Seção 3.3) centraliza o processo de *make* em cada sistema, objetivando facilitar o desenvolvimento de aplcações portáveis para vários ambientes.

5.4.1 Lmake

Similar ao MakeD, o Lmake usa o nó (conjunto de comandos) como a unidade para a execução num servidor com o *lmaked* ativo. Dessa forma, o *environment* é também passado uma única vez na execução de todos os comandos de um nó. Entretanto, o Lmake agrupa todos os comandos do nó numa linha de comando e invoca uma *shell* para executá-los, através da criação de um único processo filho (primitiva *fork()* seguida de *exec()*). Essa otimização viola a semântica padrão de *make* e cria incompatibilidades em alguns `makefiles`.

A decisão do Lmake torna-se mais eficiente quando a regra (no `makefile`) possui mais de um comando. Por exemplo, se há cinco comandos numa regra, o Lmake usa seis processos (*shell* mais um processo por comando) para executá-los, enquanto o MakeD usa dez processos (uma *shell* por comando), desde que todos os comandos necessitem da *shell*. Entretanto, a mudança do *environment* de um comando para outro é um dos problemas que podem ocorrer com o Lmake. Além disso, não é possível ignorar erros de algum comando específico.

A nível de aplicação, o Lmake não gerencia diretamente processos filhos na estação local para requisições de processamento remoto. É utilizada a rotina assíncrona `pvm_send()` do PVM. No entanto, a implementação do `pvm_send()` deve usar processos, pois o PVM não suporta *threads*. Uma vantagem é o nível de abstração fornecido ao programador.

No Lmake não existe a funcionalidade de uso da multi-tarefa nos sistemas da rede. Cada estação possui somente uma tarefa ativa num dado instante. A utilização deste recurso no MakeD visa explorar a capacidade de processamento das estações e melhorar a eficiência da aplicação distribuída.

O balanceamento de carga, apesar de simples, é uma característica presente no Lmake. A escolha do servidor a processar um nó é baseada na combinação da carga média (*load average*) das estações e em velocidades relativas configuradas pelo usuário. O MakeD faz a escolha baseado nas capacidades relativas das estações em termos da multi-tarefa e na quantidade de tarefas em processamento nos SPs, mas não leva em conta a carga corrente de cada uma das estações.

Um fator negativo de ambos os *makes* é a necessidade da configuração da capacidade relativa de processamento de cada estação envolvida. Talvez uma solução mais flexível seria usar um método que consiga distribuir as compilações baseado na experiência, ou seja, inicialmente todas as estações são consideradas equivalentes, mas a cada execução são acumuladas informações como por exemplo, o número de processos ativos, uso da CPU, paginação, *swapping*, etc, as quais influenciariam as próximas decisões. Com o passar do tempo, as tarefas, em média serão distribuídas de maneira mais justa.

A divisão em duas fases implica no armazenamento pela fase serial dos comandos e dependências num arquivo temporário, depois usado pela fase paralela para a construção de um novo grafo de dependências. Tal passo adicional é desnecessário; o grafo poderia ser gerado diretamente pela fase serial.

Com relação ao *timeout*, não existe preocupação no projeto do Lmake. É definido um valor muito alto e não acessível ao usuário.

A falta de documentação sobre o Lmake implicou no entendimento de várias partes do seu código fonte (disponível em domínio público) para compreender seu funcionamento e compará-lo com o MakeD. De certa forma, o entendimento do Lmake foi facilitado com a implementação do MakeD, através do GNU *Make*, pois o Lmake é baseado numa versão simplificada deste último – versão 3.67.

5.4.2 Dmake

No Dmake, cada requisição de processamento solicita a execução de um único comando, sendo o *environment* enviado como um dos parâmetros. Assim, se para processar um nó e gerar um *target* são necessários quatro comandos, o Dmake realiza seqüencialmente quatro solicitações, para processamento local ou remoto de tarefas. Desta maneira, além da desvantagem do fator granularidade, há a necessidade de inclusão do mesmo *environment* tantas vezes quantas forem o número de comandos de um nó, enquanto comumente um `makefile` inclui muitas tarefas simples em que o tempo de execução não justifica o estabelecimento de uma conexão e transferência dos dados.

Tanto as requisições locais quanto remotas, são feitas usando os programas *spew* (cliente) e *rspew* (servidor), semelhantes ao `rsh` (*remote shell*), mas retornando o código de término do comando, agregado aos resultados. Entretanto, são necessários três processos (*spew*, *feeder* e `rsh`) locais e dois remotos (`rshd` e *rspew*) adicionais para a execução remota de cada

comando (veja figura 3.3). Como as execuções locais e remotas são simétricas, também são necessários cinco processos para a execução local de cada comando. Por outro lado, o MakeD necessita somente de uma *thread* adicional em cada estação para para solicitar o processamento de um nó por completo. Além disso, para usar o Dmake é necessário realizar algumas alterações num `makefile` existente, incluindo a definição do compilador (`CC = spew $(CC)`).

A quantidade de processos adicionais necessários à execução de comandos é um problema que limita o uso do Dmake a poucas estações e a um baixo grau de concorrência em cada uma delas. A estação local é a mais sacrificada com a rápida ocupação da sua tabela de processos à medida que aumentam as compilações simultâneas. O próprio autor sugere formas de amenizar o problema através da reconfiguração do sistema operacional e alerta sobre a necessidade de tornar o processo de execução de comandos mais eficiente. Isso foi comprovado pelos testes de desempenho apresentados no capítulo 7.

Sem o DQS, não há uma política de escolha da estação a executar um dado comando, nem um mecanismo para diferenciar as capacidades de processamento de cada uma delas; é obedecida apenas a ordem especificada na configuração do usuário. Com o DQS, este determina a estação a ser usada através do seu mecanismo de balanceamento de carga.

Uma diferença importante do MakeD em relação ao Lmake e Dmake é a distinção no tratamento de compilações locais e remotas. As compilações locais são ativadas pelo próprio controlador *make*, enquanto para as remotas há a comunicação com o SP em outra máquina. Já nos *makes* citados, toda compilação é ativada pelo servidor, seguindo o modelo cliente/servidor. Dessa forma o MakeD evita o *overhead* de comunicação com um SP na própria estação ao custo de maior complexidade na gerência de comandos.

Semelhante ao Lmake, a falta de documentação tornou necessário o entendimento do código fonte do Dmake (disponível em domínio público). Além disso, conseguiu-se portá-lo para o ambiente Solaris 2.5 visando a análise de seu desempenho em relação ao MakeD, apresentada no capítulo 7.

5.5 Comparação entre makes

A tabela 5.1 apresenta um comparativo referente às características e funcionalidades do MakeD e dos *makes* descritos no Capítulo 3. Vale destacar que o PVMMake identifica os arquivos a ser atualizados e os transfere para cada sistema da máquina virtual PVM, onde serão compilados de maneira centralizada. Por outro lado, o *Make* Paralelo distribui as compilações individuais sem considerar a interdependência entre os arquivos, e a divisão de tarefas entre as estações é feita estaticamente.

	PVMMake	**Lmake**	**Dmake**	**Pmake**	***Make* Paralelo**	**MakeD**
Make distribuído ou paralelo?	distribuído	distribuído	distribuído	paralelo	distribuído	distribuído
Suporte ao uso recursivo do *make*	não	não	sim	sim	não	sim
Suporte a regras implícitas no `Makefile`	não	sim (padrão antigo)	sim	sim	não	sim
Uso de otimização na execução de comandos	não	não	sim	não	não	sim
`Makefile` usado é compatível com os padrões?	não	sim	não	sim	não	sim
Suprte à multi-tarefa em cada sistema da rede	não	não	sim	sim	não	sim
Mecanismo de comunicação usado na rede ou entre processadores no sistema	PVM	PVM	rsh baseado no *spew* e *rspew*	*system calls* do Paragon	Linda	RPC
balanceamento de carga?	não	sim	sim (uso do DQS)	sim	não	não
Granularidade: distribuição de compilações por comando ou nó?	compilação centralizada em cada sistema	nó	comando	comando	comando	nó
Exige um sistema de arquivos comum entre as estações ou processadores?	não	sim (NFS)	sim (NFS)	sim (Paragon)	sim (NFS)	sim (NFS)
Funcionalidades compatíveis com o *make* base?	concebido como distribuído	não	sim	não	concebido como distribuído	sim
Recursos adicionais usados na estação local para requisitar o processamento remoto de um nó	não conhecido	1 processo	3 processos para cada comando do nó	1 processo para cada comando do nó	não conhecido	1 *thread*

Tabela 5.1: Quadro comparativo entre *makes*

Capítulo 6

DISTRIBUIÇÃO DE APLICAÇÕES

Um conjunto de estações de grande poder computacional conectadas por uma rede de alta velocidade tem se tornado um importante recurso para satisfazer as necessidades da computação em larga escala [Tand96]. Isso é uma alternativa atraente para melhorar o desempenho de aplicações seqüenciais, através da identificação de tarefas que podem ser divididas em sub-tarefas passíveis de execução de maneira paralela em máquinas separadas, e sem gerar um alto *overhead* de comunicação para a coordenação das sub-tarefas. Fatores como a natureza da aplicação, desempenho, algoritmos empregados, custo da comunicação entre processadores, tolerância a falhas, entre outros, são importantes durante a análise de viabilidade da extensão da aplicação para um ambiente de rede distribuído. Este tipo de análise torna-se valioso quando se verifica que existem inúmeras aplicações que serão beneficiadas com a utilização da capacidade ociosa de processadores da rede.

Na mudança do ambiente de execução, a rede torna-se tão importante quanto os sistemas individuais. Ela é o canal que permite bancos de dados, aplicações e outros componentes de software operarem eficientemente. A disponibilidade e capacidade da rede são de vital importância. Suas limitações podem levar a severos impactos nas operações globais.

Neste capítulo são discutidos em termos gerais alguns aspectos de distribuição de aplicações relacionadas com a experiência de implementação do *make* distribuído a partir da versão centralizada.

6.1 Natureza da aplicação

Certamente nem todas as aplicações se beneficiam de um ambiente de rede. As razões são várias, e vão desde características próprias da aplicação até limitações impostas pelo

ambiente distribuído, como o tempo de comunicação. Dessa forma, a natureza da aplicação é um fator decisivo para justificar ou não a mudança do ambiente de execução.

A constante busca na melhoria do desempenho, tempo consumido e uso dos recursos computacionais são motivos que têm levado à análise no sentido de migrar a aplicação para um sistema com múltiplos processadores. Entretanto, se existe muita interação entre as partes de um programa e o tempo de comunicação é crítico, certamente um sistema multiprocessador é o ideal, e tentativas no sentido de torná-lo distribuído podem não ser satisfatórias. Outro fator importante é a relação entre custo de implementação para paralelizar o código e os benefícios alcançados com tal esforço. Em muitos casos, como em [Mor96], a paralelização de somente porções do código serial, exportando-os para outros processadores, é uma solução conveniente para melhorar o desempelho global com reduzido esforço de implementação.

O *make* é uma aplicação natural para qualquer sistema multiprocessador, em que as compilações podem ser feitas em paralelo em processadores separados [Fle89]. A distribuição destas tarefas é justificável pelo tempo gasto e esforço de implementação requerido. Além disso, não existe comunicação entre os processos filhos. Por outro lado, [Fle89] não conseguiu resultados satisfatórios em testes de multiplicação distribuída de matrizes, discutidos em [Gel85]. O tempo de comunicação na rede comparado com o tempo da tarefa realizada e o método empregado não justificaram a mudança no algoritmo, piorando o desempenho da aplicação.

6.2 Segmentação do problema em tarefas

Segmentar um problema em tarefas possíveis de serem executadas concorrentemente é altamente dependente da aplicação, da arquitetura e da organização do sistema de processamento que será empregado. Os recursos de *softwares* utilizados (como sistema operacional, mecanismos de comunicação e linguagens) também influem grandemente no que pode ser conseguido neste sentido.

O primeiro passo é conhecer os algoritmos empregados na tentativa de identificar tarefas que podem ser executadas paralelamente, e escalonadas de tal forma que o tempo de execução seja minimizado. Se não for possível segmentar o problema em tarefas, então não há como distribuir a aplicação. Uma vez identificado o paralelismo, deve-se dividir as tarefas em processos paralelos e tomar decisões quanto à *granularidade* [Bal89]. Há vantagens em distribuir uma aplicação somente se grande parte do tempo for gasto em computações, sendo as comunicações entre processos pouco freqüentes. Entretanto, tarefas com granularidade muito grande podem reduzir o paralelismo; por outro lado, a pequena granularidade pode exigir muito *overhead* de comunicação.

Outra consideração a ser analisada é o número de mensagens enviadas. O número de bytes recebidos pode ser enviado em várias mensagens pequenas ou em poucas mensagens

grandes. Poucas mensagens grandes reduzem o tempo total de *start-up* e podem não decrementar o tempo de execução global. Há casos em que o *overhead* de pequenas mensagens é compensado pelas computações. Porém, a capacidade de compensar comunicação com computação e o número ótimo de mensagens enviadas são dependentes da aplicação [Gei94a].

O entendimento do código fonte do GNU *Make* foi sem dúvida a fase mais difícil deste trabalho, embora a identificação de tarefas seja facilitada pela natureza da aplicação. No MakeD, decidiu-se agrupar os comandos de um nó para execução remota ao invés de enviar comando a comando. Adotou-se esse método porque muitos comandos triviais não justificam o *overhead* de transmissão (que inclui o ambiente de execução) e devolução dos resultados. Além disso, na prática comumente os comandos de um nó não são paralelizáveis. No exemplo abaixo, o tempo gasto pelo segundo comando que participa da geração do *target* `main.o` não justifica sua separação do primeiro para uma chamada remota, e o segundo comando só pode ser ativado após o término do primeiro.

```
main.o : main.c smk.h
        gcc -c -g main.c -o main.o
        @echo gerado o target main.o ; date
```

Existem casos em que os comandos da regra exigem muita computação e não necessitam de execução seqüencial, podendo ser paralelizados. Para cobrir este caso, seria interessante adicionar explicitamente a possibilidade de execução simultânea de seus comandos. Por exemplo, o caracter `&` após o "`:`" do *target* poderia dizer ao MakeD que é possível executar os comandos da regra simultaneamente:

```
target  :&  dep1 dep2 dep3
        comando1
        comando2
        comando3
```

Dessa forma, conhecendo a semântica dos comandos o usuário, poderia utilizar esse mecanismo para controlar melhor a granularidade das tarefas e a divisão do processamento na rede, visando minizar o tempo global.

Por outro lado, nos casos em que todos os comandos da regra são muito simples, outra extensão semelhante poderia forçar a execução local (pelo controlador *make*) de todos os comandos da regra para evitar usar os SPs remotos em computações simples e rápidas.

6.3 Mecanismos de programação concorrente

Na construção de aplicações distribuídas é necessário pensar em mecanismos de programação concorrente, principalmente na criação de servidores que irão atender a vários clientes, todos executando de forma paralela. Pedidos para a execução de serviços,

ocorrendo de forma não-determinística, exigem do servidor a capacidade de sincronizar o atendimento dessas requisições.

Os mecanismos de execução concorrente visam um aproveitamento efetivo de recursos de processamento, respeitando restrições de dependência entre resultados intermediários e acesso disciplinado a dados compartilhados. Assim, na resolução de um problema, as soluções são mais naturalmente representadas na forma de atividades paralelas, atuando de forma cooperativa ou competitiva, que colaboram cada uma com uma parcela da solução final [Gon94a].

A inserção da concorrência na aplicação não é tarefa fácil. A necessidade de utilização dos recursos de processamento e os benefícios conseguidos com o uso de múltiplas *threads* em execução, exigem a revisão do projeto no sentido de utilizar tais recursos através do suporte à multi-tarefa. A programação *multithreaded* requer mais cuidado e disciplina comparada à seqüencial, devido ao compartilhamento dos dados e outros recursos do processo.

No MakeD, tanto o controlador *make* quanto o servidor de processamento possuem a capacidade multi-tarefa. Com as *threads*, muitos dados passaram a ser compartilhados pela *thread* principal e descendentes; a alocação/liberação de memória e o acesso aos dados globais são protegidos pelas primitivas de sincronização *mutex* e variável condicional. O uso de *threads* no controlador *make* não objetiva eficiência, e sim clareza e facilidade de manutenção do código, além de torná-lo mais conciso, pois tipicamente precisa continuar usando processos filhos (gerenciados pelas *threads*) para executar comandos. Já no SP o uso de *threads* simplifica o atendimento eficiente de vários clientes ao mesmo tempo.

Para aplicações em que *threads* substituem processos, o ganho de desempenho pode ser considerável. A Tabela 6.1 [Sun94] mostra os tempos[1] consumidos para criar uma *thread*[2] e um processo.

Operação	Microssegundos	Razão
criar unbound thread	52	1
criar bound thread	350	6.7
fork()	1700	32.7

Tabela 6.1: Tempos de criação de *thread* e processo

A criação de *threads* é barata comparada a de processos, pois não é necessário um novo espaço de endereçamento. As *threads* de um processo compartilham quase tudo, por isso a comunicação entre elas é simples. Assim, os dados produzidos por uma *thread* são

[1] Valores obtidos numa SPARCstation 2 (Sun 4/75) usando o *timer built-in*.

[2] No Solaris, *bound threads* são *threads* criadas e associadas permanentemente a *Lightweight Processes* (LWP) a nível de *kernel*. Cada *thread* é ligada a um único LWP. *Unbound threads* podem migrar de um LWP para outro após um bloqueio para sincronização.

facilmente disponibilizados para as demais. Por outro lado, o uso de *threads* só se justifica se a quantidade de instruções de máquina executadas for da ordem de milhares.

Uso de *threads* para paralelizar algoritmos

Uma das potencialidades do uso de *threads* é diminuir o tempo de execução de programas com computação intensiva em multiprocessadores com memória compartilhada ou em multicomputadores. O passo mais importante na escrita do programa paralelo/distribuído é pensar cuidadosamente na estrutura global do programa e nas estruturas computacionais usadas pelas *threads* para dividir o trabalho num conjunto de tarefas (*threads*) de forma que [Kle96]:

- Haja pouca interação entre as *threads*;
- Os dados compartilhados pelas *threads* estejam contidos em poucas estruturas de dados simples podendo ser protegidas por mecanismos de sincronização a fim de evitar acessos indevidos;
- Todas as *threads* tenham tarefas computacionais semelhantes.

As operações de sincronização, necessárias para fornecer um ambiente *multithreaded* efetivo, gastam um tempo considerável para serem executadas, até mesmo quando não há contenção para recursos compartilhados.

A lei de Amdahl [Kle96] afirma que na maioria das computações há algumas porções que devem ser executadas serialmente e outras que podem ser paralelizadas. Se as porções seriais dominarem o tempo total de execução, o ganho conseguido pelas porções paralelas *multithreaded* não será significativo.

***Multithreading* código existente**

Em muitos casos é necessário alterar substancialmente a estrutura de uma aplicação *single-threaded* para realizar o uso efetivo de *threads*. Entretanto, há tipos de aplicações que podem ser parcialmente *multithreaded* de forma a simplificar uma estrutura complexa, tais como aplicações de visualização gráfica para interface com o usuário e aplicações multimídia. Funções simples podem ser escritas para cada atividade, facilitando o projeto de programas complexos e tornando-o mais adaptável a variações das demandas do usuário. O uso de novas *system calls* e a revisão daquelas já empregadas no sistema devem obedecer a classificação em categorias que definem o nível de suporte (MT-Level – *Multithreaded Level*) a *threads*, pois nem todas as *system calls* usadas na programação seqüencial podem ser usadas na multi-tarefa [Sun94a]. As principais categorias são:

- *Unsafe*: rotinas que contém dados globais e estáticos não-protegidos. Não fornecem segurança para uso numa aplicação *multithreaded*.

- *Safe*: podem ser usadas numa aplicação *multithreaded*, mas não suportam concorrência de múltiplas *threads*. Seu código é serializável pela proteção do *mutex*.
- MT-*Safe*: rotinas completamente preparadas para acesso *multithreaded*. Protegem seus dados estáticos e globais com *locks*, suportam concorrência e seu uso não afeta o desempenho.

Somente as funções classificadas como MT-*Safe* podem potencialmente ser chamadas diretamente por uma *thread*, pois suportam um nível razoável de concorrência. As demais[3] só podem ser usadas com segurança pela *thread* principal, ou então, através da sincronização de acesso às funções pelas *threads*. A proteção sincronizada deve incluir o acesso a elementos da função, tais como código de erros globais. Entretanto, este método implica em custo no desempenho.

Algumas funções não foram transformadas em MT-*Safe* para que seu desempenho não seja afetado quando utilizadas em aplicações seqüenciais e por possuírem interface *Unsafe*; por exemplo, a função pode retornar um apontador para um *buffer* alocado estaticamente [Sun94a].

Além do cuidado no uso de funções, outra dificuldade na multi-tarefa é a identificação de erros no programa, que freqüentemente se comporta de maneira diferente para duas execuções sucessivas com entradas idênticas, pelo fato da mudança na ordem de escalonamento das *threads*. Há também a falta de ferramentas de depuração adequadas para aplicações multi-tarefa, principalmente aplicações distribuídas.

A detecção de violações da disciplina de programação *multithreaded* pode ser difícil em sistemas grandes e complexos. Um exemplo simples desse erro é a falha em adquirir um *lock* de proteção para o acesso a uma variável de dados compartilhados. Erros causados por tal violação são intermitentes por natureza, pois somente seqüências particulares de execução causarão o comportamento incorreto. Mesmo quando o erro é observado (usualmente por um *crash*) é difícil determinar a sua causa original.

6.4 Mecanismos de comunicação

No desenvolvimento de uma aplicação em que é necessária a comunicação entre processos cooperantes num ambiente de rede distribuído, a escolha do mecanismo de comunicação tem influência direta no projeto, funcionamento e desempenho da aplicação. Dessa forma, o mecanismo a ser utilizado é geralmente dependente da arquitetura da aplicação e da efetiva utilização dos recursos. Duas técnicas bastante utilizadas para implementar principalmente sistemas cliente/servidor são RPC (*Remote Procedure Call*) [Bir84] e PVM (*Parallel Virtual*

[3] Todas as funções *Unsafe* do Solaris são explicitamente documentadas pela *Sun*. Aquelas em que não há identificação explícita são definidas como *Safe*.

Machine) [Beg94]. O CORBA é outro padrão que tem sido muito utilizado, principalmente na integração de ambientes.

RPC

O mecanismo RPC (*Remote Procedure Call*) simplifica o modelo de programação oferecendo transações de requisição-resposta (*request-response*) eficientes, estando disponível em quase todos os sistemas como parte do *software* de rede [Bir84]. Ao combiná-lo com o modelo de *threads* é possível explorar múltiplos processadores com memória não-compartilhada e distribuir uma aplicação tratando *workstations* como um multiprocessador.

Similar a uma chamada de função local, a operação RPC é síncrona, isto é, o processo cliente é bloqueado até que o processamento do servidor termine. Entretanto, isto pode não ser aceitável para muitas aplicações tais como as interativas ou em tempo real. Para tornar a operação assíncrona, o cliente inicia uma chamada RPC internamente a uma *thread* e prossegue com outros processamentos, deixando a *thread* esperar pelo retorno da chamada remota.

No sistema operacional distribuído Amoeba [Tan81, Mul90], a primitiva de comunicação na rede é baseada no mecanismo de RPC. Um processo (ou *cluster*) consiste de um ou mais *lightweight processes* (chamados *tasks*), que compartilham um espaço de endereçamento comum e executam em paralelo. [Bal87] descreve como a combinação de RPC e *tasks* do Amoeba pode ser usada para implementar algoritmos distribuídos. Um deles é a implementação da técnica de *branch-and-bound* [Law66] para resolver o problema do caixeiro viajante (procura da menor rota para visitar todos os nós de um grafo), usando pesquisa paralela. Os resultados das experiências de [Bal87] indicam que a combinação das primitivas de RPC e *tasks* do Amoeba fornece semântica simples e exploração eficiente do paralelismo.

Há certas circunstâncias nas quais o RPC parece não ser o paradigma de comunicação adequado. Isto corresponde a situações em que soluções baseadas em *multicast* ou *broadcast* parecem ser mais apropriadas [Bir84]. Não é conveniente também para a transferência de grande quantidade de dados ou para a comunicação sobre um meio de alta latência. Para esses casos são preferidos protocolos especiais para transferência de dados [Coo87].

PVM (*Parallel Virtual Machine*)

O PVM, um mecanismo baseado na passagem de mensagens, contém uma biblioteca de rotinas utilizáveis pelo usuário para passagem de mensagens, criação de processos, coordenação de tarefas, modificação da máquina virtual e comunicação de grupo (*multicast*). As rotinas são como comandos de linguagens de programação e fornecem um nível de

abstração bem maior que o RPC, o qual exige mais envolvimento do usuário convencional com os conceitos e detalhes da computação cliente/servidor.

O código do PVM ainda não é *multithreaded*, embora possa executar operações concorrentemente. As justificativas para a falta de suporte a *threads* são as grandes alterações no código fonte e a não disponibilidade de *threads* em todos os sistemas.

Outra consideração importante é que os *drivers* do protocolo PVM executam como processos normais (*pvmd* e tarefas)[4] e, naturalmente, esta estratégia degrada o desempenho da passagem de mensagens. Existe um custo na leitura de *timers* e gerenciamento de memória feitos a partir do ambiente de usuário (*user space*), enquanto o chaveamento extra de contexto e operações de cópia são expostos. O desempenho poderia ser melhorado se o código fosse integrado ao *kernel* [Gei94a].

No MakeD foi usado o RPC pela sua eficiência e capacidade *multithreaded*, característica importante para a construção do servidor de processamento. Além disso, não foram usados mecanismos de comunicação em grupo.

CORBA

O padrão CORBA (*Common Object Request Broker*) [OMG97] especifica mecanismos utilizados para a comunicação transparente entre componentes (objetos) de uma aplicação distribuída envolvendo diferentes plataformas, sistemas e linguagens.

Os objetos distribuídos pela rede possuem interfaces bem definidas e se comunicam através de um ORB (*Object Request Broker*) – componente de *software* que provê serviços para a interação transparente entre objetos clientes e servidores. A linguagem IDL (*Interface Definition Language*), também especificada pelo CORBA, possibilita a comunicação entre objetos independente da linguagem em que estão implementados, provendo separação entre interface e implementação. O protocolo de comunicação entre ORB's, IIOP (*Internet Inter Orb Protocol*), possibilita a comunicação entre objetos de ORB's distintos através da Internet.

Um cliente solicita a execução de um serviço através de uma requisição; a função do ORB é encontrar o objeto que deve responder a uma requisição e transportar os dados envolvidos. A maneira como os clientes e servidores interagem através do ORB é independente da sua localização e da linguagem de programação em que estão implementados.

Dessa forma, componentes modelados pelo CORBA são ideais para o desenvolvimento de aplicações distribuídas cliente/servidor flexíveis e podem facilitar o uso do MakeD em diversas plataformas. Além do mecanismo de comunicação, o CORBA fornece um ambiente de programação baseando em objetos de mais alto nível comparado ao RPC e maior flexibilidade para a integração do MakeD com outras aplicações. O RPC, apesar de

[4] No PVM existe comunicação entre *pvmd-pvmd* ou entre *pvmd*-tarefa. Além disso, um *daemon pvmd* serve a um único usuário e não há interação entre *daemons* de usuários diferentes.

disponível na maioria dos sistemas, apresenta variações dependentes de cada sistema operacional que acabam dificultando a portabilidade de aplicações distribuídas.

O MakeD pode ser facilmente portado para uma implementação do CORBA, como por exemplo o Orbix [IONA96], o qual permite a criação de servidores e clientes *multithreaded* através do componente Orbix-MT, que fornece todas as funções *thread-safe*. Como o MakeD é implementado em linguagem C e a IDL do Orbix gera código em C++, é necessário mapear as funções (em C) do controlador *make* que fazem a interface com os *stubs* (objetos C++) gerados pela IDL do CORBA.

6.5 Distribuição de tarefas

Os avanços tecnológicos têm proporcionado grande aumento na produção de computadores com alta capacidade de processamento e de armazenamento. Atualmente, as estações clientes são tão potentes quanto os servidores, sendo a diferença da capacidade de processamento não mais exorbitante como no passado. Com isso uma distribuição balanceada de tarefas é ideal para maximizar a utilização dos recursos da rede sem sobrecarga de alguns equipamentos. Entretanto, o método de distribuição de tarefas depende da maneira particular como a aplicação distribuída trabalha.

Para uma aplicação cliente/servidor como o MakeD, a escolha da estação gerente (com o controlador *make*) é importante não só por realizar as tarefas de construir as dependências e coordenar a distribuição de tarefas, mas também por possuir a funcionalidade de servidor de processamento embutida no cliente. Nesse caso, a estação cliente é bastante exigida, principalmente se mais de um usuário usar a aplicação ao mesmo tempo ou se a estação agir também como um servidor de processamento de requisições remotas (*daemon* servidor ativo).

A falta de atualização do cliente com relação à carga dos SPs faz com que estes trabalhem como escravos de clientes no MakeD. Mesmo que um SP esteja sobrecarregado ele pode continuar recebendo pedidos de compilações e colocando-as na fila, com a implicação da demora na resposta. Um método alternativo para manter o cliente atualizado ao longo do processo de *make* é a inclusão da carga do servidor na resposta de cada chamada RPC. Entretanto, é necessário selecionar os parâmetros que serão considerados na medida da carga da estação, e combiná-los numa fórmula para a tomada de decisão no cliente. Tal método é importante, pois aplicações distribuídas têm se tornado cada vez mais freqüentes e o MakeD pode estar competindo por recursos comuns a outras aplicações distribuídas ou não-distribuídas.

Parâmetros importantes para a decisão na distribuição da aplicação incluem: velocidade relativa do *hardware*, carga média da CPU, memória e espaço de *swap* disponíveis e necessários pela aplicação, número de CPUs, processos na fila de execução, presença/ausência de usuários interativos, etc.

A reconfiguração dinâmica do limite de tarefas simultâneas no servidor é outra alternativa para adequar sua quantidade de trabalho à carga da estação. A reconfiguração pode acontecer quando forem atingidos patamares mínimos e máximo definidos para a carga de CPU, número de processos em execução e número de usuários "logados". O limite de tarefas concorrentes pode ser aumentado ou reduzido caso a estação esteja ociosa ou sobrecarregada, respectivamente.

Pode-se considerar também a possibilidade de migração de processos entre SPs, objetivando balancear a carga de trabalho entre as estações. Neste caso, os SPs devem trocar informações periódicas e implementar um política de migração de processos (geralmente dependente do sistema) que envolve questões como gerenciamento de memória, comunicação e *overhead*.

6.6 Definição do valor de *timeout*

O *timeout* é útil para a tomada de decisões quando uma requisição não recebe resposta após o tempo de espera. É difícil saber a causa do problema. O serviço pode estar ainda em processamento ou haver problemas com a requisição, servidor, resposta, rede, etc.

O tempo de execução de tarefas remotas é variável, tanto em função da capacidade da estação quanto pela sua carga de trabalho num dado instante. Numa aplicação como o *make*, os comandos são heterogêneos e conseqüentemente os tempos também podem ser muito diferentes.

Dessa forma, é muito difícil para o usuário conseguir uma definição eficiente para o *timeout*, que exige uma estimativa prévia do tempo gasto para construir cada *target* (soma dos tempos de execução das tarefas que o geram). Uma possível solução seria adicionar um recurso para permitir a definição do *timeout* por regra, dentro do `makefile`. Isto contribui principalmente quando as chamadas recursivas do MakeD são realizadas por um SP, pois nesse caso o valor do *timeout* deve prever a execução remota de uma instância do MakeD. Por outro lado, é possível analisar as regras para verificar a presença de recursividade e sempre forçar a execução recursiva do MakeD através da estação local, evitando assim possíveis problemas com a especificação do *timeout* para este caso, pois exige valor bastante alto comparado às requisições simples, causando espera inútil em caso de falha.

Muitas outras aplicações distribuídas possuem problemas de *timeout* semelhante ao MakeD.

6.7 Servidor de *make* distribuído multi-usuário

O MakeD acelera a atualização dos *targets* aproveitando eficientemente os recursos de rede (com os SPs) ou multiprocessamento (com *threads*), mas há possibilidades de melhorias:

- Embora os SPs sejam multi-tarefa e multi-usuário, o MakeD (tal como a versão convencional) não suporta bem requisições simultâneas de vários usuários para o mesmo `makefile`. As diferentes chamadas do MakeD não compartilham informações, e conseqüentemente pode ocorrer que um projeto não seja atualizado da melhor forma.
- Para grandes projetos, é gasto esforço considerável pela aplicação *make* para análise do `makefile`, montagem do grafo e verificação de dependências. Armazenar essas informações de modo persistente, assim como organizar e disciplinar o acesso às mesmas, poderia acelerar significativamente a geração dos *targets* e eliminar esforços duplicados.
- Um servidor com informações persistentes poderia ajustar automaticamente parâmetros como nível de paralelismo por máquina e *timeout* de término de comando, dispensando ajustes pelo usuário.

Uma possível evolução para o MakeD, no sentido de diminuir esforços gastos na manutenção e gerência de projetos envolvendo múltiplos usuários, é a sua transformação num Servidor de *Make* Distribuído Multi-usuário (SMakeDM), responsável por cadastrar, gerenciar e manter um número arbitrário de projetos.

Numa versão mais simples, o SMakeDM teria que suportar vários usuários possivelmente distribuídos, e cada um deles, mesmo se alocado no mesmo projeto, teria sua própria cópia do `makefile`. Ao estabelecer uma sessão (processo de *login*) no SMakeDM e informar o projeto, seria disponibilizado uma cópia do `makefile` ao usuário, estando a partir daí apto a realizar atualizações de *targets*. Na requisição do serviço de *logout*, a memória utilizada pelo grafo correspondente ao `makefile` seria liberada.

A geração do grafo a partir do `makefile` e a busca não sofrerão alterações com relação ao controlador *make*. Por outro lado, a distribuição de tarefas para os SPs deve ser refeita de modo a abranger as requisições de todos os usuários com sessões no SMakeDM solicitando compilações. Os SPs serão os mesmos usados no MakeD.

A figura 6.1 ilustra um SMakeDM simplificado com os métodos de *login*, *make* e *logout* e a configuração de dois projetos (`makefile` e grafo de dependências), que será fornecida aos usuários que estabelecerem sessões nos mesmos.

O SMakeDM simplificado resolve o problema de interpretação do `makefile` e montagem do grafo a cada processo de *make* realizado pelo usuário. Entretanto, não há compartilhamento de informações entre os usuários, implicando em muito trabalho duplicado e atualização do projeto de forma não otimizada.

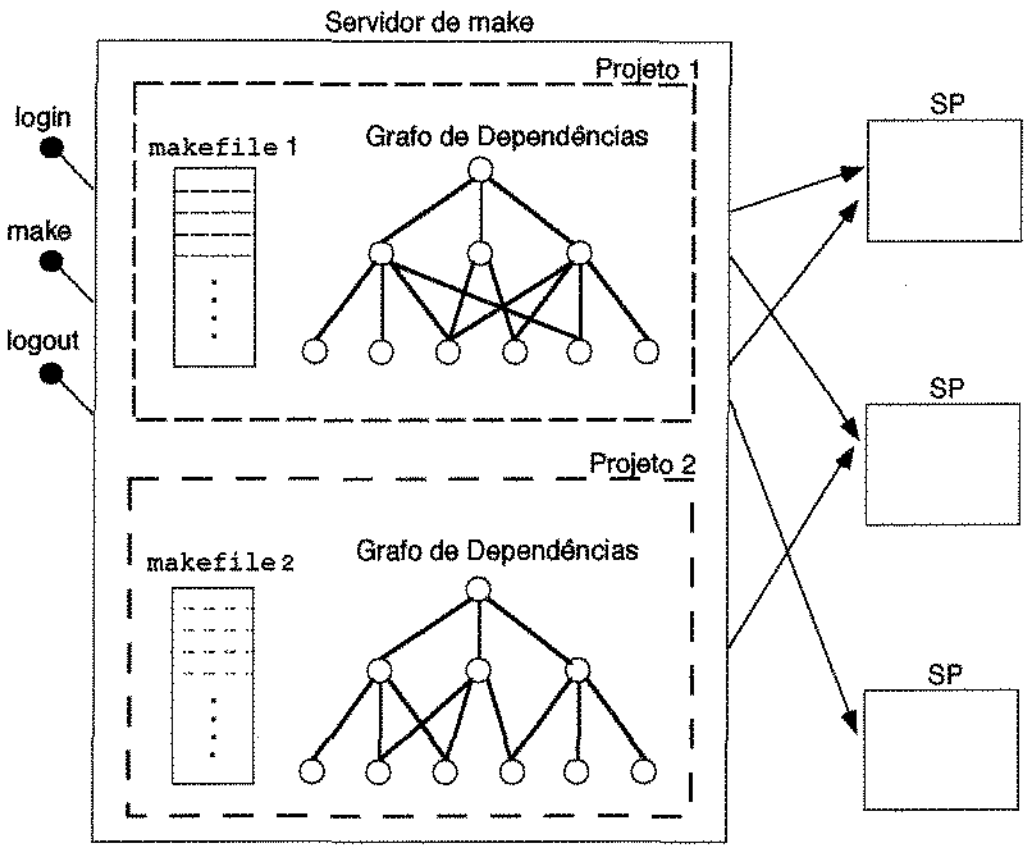


Figura 6.1: SMakeDM simplificado

Para uma versão mais elaborada do SMakeDM, projetos seriam definidos por `makefiles` e o grafo montado a partir de um `makefile` deveria ser compartilhado por todos os usuários participantes do projeto. Entretanto, problemas interessantes ocorrem com usuários simultâneos no mesmo projeto. O grafo de dependências poderia ser criado apenas uma vez pelo primeiro usuário e compartilhado pelas demais sessões? Isso não é possível com as estruturas originais do GNU *Make*, pois o grafo e outras estruturas são alteradas durante a invocação dos comandos, refletindo a atualização dos *targets*.

Por outro lado, várias estruturas (ou campos de estruturas) são apenas lidas e podem ser compartilhadas sem problemas. No caso geral, as estruturas terão alguns campos de leitura e outros que poderão ser alterados. Qualquer adaptação no sentido de permitir o compartilhamento dessas estruturas implica na reescrita de boa parte do programa original centralizado e mono-usuário.

A figura 6.2 apresenta um SMakeDM no qual dois usuários do projeto compartilham o grafo nos seus processos de *make*. Ainda na figura, caso outro usuário tentasse atualizar o mesmo projeto por completo, ele compartilharia o grafo com os usuários 1 e 2 e seria beneficiado pelo trabalho já feito pelos *makes* destes usuários.

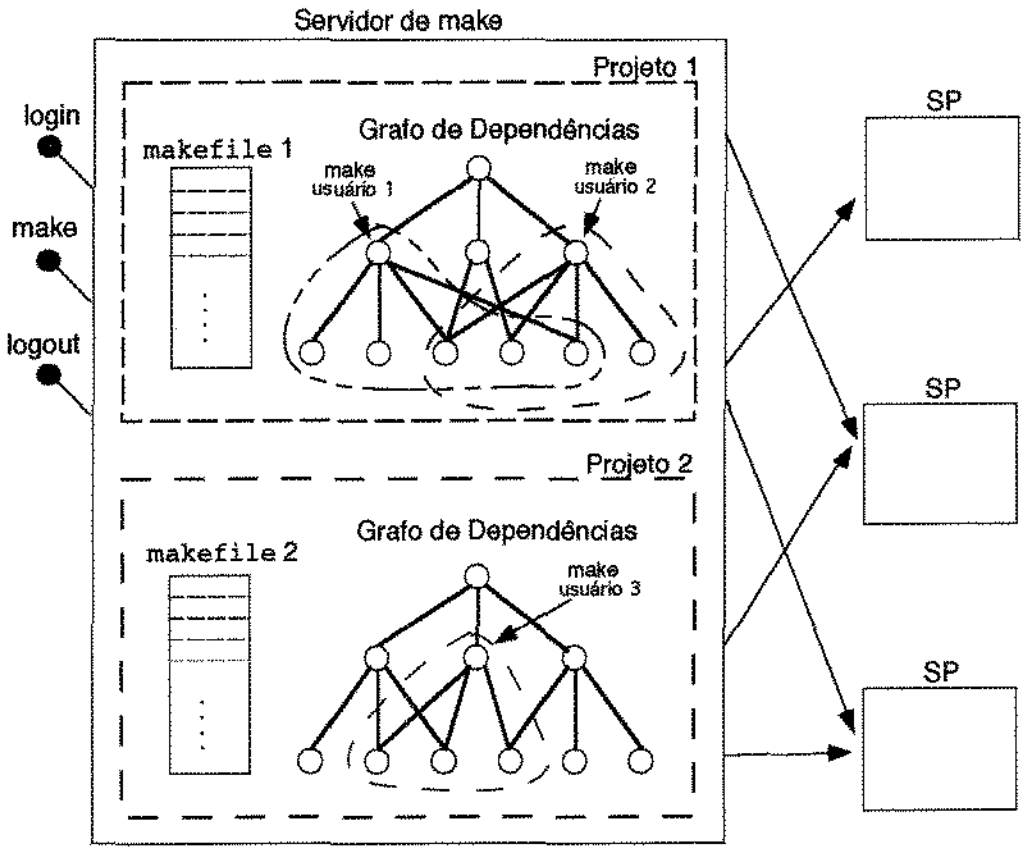


Figura 6.2: SMakeDM compartilhando o grafo entre usuários do mesmo projeto

A proposta de transformação do MakeD num servidor de *make* distribuído multi-usuário é ainda muito geral. Ela necessita ser refinada para compreender melhor os possíveis problemas que podem ocorrer, além da avaliação do *overhead* causado pela gerência de projetos e da sincronização necessária para acesso às estruturas dos grafos.

Com um estudo mais completo do problema, pode-se conseguir uma ferramenta prática, principalmente para ambiente Windows NT/95, e com várias utilidades como o MakeD e o *make* convencional. Além disso, o problema pode auxiliar no estudo de vários outros de natureza distribuída e multi-usuário.

Capítulo 7

ANÁLISE DE DESEMPENHO DO MAKED

Neste capítulo é apresentado uma análise de desempenho da ferramenta MakeD em relação ao GNU *Make* original e ao Dmake. Para utilizar este último foi necessário realizar alterações no seu código fonte no sentido de portá-lo para a execução no ambiente Solaris 2.5. A exclusão do Lmake nos testes é devido ao fato desta aplicação executar somente em sistemas SunOS[1], além de não possuir a alternativa de execução de múltiplos *jobs* simultâneos (multi-tarefa) em cada sistema, o que limita a comparação com os demais *makes*. No entanto, o Lmake foi encontrado na fase final deste trabalho e a falta de documentação dificultou seu entendimento e porte para o ambiente Solaris.

É importante verificar o desempenho do MakeD em relação a outra aplicação semelhante, como o Dmake, para analisar os possíveis ganhos e a eficiência dos métodos empregados no desenvolvimento de uma nova aplicação.

O ambiente de testes utilizado é o tópico da Seção 7.1. A análise dos resultados de compilações centralizadas e distribuídas, usando os *makes* citados acima é o assunto das duas próximas seções. Por fim são colocadas algumas considerações sobre os testes.

7.1 Ambiente de testes

Para a realização dos testes foram usadas estações de trabalho *Sun SPARCstation* 4[2] presentes numa mesma rede do Instituto de Computação (IC) da Unicamp e todos os arquivos localizados remotamente, acessíveis através de um servidor de arquivos situado na rede principal do Instituto.

[1] SunOS 4.1.3 e SunOS 4.1.4

[2] Configuração das estações: 1 CPU de 110 Mhz, 32 Mbytes de RAM e sistema operacional Solaris 2.5 (SunOS 5.5).

Nas compilações foram utilizados os próprios arquivos do GNU *Make* (30 no total) e foram criadas cinco cópias de cada um deles (`job1.c` até `job5.c`, por exemplo), perfazendo 150 arquivos a serem compilados, dispostos numa relação de dependências (arquivo `makefile`) semelhante à utilizada pelo *make* original. Esta quantidade de arquivos utilizada nos testes é uma tentativa de simular a construção de um sistema em que o tempo gasto na geração do *target* final seja da ordem de dezenas de minutos.

As medições de tempo foram todas realizadas com a ajuda do comando `/usr/bin/time`. Este recebe um outro comando como argumento e oferece após o término deste último um sumário dos tempos de execução:

- o "real" correspondente a um relógio convencional (*elapsed time*);
- o de uso do processador pelo comando (*user time*);
- o de uso do processador pelo sistema (*system time*).

Para efeito de análise, o tempo real é o de maior relevância por refletir a demora em obter o resultado desejado pelo usuário. Entretanto, é sujeito a variações devido a carga do sistema e concorrência pelo uso do processador por vários processos (além de ser dificilmente aplicável a exemplos interativos). Por outro lado, o comando `time` não contabiliza o tempo de sistema (*system time*) e tempo de usuário (*user time*) de processos remotos e de processos não invocados pelo processo contabilizado (como servidores).

7.2 Compilações centralizadas

O conjunto de arquivos citado na seção anterior foi compilado várias vezes numa estação utilizando o GNU *Make*, MakeD e Dmake, ambos configurados para usar um número diferente de tarefas simultâneas em cada processo de *make*. Os tempos reais (em minutos) de cada processo de *make* são apresentados na Tabela 7.1 segundo o número de tarefas concorrentes (J). Estes e os demais tempos (usuário e sistema) são colocados no Apêndice B, para maiores detalhes.

J	GNUMake	MakeD	Dmake
1	10:42.9	10:46.3	14:50.8
2	09:41.8	09:49.6	11:29.3
4	08:12.8	08:33.1	10:21.8
6	07:45.1	08:00.1	10:21.4
8	07:40.6	07:57.4	10:28.8
10	07:41.7	07:52.7	10:32.6
12	07:44.5	08:00.2	10:38.1
16	07:42.3	07:59.3	10:48.5

Tabela 7.1: Tempos reais de processos de *make* numa estação

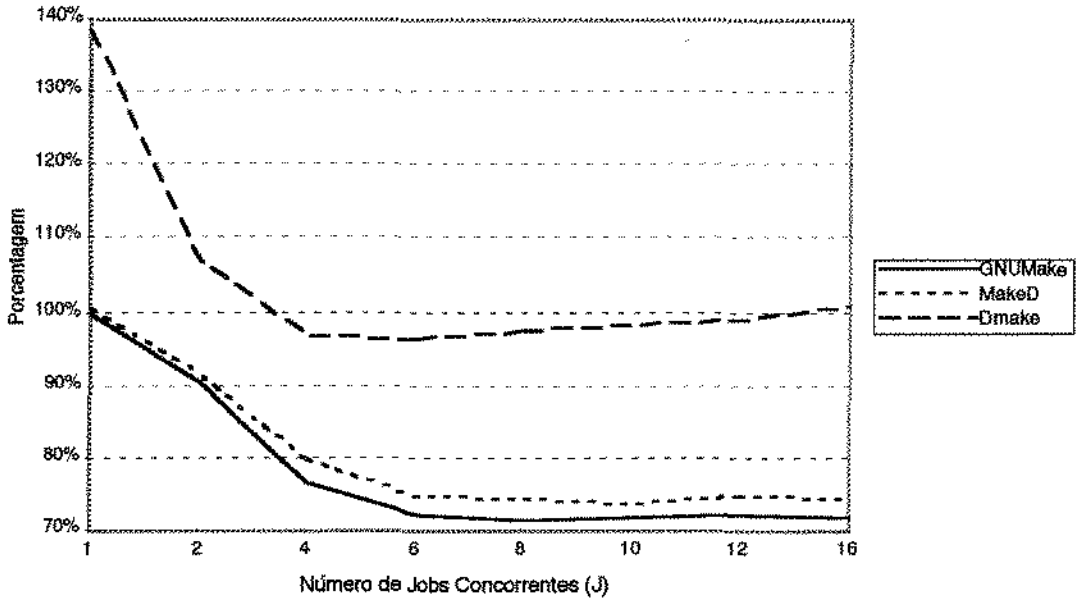


Gráfico 7.1: Porcentagem relativa ao GNU Make com J=1

É de se esperar que o GNU *Make* seja mais rápido para todos os valores de J devido ao *overhead* adicional dos *makes* distribuídos. O pior desempenho do Dmake também era esperado pelo fato de usar a comunicação Cliente/Servidor (*spew*, *rspew*) via `rsh` dentro da própria estação. São necessários cinco processos adicionais (*spew*, *feeder*, `rsh`, `rshd` e *rspew*) para a execução de cada comando devido à uniformidade no tratamento de execuções locais e remotas. No caso do MakeD, o *overhead* é causado principalmente pelo uso de *threads* (uma por nó) para gerenciar a execução dos comandos que geram um *target*.

Observa-se que o uso da concorrência é vantajoso para todos os *makes* e sempre fornece resultados melhores que em processos de *make* com compilações seqüenciais. Entretanto, conseguem-se ganhos muito pequenos com mais de seis tarefas concorrentes no GNU *Make* e MakeD e mais de quatro no Dmake. Acima desses valores, a relação do possível benefício versus degradação do sistema torna-se inviável sob as configurações do ambiente de testes. Nota-se também que um alto grau de concorrência como, por exemplo, o uso de J=16, acarreta grande sobrecarga no sistema.

O gráfico 7.1 mostra que o desempenho do Dmake rapidamente deixa de melhorar para valores crescentes de J. No entanto, com multi-tarefa local em quatro *jobs* simultâneos, o Dmake reduz seu tempo em 42%, e o GNU *Make* e MakeD reduzem 23% e 21%, respectivamente (veja os percentuais no Apêndice B). Além desse limite, o Dmake não consegue mais tirar proveito com o aumento do grau de concorrência.

Verifica-se no Apêndice B que os tempos de sistema do MakeD são ligeiramente maiores que os do GNU *Make*, ocorrendo o contrário com os tempos de usuário. Este comportamento é certamente o efeito do uso de *threads* no MakeD. No Solaris, as *threads*

são em dois níveis: a nível de usuário (bibliotecas) e a nível de *kernel*[3]. As *threads* a nível de usuário são multiplexadas em *threads* de controle suportadas pelo *kernel*, as quais requerem recursos de sistema para fornecer o suporte ao paralelismo neste nível, sendo relativamente bem mais "caras".

Ainda no Apêndice B, os tempos de sistema e usuário do Dmake são bastante pequenos comparados ao MakeD e GNU *Make*. Esta vantagem é ilusória devido ao uso da separação cliente/servidor com o `rsh` dentro da estação. Em termos de contagem de tempo de sistema e usuário é como se toda a compilação fosse remota e portanto não contabilizada por `time`.

7.3 Compilações distribuídas

O mesmo conjunto de arquivos dos testes locais foi empregado nos processos de *make* distribuídos usando o MakeD e Dmake. Foram utilizados grupos variando de dois a seis estações; para cada grupo foram realizados testes usando um e mais processos simultâneos em cada estação. Os resultados dos tempos reais (em minutos) são mostrados na Tabela 7.2[4]. Estes e os demais tempos são apresentados no Apêndice C, juntamente com os percentuais calculados para gerar todos os gráficos seguintes.

J	GNUMake	MakeD1	MakeD2	MakeD3	MakeD4	MakeD5	MakeD6
1	10:42.9	10:46.3	06:45.0	05:29.1	03:58.3	03:34.8	03:18.1
2	09:41.8	09:49.6	05:49.6	04:38.7	03:16.1	02:56.0	02:30.5
4	08:12.8	08:33.1	05:13.4	03:39.6	02:54.9	02:42.3	02:07.9
6	07:45.1	08:00.1	04:33.7	03:48.7	02:53.8	02:30.1	02:04.7
8	07:40.6	07:57.4	05:04.8	03:57.5	02:47.4	02:39.9	02:07.5
10	07:41.7	07:52.7	05:03.9	03:33.0	02:48.8	02:15.0	02:19.9
J	**GNUMake**	**Dmake1**	**Dmake2**	**Dmake3**	**Dmake4**	**Dmake5**	**Dmake6**
1	10:42.9	14:50.8	11:22.8	05:50.8	05:26.2	04:10.9	03:58.5
2	09:41.8	11:29.3	08:31.8	04:42.6	04:32.3	03:30.9	02:58.7
4	08:12.8	10:21.8	07:07.5	04:21.7	04:05.9	02:59.1	02:52.4
6	07:45.1	10:21.4	08:07.7	04:29.3	03:32.5	03:10.8	03:13.2
8	07:40.6	10:28.8	06:06.6	04:35.8	03:31.5	03:23.4	02:52.3
10	07:41.7	10:32.6	05:45.6	04:37.1	03:14.8	03:11.0	02:55.6

Tabela 7.2: Tempos reais de processos de *make* distribuídos[4].

[3] *Threads* a nível de usuário são representadas por estruturas de dados no espaço de endereçamento de um programa. *Threads* a nível de *kernel*, chamadas *Lightweight Processes* (LWP), são alocadas por processo e compartilham seu espaço de endereçamento. Cada LWP pode ser interpretada como uma CPU virtual que está disponível para execução de código e *system calls*.

[4] O número junto ao nome do *make* indica a quantidade de máquinas utilizadas no processo de *make*, e J as tarefas concorrentes.

Similar às compilações locais, para cada grupo de estações o pior caso é quando não há concorrência e os melhores valores para a quantidade de tarefas simultâneas em cada estação de um dado grupo continuam sendo quatro para o Dmake, e seis para o MakeD. Acima desses valores, o possível benefício também é bastante pequeno ou inexistente.

Compilações fazem uso relativamente intenso do sistema de arquivos e não dependem apenas do processamento na CPU. As leituras/escritas em disco competem por um único servidor de arquivos central e todo o tráfego passa por uma rede *Ethernet* de 10 Mbps compartilhada. Foi observado que dezesseis tarefas concorrentes em cada uma do grupo de seis estações geram alto tráfego na rede, sobrecarregam os sistemas individuais e mantêm grande quantidade de arquivos abertos para leitura/escritas no servidor de arquivos, dificultando a utilização da rede por outras aplicações.

Para ambos os *makes*, o desempenho é melhor quando as compilações são distribuídas num número maior de estações ao invés de aumentar a quantidade de execuções paralelas por estação. Por exemplo, é mais vantajoso executar os módulos em quatro estações ao invés de executá-los em duas, com dois módulos simultâneos em cada.

Os gráficos 7.2 e 7.3 (percentuais no Apêndice C) mostram que, somente na distribuição simples de tarefas para seis máquinas, ambos os *makes* reduzem em média 2/3 do tempo com relação ao *make* original. Ao se combinar a distribuição e multi-tarefa nas estações, conseguem-se ganhos de até 80% e 73% no MakeD e Dmake, respectivamente, usando J=8 em cada uma das seis estações. Em relação ao Dmake, o MakeD consegue um ganho mais uniforme com o crescimento de J e do número de estações. O Dmake tem sensível melhora quando empregado com duas e três estações, pois nesses casos o seu *overhead* é distribuído entre os sistemas, mas ele volta a ser significativo à medida que cresce a concorrência dentro das estações.

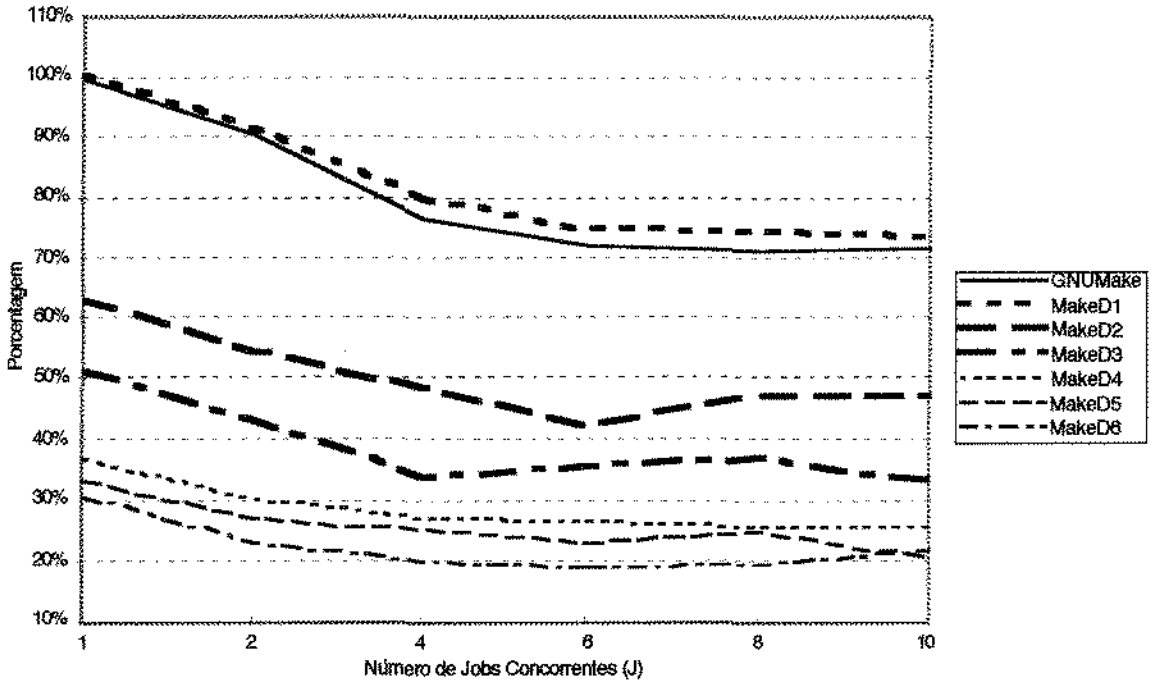


Gráfico 7.2: MakeD - Porcentagem relativa ao GNU Make com J=1

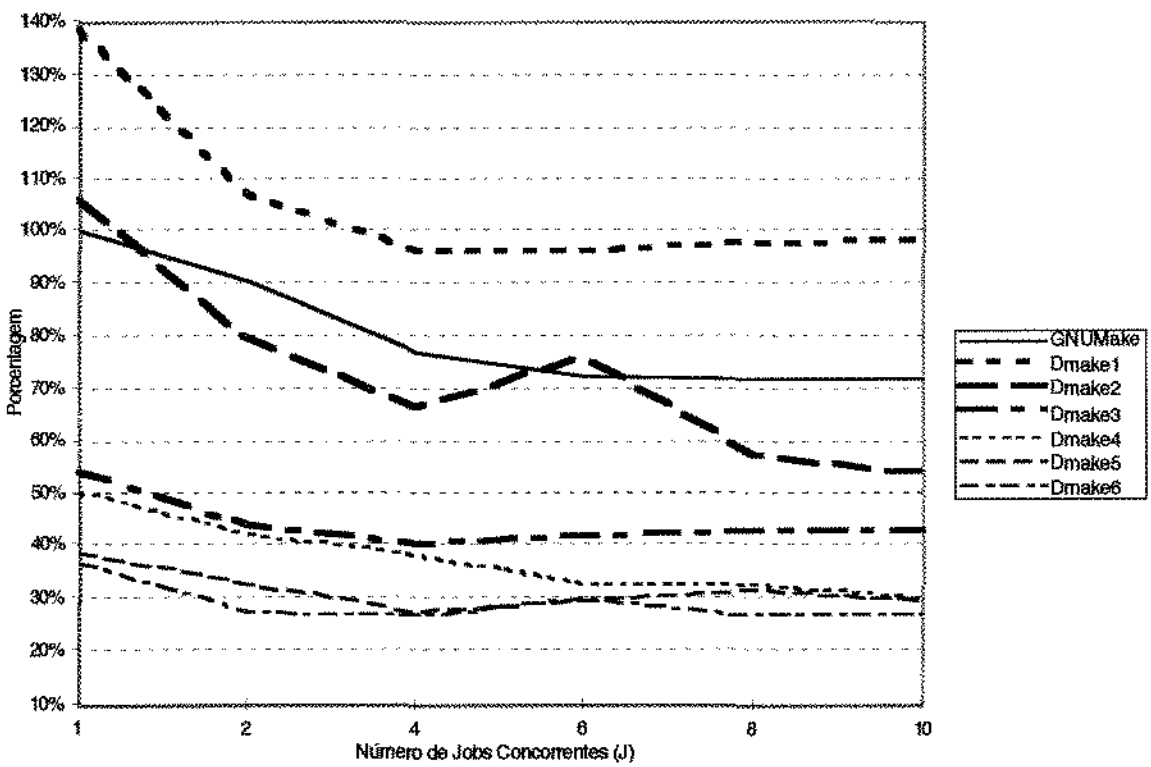


Gráfico 7.3: Dmake - Porcentagem relativa ao GNU Make com J=1

O próprio autor alerta sobre as limitações de distribuição do Dmake e recomenda utilizá-lo em poucas estações. O principal problema é a quantidade de processos adicionais necessários para a execução de comandos. Isto foi observado durante vários testes em que os recursos dos sistemas eram excedidos. Um exemplo é a compilação do conjunto de arquivos usados nos testes utilizando a opção de otimização do compilador. O Dmake não consegue fazê-lo em mais de quatro máquinas com J=6 em cada sem a devida reconfiguração do ambiente do sistema operacional, ampliando bastante o número de entradas na tabela de processos, a área de *swap*, etc.

Os resultados obtidos quando usados J=12 e J=16 não foram apresentados por não serem significativos, mas são colocados no Apêndice C. O ganho conseguido utilizando tais valores para a multi-tarefa não justifica a sobrecarga gerada em cada sistema participante do processo de *make*.

O gráfico 7.4 representa o tempo percentual do MakeD em relação ao Dmake, com mesmo J. Em todos os casos, o MakeD tem desempenho superior (percentuais abaixo de 100%).

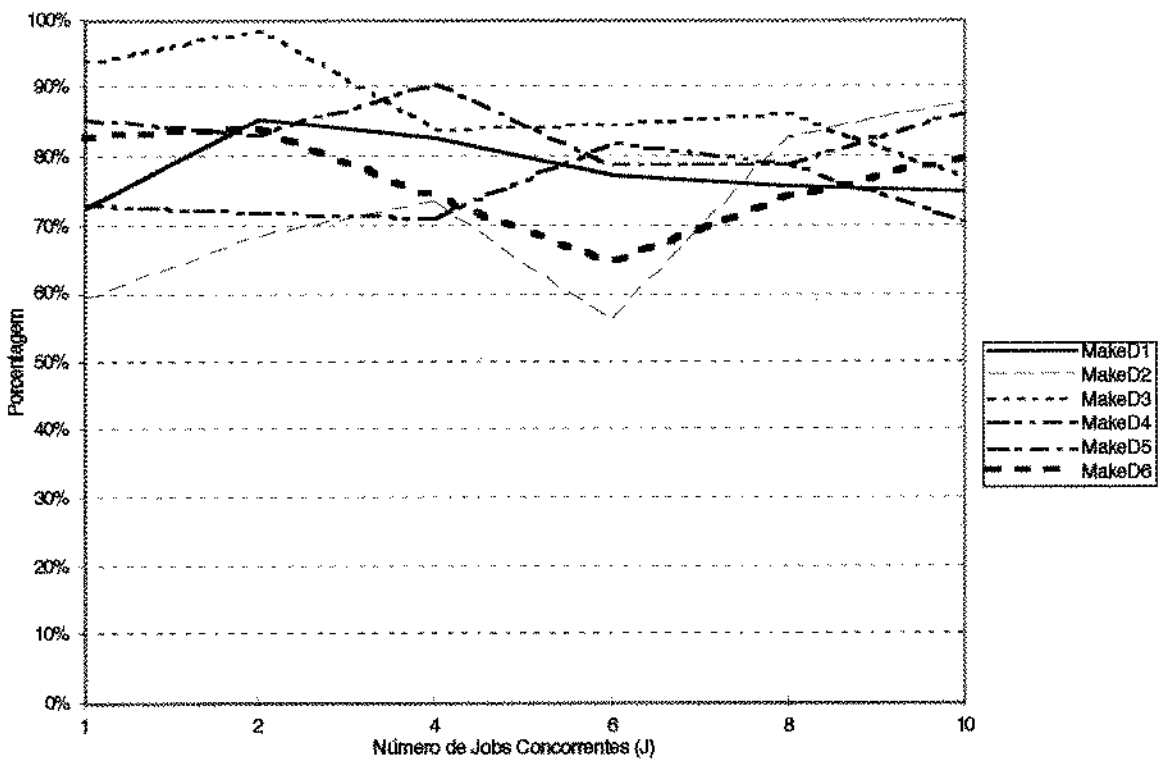


Gráfico 7.4: MakeD - Porcentagem relativa ao Dmake com mesmo J

O gráfico 7.5 representa o desempenho do MakeD (tempos em segundos) à medida que aumenta o número de estações. Cada curva indica a quantidade de tarefas simultâneas utilizadas por estação. Acima de seis tarefas simultâneas, as curvas estão muito próximas, confirmando o ganho quase desprezível para J superior a seis.

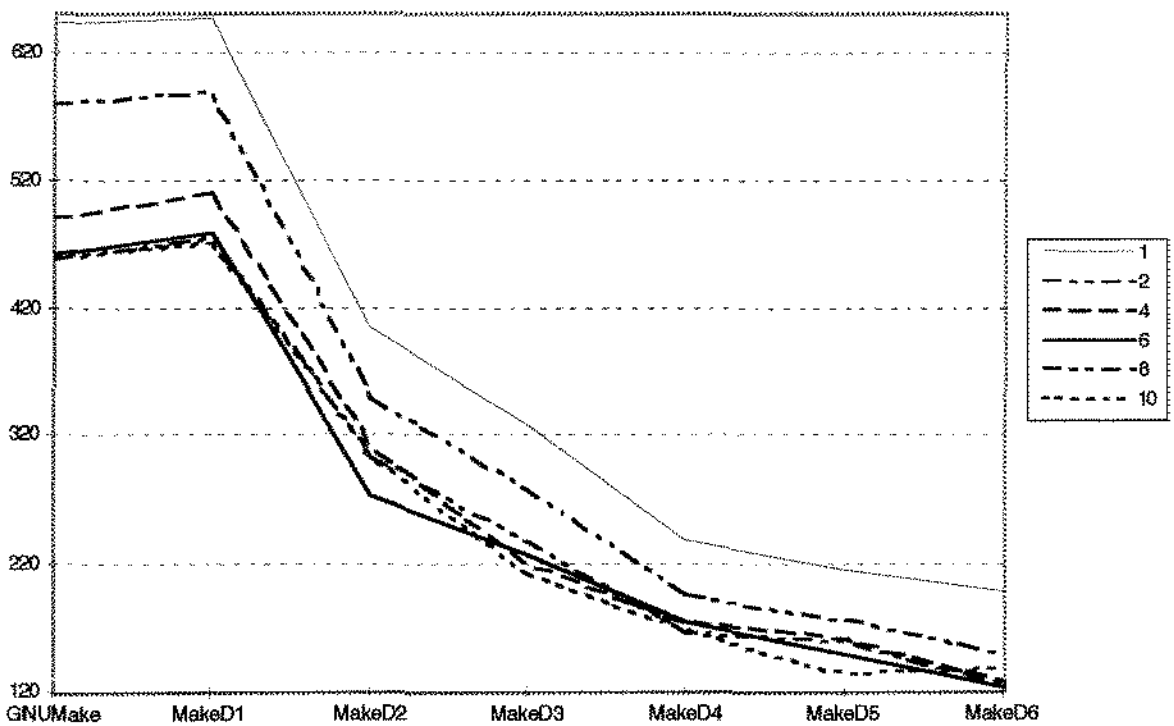


Gráfico 7.5: MakeD – Tempos de execução (segundos)

Considerando a relação entre a concorrência de várias tarefas num processador (gráfico 7.2) e a distribuição na rede (gráfico 7.5) – percentuais de ambos na tabela C.4, em geral a concorrência não resulta em melhora significativa. No melhor caso, enquanto um processo está fazendo I/O, outro está utilizando a CPU. Além disso, o tempo necessário para realizar *swapping* ou paginação aumenta com o crescimento do número de processos. O resultado é que os processos em execução simultânea gastam mais tempo para finalizar.

Os tempos de sistema e usuário (Apêndice C) do MakeD na estação com o controlador *make* são reduzidos à medida que a distribuição aumenta com o número de máquinas devido à divisão de tarefas entre os diferentes sistemas. No Dmake, estes tempos possuem pouca variação com a distribuição devido ao tratamento cliente/servidor com `rsh` a toda compilação e a inexistência de *threads* locais.

7.4 Considerações sobre os testes

O ganho de tempo do MakeD e Dmake em comparação ao GNU *Make* original depende principalmente do número de estações configuradas e do grau de dependência entre os *targets* presentes no `makefile`. Quanto maior a hierarquia de dependências, maior a profundidade do grafo e conseqüentemente a espera pela geração de *targets* intermediários. O número de máquinas pode aumentar até um limite a partir do qual o *overhead* de

gerenciamento, tráfego na rede, acessos ao servidor de arquivos, entre outros, impede melhorias de desempenho global.

Nota-se que pode ser analisado o comportamento do MakeD com a mudança de vários parâmetros. Por exemplo, seria interessante realizar testes alterando a relação de dependências de forma a variar a profundidade e a largura do grafo. Outras possibilidades seriam tornar a estação gerente (cliente) o servidor de arquivos ou o servidor de arquivos participar também como servidor de computações (SP ativo).

O controlador *make* (cliente) gerencia também *threads* responsáveis pela execução de comandos locais via processos. Assim, o controlador *make* compete pela CPU com seus processos filhos (gerados pelas *threads*) e com os demais processos do sistema. É possível aumentar a prioridade do controlador *make* em relação aos seus processos, enfatizando a distribuição de tarefas e processamento das respostas de compilações locais e remotas. Entretando, com testes, não foi verificado melhoria no desempenho global do MakeD quando a prioridade do controlador *make* é superior à de seus processos em uma ou duas unidades.

Características de rede como tráfego, localização do servidor de arquivos e a sua freqüência de utilização são fatores importantes que influem nos tempos de execução. O servidor utilizado nos testes apresentados fornece serviços a várias outras redes. Dessa forma, as compilações foram repetidas várias vezes em períodos de baixa utilização da rede, mostrando resultados similares.

Com o MakeD foi possível mostrar o ganho real de tempo distribuindo as tarefas entre os diversos sistemas na rede. No entanto, o desempenho foi medido de maneira egoística, ou seja, os recursos foram considerados como sendo utilizados somente por esta aplicação. É preciso como próximo passo implementar um método de distribuição de tarefas baseado no conhecimento recente da carga dos sistemas. Assim, além de explorar os recursos de maneira otimizada, pode-se verificar o comportamento do MakeD num ambiente real em que existe concorrência com outras aplicações (distribuídas ou não-distribuídas) ativas nos diversos sistemas com capacidades de processamento heterogêneas.

Portar o Lmake para o ambiente Solaris 2.5 e alterar o seu esquema de execução de comandos para suportar multi-tarefa em cada sistema, tornaria possível verificar o desempenho de duas aplicações que utilizam diferentes mecanismos de comunicação, RPC no MakeD e PVM no Lmake. Dessa forma, seria possível tirar conclusões sobre a diferença de desempenho devido a esses mecanismos de comunicação.

Capítulo 8

CONCLUSÕES

Uma das principais propostas deste trabalho é mostrar a possibilidade de transformar a aplicação centralizada *make* numa ferramenta distribuída que permite a utilização de uma rede de computadores como um recurso computacional único. A viabilidade desta transformação foi mostrada com a implementação da aplicação MakeD, atualmente em utilização no Laboratório A-HAND (Instituto de Computação – Unicamp). Os resultados obtidos comprovam que o processo de *make* pode ser realizado com maior rapidez quando utilizada a nova aplicação em períodos de baixa utilização da rede.

A transformação de aplicações centralizadas em distribuídas ganhará espaço à medida que as ferramentas de suporte à programação distribuída simplifiquem a construção de aplicações eficientes e portáveis. A disponibilidade de redes de alta velocidade e o alto poder de processamento dos recursos agregados permitem melhorar o desempenho de aplicações seqüenciais, mas a análise de custo/benefício pode requerer simulações e até mesmo definir a paralelização de somente algumas porções do código visando equilibrar o esforço de implementação com as necessidades de melhoria no desempenho global. O MakeD mostra que é possível migrar uma aplicação para um ambiente distribuído usando o que está disponível no UNIX, apesar das dificuldades ainda existentes, principalmente com a utilização dos recursos de multi-tarefa (*multithreading*). Por outro lado, no MakeD são paralelizadas somente as tarefas de processamento de nós para a geração de *targets*, pois são elas as grandes consumidoras de tempo e processamento em relação ao trabalho global do *make*.

O MakeD é uma das aplicações que pode se beneficiar da ociosidade das estações da rede, principalmente das clientes, pois num ambiente cliente/servidor típico existem poucos servidores (alta capacidade de processamento) e vários clientes (pequena/média e alguns com alta capacidade de processamento). Distribuindo o processamento, o MakeD pode

maximizar o aproveitamento da rede utilizando também os clientes normalmente ociosos ou com baixa utilização. Dessa forma, há melhoria na relação custo/benefício em relação ao uso de um computador mais rápido e mais custoso.

Os experimentos envolvendo o GNU *Make*, MakeD e Dmake demonstram a viabilidade da utilização prática e as vantagens que a distribuição da aplicação *make* pode oferecer aos usuários. A combinação do mecanismos de RPC com recursos de *multithreading* torna o MakeD mais eficiente que o Dmake. Quando combinadas a distribuição de tarefas na rede e a multi-tarefa em cada sistema, o MakeD mostra maior escalabilidade e melhor utilização dos recursos computacionais comparado ao Dmake. Para ambos os *makes*, o desempenho é melhor quando as compilações são distribuídas para um número maior de estações ao invés de aumentar a multi-tarefa em cada sistema.

8.1 Trabalhos Futuros

A ferramenta desenvolvida oferece nítida utilidade prática e melhoria no desempenho. No entanto, a atual implementação é baseada no ambiente UNIX (Solaris 2.5) e segue o padrão de *threads* do Solaris, além de não considerar o balanceamento de carga na distribuição do processamento. Dessa forma, possíveis evoluções no sentido de enriquecer a aplicação e ampliar a sua utilização são:

- **Portabilidade para o ambiente Windows:** as plataformas Windows NT e Windows 95 são muito populares em rede local e as estações clientes devem possuir alto desempenho para que possam executar adequadamente aplicativos comuns como planilhas, *browsers*, jogos, etc. Dessa forma, o MakeD pode utilizar os computadores pessoais da rede para agilizar o processo de *make* de forma transparente ao usuário.
- **Portabilidade do MakeD para suportar POSIX *threads*:** uma das vantagens do suporte às *threads* do padrão POSIX é oferecer a possibilidade de se utilizar o MakeD em vários outros sistemas UNIX. Entretanto, o mecanismo de RPC é não-*Multithreaded Safe* em muitos UNIX (os stubs gerados para o cliente e servidor não são *MT-Safe*) e parcialmente *Multithreaded Safe* em alguns (os stubs do servidor não não *MT-Safe*). É preciso encontrar formas de resolver o problema para manter as capacidades multi-tarefa e multi-usuário nos SPs.
- **Implementação de balanceamento de carga:** o balanceamento dinâmico de carga para o MakeD pode contribuir principalmente por permitir a utilização de forma otimizada de estações com diferentes capacidades de processamento. Além disso, a constante atualização sobre a disponibilidade dos recursos computacionais pode melhorar o desempenho da aplicação em resposta à maneira de distribuir o processamento. Por outro lado, o grau de comunicação na rede exigido pelo método usado pode gerar um *overhead* que o torna não atrativo.

- **Portabilidade para a plataforma CORBA:** Atualmente os recursos do CORBA podem ser pouco utilizados pelo MakeD, entretanto esta tecnologia fornece mecanismos para a comunicação transparente entre componentes de aplicações distribuídas envolvendo diferentes plataformas, sistemas e linguagens. Dessa forma, a flexibilidade do CORBA pode facilitar a integração do MakeD com outras aplicações e possibilitar seu uso em diversas plataformas.

Os potenciais benefícios conseguidos com o MakeD justificam a continuação deste trabalho na linha do MakeDM (proposto no capítulo 6) para a manutenção e gerenciamento de projetos envolvendo múltiplos usuários compartilhando `makefiles`.

Apêndice A

ESTRUTURAS DE DADOS DO GNU *MAKE*

As principais estruturas de dados do GNU *Make*, extraídas da versão 3.74, são apresentadas a seguir.

Tabela de Opções de Comando

A estrutura `command_switch` define a composição da tabela de opções de comando.

```
/* The structure that describes an accepted command switch.  */

struct command_switch
  {
    char c;                     /* The switch character.  */

    enum                        /* Type of the value.  */
      {
        flag,                   /* Turn int flag on.  */
        flag_off,               /* Turn int flag off.  */
        string,                 /* One string per switch.  */
        positive_int,           /* A positive integer.  */
        floating,               /* A floating-point number (double).  */
        ignore                  /* Ignored.  */
      } type;

    char *value_ptr;              /* Pointer to the value-holding variable.  */
    unsigned int env:1;           /* Can come from MAKEFLAGS.  */
    unsigned int toenv:1;         /* Should be put in MAKEFLAGS.  */
    unsigned int no_makefile:1;   /* Don't propagate when remaking makefiles.  */

    char *noarg_value;            /* Pointer to value used if no argument
  is given.  */
    char *default_value;          /* Pointer to default value.  */
    char *long_name;              /* Long option name.  */
    char *argdesc;                /* Descriptive word for argument.  */
    char *description;            /* Description for usage message.  */
  };
```

A estrutura `switches` é a tabela propriamente dita e contém informações estáticas e apontadores para variáveis globais que armazenam o conteúdo das opções.

```
/* The table of command switches.  */

static const struct command_switch switches[] =
  {
    { 'b', ignore, 0, 0, 0, 0, 0, 0,
      0, 0,
      "Ignored for compatibility" },
    { 'C', string, (char *) &directories, 0, 0, 0, 0, 0,
      "directory", "DIRECTORY",
      "Change to DIRECTORY before doing anything" },
    { 'd', flag, (char *) &debug_flag, 1, 1, 0, 0, 0,
      "debug", 0,
      "Print lots of debugging information" },
    { 'e', flag, (char *) &env_overrides, 1, 1, 0, 0, 0,
      "environment-overrides", 0,
      "Environment variables override makefiles" },
    { 'f', string, (char *) &makefiles, 0, 0, 0, 0, 0,
      "file", "FILE",
      "Read FILE as a makefile" },
    { 'h', flag, (char *) &print_usage_flag, 0, 0, 0, 0, 0,
      "help", 0,
      "Print this message and exit" },
    { 'i', flag, (char *) &ignore_errors_flag, 1, 1, 0, 0, 0,
      "ignore-errors", 0,
      "Ignore errors from commands" },
    { 'I', string, (char *) &include_directories, 1, 1, 0, 0, 0,
      "include-dir", "DIRECTORY",
      "Search DIRECTORY for included makefiles" },
    { 'j', positive_int, (char *) &job_slots, 1, 1, 0,
      (char *) &inf_jobs, (char *) &default_job_slots,
      "jobs", "N",
      "Allow N jobs at once; infinite jobs with no arg" },
    { 'k', flag, (char *) &keep_going_flag, 1, 1, 0,
      0, (char *) &default_keep_going_flag,
      "keep-going", 0,
      "Keep going when some targets can't be made" },
    { 'l', floating, (char *) &max_load_average, 1, 1, 0,
      (char *) &default_load_average, (char *) &default_load_average,
      "load-average", "N",
      "Don't start multiple jobs unless load is below N" },
    { 'm', ignore, 0, 0, 0, 0, 0, 0,
      0, 0,
      "-b" },
    { 'n', flag, (char *) &just_print_flag, 1, 1, 1, 0, 0,
      "just-print", 0,
      "Don't actually run any commands; just print them" },
    { 'o', string, (char *) &old_files, 0, 0, 0, 0, 0,
      "old-file", "FILE",
      "Consider FILE to be very old and don't remake it" },
    { 'p', flag, (char *) &print_data_base_flag, 1, 1, 0, 0, 0,
      "print-data-base", 0,
      "Print make's internal database" },
    { 'q', flag, (char *) &question_flag, 1, 1, 1, 0, 0,
      "question", 0,
      "Run no commands; exit status says if up to date" },
    { 'r', flag, (char *) &no_builtin_rules_flag, 1, 1, 0, 0, 0,
      "no-builtin-rules", 0,
      "Disable the built-in implicit rules" },
```

```
    { 's', flag, (char *) &silent_flag, 1, 1, 0, 0, 0,
        "silent", 0,
        "Don't echo commands" },
    { 'S', flag_off, (char *) &keep_going_flag, 1, 1, 0,
        0, (char *) &default_keep_going_flag,
        "no-keep-going", 0,
        "Turns off -k" },
    { 't', flag, (char *) &touch_flag, 1, 1, 1, 0, 0,
        "touch", 0,
        "Touch targets instead of remaking them" },
    { 'v', flag, (char *) &print_version_flag, 1, 1, 0, 0, 0,
        "version", 0,
        "Print the version number of make and exit" },
    { 'w', flag, (char *) &print_directory_flag, 1, 1, 0, 0, 0,
        "print-directory", 0,
        "Print the current directory" },
    { 2, flag, (char *) &inhibit_print_directory_flag, 1, 1, 0, 0, 0,
        "no-print-directory", 0,
        "Turn off -w, even if it was turned on implicitly" },
    { 'W', string, (char *) &new_files, 0, 0, 0, 0, 0,
        "what-if", "FILE",
        "Consider FILE to be infinitely new" },
    { 3, flag, (char *) &warn_undefined_variables_flag, 1, 1, 0, 0, 0,
        "warn-undefined-variables", 0,
        "Warn when an undefined variable is referenced" },
    { '\0', }
  };
```

Tabela de *Hashing* de Arquivos - THA

```
/* Hash table of files the makefile knows how to make.  */

#ifndef FILE_BUCKETS
#define FILE_BUCKETS    1007
#endif

static struct file *files[FILE_BUCKETS];
```

Definição da estrutura `file`:

```
struct file
  {
    struct file *next;
    char *name;
    struct dep *deps;
    struct commands *cmds;      /* Commands to execute for this target.  */
    int command_flags;         /* Flags OR'd in for cmds; see commands.h.  */
    char *stem;                 /* Implicit stem, if an implicit
                                   rule has been used */
    struct dep *also_make;      /* Targets that are made by making this.  */
    time_t last_mtime;          /* File's modtime, if already known.  */
    struct file *prev;          /* Previous entry for same file name;
                                   used when there are multiple double-colon
                                   entries for the same file.  */

    /* File that this file was renamed to.  After any time that a
       file could be renamed, call `check_renamed' (below).  */
    struct file *renamed;

    /* List of variable sets used for this file.  */
```

```
    struct variable_set_list *variables;

    /* Immediate dependent that caused this target to be remade,
       or nil if there isn't one.  */
    struct file *parent;

    /* For a double-colon entry, this is the first double-colon entry for
       the same file.  Otherwise this is null.  */
    struct file *double_colon;

    short int update_status;     /* Status of the last attempt to update,
                                    or -1 if none has been made.  */

    enum                         /* State of the commands.  */
      {         /* Note: It is important that cs_not_started be zero.  */
        cs_not_started,          /* Not yet started.  */
        cs_deps_running,         /* Dep commands running.  */
        cs_running,              /* Commands running.  */
        cs_finished              /* Commands finished.  */
      } command_state ENUM_BITFIELD (2);

    unsigned int precious:1;     /* Non-0 means don't delete file on quit */
    unsigned int tried_implicit:1; /* Nonzero if have searched
                                      for implicit rule for making
                                      this file; don't search again.  */
    unsigned int updating:1;     /* Nonzero while updating deps of this file */
    unsigned int updated:1;      /* Nonzero if this file has been remade.  */
    unsigned int is_target:1;    /* Nonzero if file is described as target.  */
    unsigned int cmd_target:1;   /* Nonzero if file was given on cmd line.  */
    unsigned int phony:1;        /* Nonzero if this is a phony file
                                    i.e., a dependency of .PHONY.  */
    unsigned int intermediate:1;/* Nonzero if this is an intermediate file.  */
    unsigned int dontcare:1;     /* Nonzero if no complaint is to be made if
                                    this target cannot be remade.  */
  };
```

Definição da estrutura dep:

```
/* Structure representing one dependency of a file.
   Each struct file's `deps' points to a chain of these,
   chained through the `next'.

struct dep
  {
    struct dep *next;
    char *name;
    struct file *file;
    int changed;
  };
```

Definição dos comandos (de `file`) para gerar um *target*:

```
/* Structure that gives the commands to make a file
   and information about where these commands came from.  */

struct commands
  {
    char *filename;              /* File that contains commands.  */
    unsigned int lineno;         /* Line number in file.  */
    char *commands;              /* Commands text.  */
    unsigned int ncommand_lines;/* Number of command lines.  */
```

```
    char **command_lines;       /* Commands chopped up into lines.  */
    char *lines_flags;          /* One set of flag bits for each line.  */
    int any_recurse;            /* Nonzero if any `lines_recurse' elt has */
                                /* the COMMANDS_RECURSE bit set.  */
  };
```

Conjunto de Variáveis de Arquivo

```
/* Structure that represents a list of variable sets.  */

struct variable_set_list
  {
    struct variable_set_list *next;     /* Link in the chain.  */
    struct variable_set *set;           /* Variable set.  */
  };


/* Structure that represents a variable set.  */

struct variable_set
  {
    struct variable **table;    /* Hash table of variables.  */
    unsigned int buckets;       /* Number of hash buckets in `table'.  */
  };


/* Structure that represents one variable definition.
   Each bucket of the hash table is a chain of these,
   chained through `next'.  */

struct variable
  {
    struct variable *next;      /* Link in the chain.  */
    char *name;                 /* Variable name.  */
    char *value;                /* Variable value.  */
    enum variable_origin
      origin ENUM_BITFIELD (3); /* Variable origin.  */
    unsigned int recursive:1;   /* Gets recursively re-evaluated.  */
    unsigned int expanding:1;   /* Nonzero if currently being expanded.  */
    enum
      {
        v_export,               /* Export this variable.  */
        v_noexport,             /* Don't export this variable.  */
        v_ifset,               /* Export it if it has a non-default value.  */
        v_default               /* Decide in target_environment.  */
      } export ENUM_BITFIELD (2);
  };

/* Codes in a variable definition saying where the definition came from.
   Increasing numeric values signify less-overridable definitions.  */
enum variable_origin
  {
    o_default,          /* Variable from the default set.  */
    o_env,              /* Variable from environment.  */
    o_file,             /* Variable given in a makefile.  */
    o_env_override,     /* Variable from environment, if -e.  */
    o_command,          /* Variable given by user.  */
    o_override,         /* Variable from an `override' directive.  */
    o_automatic,        /* Automatic variable -- cannot be set.  */
    o_invalid           /* Core dump time.  */
```

```
};
```

Função que aloca memória para as estruturas de dados de um `file` (arquivo – node) e seus ascendentes no grafo (tabela de *hashing*):

```
/* Hash table of all global variable definitions.  */

#ifndef VARIABLE_BUCKETS
#define VARIABLE_BUCKETS                523
#endif
#ifndef PERFILE_VARIABLE_BUCKETS
#define PERFILE_VARIABLE_BUCKETS        23
#endif

void initialize_file_variables (file)
     struct file *file;
{
  register struct variable_set_list *l = file->variables;
  if (l == 0)
    {
      l = (struct variable_set_list *)
        xmalloc (sizeof (struct variable_set_list));
      l->set = (struct variable_set *) xmalloc (sizeof (struct variable_set));
      l->set->buckets = PERFILE_VARIABLE_BUCKETS;
      l->set->table = (struct variable **)
        xmalloc (l->set->buckets * sizeof (struct variable *));
      bzero ((char *) l->set->table,
             l->set->buckets * sizeof (struct variable *));
      file->variables = l;
    }

  if (file->parent == 0)
    l->next = &global_setlist;
  else
    {
      if (file->parent->variables == 0)
        initialize_file_variables (file->parent);
      l->next = file->parent->variables;
    }
}
```

Conjunto de Variáveis Gerais

```
/* Hash table of all global variable definitions.  */

#ifndef VARIABLE_BUCKETS
#define VARIABLE_BUCKETS                523
#endif

static struct variable *variable_table[VARIABLE_BUCKETS];
static struct variable_set global_variable_set
  = { variable_table, VARIABLE_BUCKETS };
static struct variable_set_list global_setlist
  = { 0, &global_variable_set };
struct variable_set_list *current_variable_set_list = &global_setlist;


/* Structure that represents one variable definition.
   Each bucket of the hash table is a chain of these,
   chained through `next'.  */
```

```
struct variable
  {
    struct variable *next;       /* Link in the chain.  */
    char *name;                  /* Variable name.  */
    char *value;                 /* Variable value.  */
    enum variable_origin
      origin ENUM_BITFIELD (3); /* Variable origin.  */
    unsigned int recursive:1;    /* Gets recursively re-evaluated.  */
    unsigned int expanding:1;    /* Nonzero if currently being expanded.  */
    enum
      {
        v_export,                /* Export this variable.  */
        v_noexport,              /* Don't export this variable.  */
        v_ifset,                 /* Export it if it has a non-default value.  */
        v_default                /* Decide in target_environment.  */
      } export ENUM_BITFIELD (2);
  };

/* Codes in a variable definition saying where the definition came from.
   Increasing numeric values signify less-overridable definitions.  */
enum variable_origin
  {
    o_default,          /* Variable from the default set.  */
    o_env,              /* Variable from environment.  */
    o_file,             /* Variable given in a makefile.  */
    o_env_override,     /* Variable from environment, if -e.  */
    o_command,          /* Variable given by user.  */
    o_override,         /* Variable from an `override' directive.  */
    o_automatic,        /* Automatic variable -- cannot be set.  */
    o_invalid           /* Core dump time.  */
  };
```

Lista de Processos em Execução (LPE)

```
/* Structure describing a running or dead child process.  */

struct child
  {
    struct child *next;          /* Link in the chain.  */

    struct file *file;           /* File being remade.  */

    char **environment;          /* Environment for commands.  */

    char **command_lines;        /* Array of variable-expanded cmd lines.  */
    unsigned int command_line;   /* Index into above.  */
    char *command_ptr;           /* Ptr into command_lines[command_line].  */

    pid_t pid;                   /* Child process's ID number.  */
    unsigned int remote:1;       /* Nonzero if executing remotely.  */

    unsigned int noerror:1;      /* Nonzero if commands contained a `-'.  */

    unsigned int good_stdin:1;   /* Nonzero if this child has a good stdin.  */
    unsigned int deleted:1;      /* Nonzero if targets have been deleted.  */
  };
```

Apêndice B

RESULTADOS DE TESTES CENTRALIZADOS

A tabela B.1 apresenta os tempos reais, de usuário e de sistema (em minutos) de cada processo de *make* centralizado utilizando o GNU *Make*, MakeD e Dmake. A coluna J indica a quantidade de tarefas concorrentes.

	GNU Make			MakeD			Dmake		
J	real	usuário	sistema	real	usuário	sistema	real	usuário	sistema
1	10:42.9	04:59.1	01:51.8	10:46.3	05:00.9	02:03.1	14:50.8	00:13.5	00:33.9
2	09:41.8	05:03.4	01:42.6	09:49.6	05:03.0	01:59.0	11:29.3	00:14.2	00:35.4
4	08:12.8	05:06.7	01:42.3	08:33.1	05:04.9	02:03.6	10:21.8	00:14.8	00:35.4
6	07:45.1	05:11.8	01:40.2	08:00.1	05:06.5	02:03.4	10:21.4	00:14.5	00:35.8
8	07:40.6	05:13.7	01:40.1	07:57.4	05:07.2	02:05.6	10:28.8	00:14.5	00:36.4
10	07:41.7	05:14.6	01:38.8	07:52.7	05:07.8	02:04.5	10:32.6	00:14.9	00:37.4
12	07:44.5	05:13.1	01:42.3	08:00.2	05:07.9	02:07.5	10:38.1	00:14.9	00:37.4
16	07:42.3	05:16.0	01:41.5	07:59.3	05:09.6	02:07.8	10:48.5	00:14.9	00:39.6

Tabela B.1: Tempos reais, de usuário e de sistema de processos de *make*.

Os percentuais de tempos reais relativos ao GNU *Make* com J=1 estão na tabela B.2 e são usados para gerar o gráfico 7.1.

J	GNUMake	MakeD	Dmake
1	100%	101%	139%
2	90%	92%	107%
4	77%	80%	97%
6	72%	75%	97%
8	72%	74%	98%
10	72%	74%	98%
12	72%	75%	99%
16	72%	75%	101%

Tabela B.2: Percentuais relativos ao GNU *Make* com J=1

Apêndice C

RESULTADOS DE TESTES DISTRIBUÍDOS

A tabela C.1 apresenta os tempos reais do GNU *Make* (centralizado), MakeD e Dmake. Para os dois últimos são apresentados os resultados em que a quantidade de estações participantes do processo de *make* varia de um a seis. Embora não mostrados graficamente, os resultados para J=12 e J=16 são incluídos em todas as tabelas deste Apêndice. O número junto ao nome do *make* indica a quantidade de estações utilizadas no processo de *make*, e J as tarefas concorrentes.

J	GNU Make	MakeD1	MakeD2	MakeD3	MakeD4	MakeD5	MakeD6
1	10:42.9	10:46.3	06:45.0	05:29.1	03:58.3	03:34.8	03:18.1
2	09:41.8	09:49.6	05:49.6	04:38.7	03:16.1	02:56.0	02:30.5
4	08:12.8	08:33.1	05:13.4	03:39.6	02:54.9	02:42.3	02:07.9
6	07:45.1	08:00.1	04:33.7	03:48.7	02:53.8	02:30.1	02:04.7
8	07:40.6	07:57.4	05:04.8	03:57.5	02:47.4	02:39.9	02:07.5
10	07:41.7	07:52.7	05:03.9	03:33.0	02:48.8	02:15.0	02:19.9
12	07:44.5	08:00.2	05:12.0	03:19.3	02:38.6	02:35.3	01:51.3
16	07:42.3	07:59.3	04:57.4	03:22.8	02:50.1	02:17.0	02:19.9
J	GNU Make	Dmake1	Dmake2	Dmake3	Dmake4	Dmake5	Dmake6
1	10:42.9	14:50.8	11:22.8	05:50.8	05:26.2	04:10.9	03:58.5
2	09:41.8	11:29.3	08:31.8	04:42.6	04:32.3	03:30.9	02:58.7
4	08:12.8	10:21.8	07:07.5	04:21.7	04:05.9	02:59.1	02:52.4
6	07:45.1	10:21.4	08:07.7	04:29.3	03:32.5	03:10.8	03:13.2
8	07:40.6	10:28.8	06:06.6	04:35.8	03:31.5	03:23.4	02:52.3
10	07:41.7	10:32.6	05:45.6	04:37.1	03:14.8	03:11.0	02:55.6
12	07:44.5	10:38.1	05:38.7	04:55.0	03:35.7	02:55.8	03:13.8
16	07:42.3	10:48.5	05:59.5	04:30.7	03:45.0	03:14.7	03:13.7

Tabela C.1: Tempos reais para GNU *Make*, MakeD e Dmake.

Os respectivos tempos de usuário e sistema para o cenário de execução da tabela acima são mostrados nas tabela C.2 e C.3, respectivamente. Estes são tempos gastos somente na estação com o controlador *make*.

J	GNU Make	MakeD1	MakeD2	MakeD3	MakeD4	MakeD5	MakeD6
1	04:59.1	05:00.9	02:04.3	01:33.5	00:56.6	01:03.0	00:59.2
2	05:03.4	05:03.0	02:46.7	02:18.5	01:21.5	01:04.0	01:14.7
4	05:06.7	05:04.9	02:23.1	01:49.4	01:13.3	01:06.4	01:09.6
6	05:11.8	05:06.5	02:47.1	01:46.4	01:53.9	01:10.9	00:59.4
8	05:13.7	05:07.2	02:37.9	02:45.0	01:34.3	01:19.8	01:23.0
10	05:14.6	05:07.8	02:49.8	01:58.6	01:36.6	01:30.4	01:14.5
12	05:13.1	05:07.9	03:33.5	01:44.8	01:46.3	01:32.8	01:11.0
16	05:16.0	05:09.6	02:49.2	02:10.5	01:42.8	01:10.3	01:13.4
J	GNU Make	Dmake1	Dmake2	Dmake3	Dmake4	Dmake5	Dmake6
1	04:59.1	00:13.5	00:13.5	00:14.2	00:14.0	00:14.8	00:14.6
2	05:03.4	00:14.2	00:14.0	00:14.9	00:14.5	00:15.2	00:15.5
4	05:06.7	00:14.8	00:15.5	00:15.7	00:15.1	00:16.7	00:16.4
6	05:11.8	00:14.5	00:14.3	00:15.4	00:16.0	00:16.1	00:16.1
8	05:13.7	00:14.5	00:16.1	00:15.6	00:16.5	00:15.7	00:16.1
10	05:14.6	00:14.9	00:15.3	00:16.0	00:16.7	00:15.9	00:16.2
12	05:13.1	00:14.9	00:15.4	00:16.1	00:16.3	00:16.2	00:16.6
16	05:16.0	00:14.9	00:15.5	00:15.7	00:16.4	00:16.2	00:16.2

Tabela C.2: Tempos de usuário para GNU *Make*, MakeD e Dmake.

J	GNU Make	MakeD1	MakeD2	MakeD3	MakeD4	MakeD5	MakeD6
1	01:51.8	02:03.1	01:03.1	00:41.4	00:33.7	00:29.0	00:23.1
2	01:42.6	01:59.0	01:03.8	00:45.6	00:33.4	00:27.5	00:25.5
4	01:42.3	02:03.6	01:00.3	00:42.6	00:34.4	00:31.5	00:26.8
6	01:40.2	02:03.4	01:01.1	00:47.8	00:36.9	00:31.1	00:25.9
8	01:40.1	02:05.6	01:04.9	00:49.9	00:38.7	00:33.5	00:27.3
10	01:38.8	02:04.5	01:07.3	00:48.3	00:39.1	00:33.3	00:33.4
12	01:42.3	02:07.5	01:08.4	00:49.7	00:37.6	00:38.8	00:27.7
16	01:41.5	02:07.8	01:11.7	00:51.7	00:47.7	00:39.1	00:38.9
J	GNU Make	Dmake1	Dmake2	Dmake3	Dmake4	Dmake5	Dmake6
1	01:51.8	00:33.9	00:34.4	00:36.1	00:37.0	00:38.6	00:37.8
2	01:42.6	00:35.4	00:35.7	00:39.7	00:38.8	00:41.3	00:42.2
4	01:42.3	00:35.4	00:38.3	00:41.2	00:42.1	00:44.5	00:47.8
6	01:40.2	00:35.8	00:37.4	00:42.6	00:44.7	00:45.6	00:44.8
8	01:40.1	00:36.4	00:39.6	00:43.2	00:44.5	00:44.3	00:44.4
10	01:38.8	00:37.4	00:41.4	00:43.4	00:44.2	00:44.4	00:46.1
12	01:42.3	00:37.4	00:39.8	00:44.0	00:44.2	00:46.4	00:46.3
16	01:41.5	00:39.6	00:40.3	00:43.6	00:45.7	00:45.1	00:44.5

Tabela C.3: Tempos de sistema para GNU *Make*, MakeD e Dmake.

Para o MakeD, os percentuais de tempos reais relativos ao GNU *Make* com J=1 estão na tabela C4 e são usados para gerar o gráfico 7.2. Para o Dmake, os percentuais se encontram na tabela C.5 e são usados para gerar o gráfico 7.3.

J	GNU Make	MakeD1	MakeD2	MakeD3	MakeD4	MakeD5	MakeD6
1	100%	101%	63%	51%	37%	33%	31%
2	90%	92%	54%	43%	31%	27%	23%
4	77%	80%	49%	34%	27%	25%	20%
6	72%	75%	43%	36%	27%	23%	19%
8	72%	74%	47%	37%	26%	25%	20%
10	72%	74%	47%	33%	26%	21%	22%
12	72%	75%	49%	31%	25%	24%	17%
16	72%	75%	46%	32%	26%	21%	22%

Tabela C.4: Percentuais do MakeD relativos ao GNU *Make* com J=1

J	GNU Make	Dmake1	Dmake2	Dmake3	Dmake4	Dmake5	Dmake6
1	100%	139%	106%	55%	51%	39%	37%
2	90%	107%	80%	44%	42%	33%	28%
4	77%	97%	66%	41%	38%	28%	27%
6	72%	97%	76%	42%	33%	30%	30%
8	72%	98%	57%	43%	33%	32%	27%
10	72%	98%	54%	43%	30%	30%	27%
12	72%	99%	53%	46%	34%	27%	30%
16	72%	101%	56%	42%	35%	30%	30%

Tabela C.5: Percentuais do Dmake relativos ao GNU *Make* com J=1.

A tabela C.6 apresenta os percentuais de desempenho do MakeD relativos ao Dmake para cada processo de *make* com mesmo valor de J. Os percentuais da tabela são usados para formar o gráfico 7.4.

J	MakeD1	MakeD1	MakeD3	MakeD4	MakeD5	MakeD6
1	73%	59%	94%	73%	86%	83%
2	86%	68%	99%	72%	83%	84%
4	83%	73%	84%	71%	91%	74%
6	77%	56%	85%	82%	79%	65%
8	76%	83%	86%	79%	79%	74%
10	75%	88%	77%	87%	71%	80%
12	75%	92%	68%	74%	88%	57%
16	74%	83%	75%	76%	70%	72%

Tabela C.6: Percentuais do MakeD relativos ao Dmake, com mesmo J.

O desempenho do MakeD (tempos em segundos) à medida que aumenta o número de estações é colocado na tabela C.7 e usado para gerar o gráfico 7.5.

J	GNU Make	MakeD1	MakeD2	MakeD3	MakeD4	MakeD5	MakeD6
1	643	646	405	329	238	215	198
2	582	590	350	279	196	176	150
4	493	513	313	220	175	162	128
6	465	480	274	229	174	150	125
8	461	477	305	238	167	160	127
10	462	473	304	213	169	135	140
12	465	480	312	199	159	155	111
16	462	479	297	203	170	137	140

Tabela C.7: Tempos reais de execução do MakeD.

REFERÊNCIAS BIBLIOGRÁFICAS

[Acc86] Mike Acceta, Robert Baron, David Golub, et. ali., "Mach: A New Kernel Foundation for UNIX Development", Technical Report, Carnegie Mellon University, August 1986.

[And83] Gregory R. Andrews e Fred B. Schneider, "Concepts and Notations for Concurrent Programming", *ACM Computing Surveys*, Vol. 15, No. 1, pp. 3-43, March 1983.

[Bal87] Henry E. Bal, R. van Renesse and Andrew S. Tanenbaum, "Implementing Distributed Algorithms Using Remote Procedure Calls", *Proceedings of the 1987 National Computer Conference*, Chicago, Illinois, pp. 499-506, June 1987.

[Bal89] Henry E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum, "Programming Languages for Distributed Computing Systems", *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261-322, September 1989.

[Beg94] A. Beguelin, A. Geist, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM 3 User's Guide and Reference Manual*, ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, May 1994.

[Bir84] Andrew D. Birrell and Bruce Jay Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, February 1984.

[Car93] Ivonne M. Carrazana, *Sistema Gerenciador de Processamento Cooperativo - SGPC*, Tese de Mestrado, DCC-IMECC-UNICAMP, 1993.

[Che88] David R. Cheriton, "The V Distributed System", *Communications of the ACM, Vol. 31* (3), pp. 314-333, March 1988.

[Cho90] "Overview of the CHORUS® Distributed Operating System", Technical Report, Chorus Systèms CS/TR-09-25, April 1990.

[Cia94] Cassius Di Cianni, *OMNI - Sistema de Suporte a Aplicações Distribuídas*, Tese de Mestrado, DCC-IMECC-UNICAMP, agosto 1994.

[Coo87] Eric. C. Cooper, "Distributed Systems Technology Survey", Technical Report CMU/SEI-87-TR-5, Software Engineering Institute, Carnegie Mellon University, 1987.

[Dev95] Judith E. Devaney, Robert Lipman, Minwen Lo, Willian F. Mitchell, *The Parallel Applications Development Environment (PADE) User's Manual*, National Institute of Standards and Technology, November 21, 1995.

[Don93] J. Dongarra, G. A. Geist, R. Manchek, V. S. Sunderam, "Supporting Heterogeneous Network Computer: PVM", Oak Ridge National Laboratory and University of Tennessee, 1993.

[Duk94] D.W. Duke, T. P. Green, J. L. Pasko, "Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queue System Version 3.0", Relatório Técnico, Supercomputer Computations Research Institute Florida State University, Tallahassee, Florida, March 2, 1994.

[Dwy94] Frank Dwyer, "Using the Distributed Queueing System Parallel Make Utility", Relatório Técnico, Supercomputer Computations Research Institute, Florida State University, January 14, 1994.

[Eyk92] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Willians, "Beyond Multiprocessing: Multithreading the SunOS Kernel", Proceedings USENIX, Summer 1992.

[Fel79] Stuart I. Feldman, "MAKE – A Program for Maintaining Computer Programs", *Software – Practice and Experience*, Vol. 9(4), pp. 255-265, April 1979.

[Fle89] Charles J. Fleckenstein and David Hemmendinger, "Using a Global Name Space for Parallel Execution of UNIX Tools", *Communications of the ACM, Volume 32*, 9, September 1989.

[Gei93] G. A. Geist, V. S. Sunderam, "Network Based Concurrent Computing on the PVM System", Relatório Técnico, Oak Ridge National Laboratory, Oak Ridge, TN 37831.

[Gei94] A. Geist, A. Geguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*, Massachusetts Institute of Technology, Cambridge, MA 02142, 1994.

[Gei94a] A.Geist, A. Geguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM3 User's Guide and Reference Manual*, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, May 1994.

[Gel85] D. Gelernter, "Generative Communication in Linda", *ACM Transactions on Programming Language Systems*, 7,1, January 1985, pp 80-112.

[Gon94] Celso Gonçalves Jr., *Objetos Distribuídos*. Teste de Mestrado, DCC-IMECC-UNICAMP, agosto 1994.

[Gon94a] Celso Gonçalves Jr., B. Coutinho e Rogério Drummond, "Controle de Concorrência em Objetos", Relatório Técnico, Laboratório A-Hand, DCC-IMECC-UNICAMP, junho 1994.

[Int94] Intel Corporation Inc., *Paragon Application Tools User's Guide*, June 1994.

[Int94a] Intel Corporation Inc., *Paragon User's Guide*, June 1994.

[IONA96] IONA Technologies, Ltd., *Orbix Programming Guide*, October 1996

[Kle96] Steve Kleiman, Devang Shah, Bart Smaalders, *Programming with Threads*, SunSoft Press, 1996.

[Law66] E. L. Lawyer and D. E. Wood, "Branch-and-bound Methods: a Survey", *Operating Research*, Vol. 14, No. 4, July 1966, pp. 699-719.

[Loi96] Martin Loitz, Código Fonte do Lmake, Technical University Braunschweig, Germany, 1996.

[Mor96] D. J. Morton and J. M. Tyler, "Minimizing Development Overhead with Partial Parallelization", *IEEE Parallel & Distributed Technology*, Fall 1996, Vol. 4, Number 3.

[Mul90] Sape J. Mullender, Guido van Rossum, Andrew S. Tenenbaum, Robbert van Renesse, Hans van Staveren, "Amoeba: A Distributed Operating System for the 1990s", *Computer*, pp. 44-53, May 1990.

[OMG97] Object Management Group, Inc., "The Common Object Request Broker: Architecture and Specification", Revision 2.1, August 1997.

[OSF90] Open Software Foundation, "Distributed Computing Environment Overview", White paper, 1990.

[Pal87] Rich Palkovic, "SunPro: The Sun Programming Environment", Sun Technical Report, Sun Microsystems, Inc., 1987, pp. 67-86.

[Pow91] M. L. Power, S. R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, "The SunOS Multi-thread Architecture", *Proceedings Winter 1991 USENIX Conference*, January, 1991.

[Rev92] Louis S. Revor, *DQS User's Guide*, Computing and Telecommunications Division. Argonne National Laboratory, September 15, 1992.

[Sch96] B. Schnor, S. Petri, R. Oleyniczak, H. Langendörfer, "Scheduling of Parallel Applications on Heterogeneous Workstations Clusters", *Proceedings of PCDS'96, the ISCA 9th International Conference on Parallel and Distributed Computing Systems*, pp. 330-337, France, September 1996.

[Sta95] Richard M. Stallman and Roland McGrath, *GNU Make User's Guide for Make Version 3.73 Beta*, Free Software Foundation, Inc., April 1995.

[Ste92] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley Publishing Company, 1992.

[Sun87] Sun Microsystems, Inc., "The Sun Network File System: Design, Implementation and Experience", Mountain View (CA), 1987.

[Sun87a] Sun Microsystems, Inc., "XDR: External Data Representation Standard", Internet Request For Comments – RFC1014, June 1987.

[Sun94] Sun Microsystems, Inc., *Network Interfaces Programmer's Guide*, Mountain View (CA), August 1994. Manual técnico para programação RPC no SunOS 5.4 (Solaris 2.4).

[Sun94a] Sun Microsystems, Inc., *Multithreaded Programming Guide*, Mountain View (CA), August 1994. Manual técnico para programação *multithreaded* no SunOS 5.4 (Solaris 2.4).

[Sun97] Sun Microsystems, Inc., "Java Remote Method Invocation – Distributed Computing for java", White paper, 1997.

[Tan81] Andrew S. Tanenbaum and S. J. Mullender, "An Overview of the Amoeba Distributed Operating System", *Operating Systems Review*, Vol. 15, No. 3, July 1981, pp. 51-64.

[Tand96] Fredy Tandiary, Suraj C. Kothari, Ashish Dixit and E. Walter Anderson, "Batrun: Utilizing Idle Workstations for Large-Scale Computing", *IEEE Parallel & Distributed Technology*, Summer 1996, Volume 4, 2, pp 41-48.

[Yac88] Greg Yachuk, Código Fonte do Gymake, Informix Software, Inc., Menlo Park CA 94114, 1988.