MakeD – Make Distribuído[1]

Aredis S. de Oliveira e Rogério Drummond
e-mail: {aredis, rog}@ahand.unicamp.br

Instituto de Computação, UNICAMP
Laboratório A-HAND
Rua Roxo Moreira, 1076
13083–591 Campinas, SP

(Agosto de 1997)

Resumo

Este artigo descreve a transformação da aplicação *make* centralizada numa ferramenta distribuída que usa os recursos disponíveis na rede local. O *Make* Distribuído (MakeD) foi implementado com o objetivo de tornar o processo de *make* mais eficiente, e analisar aspectos do desenvolvimento de aplicações distribuídas a partir de aplicações centralizadas e de grande porte.

A implementação da aplicação cliente/servidor 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 obtidos demonstram um ganho no desempenho, que atinge 65% em relação ao *make* original. Se combinar a distribuição e a concorrência nas estações, o ganho relativo chega a 80%.

Palavras-chave: *Make* distribuído, aplicações distribuídas, *threads*, transformação de aplicações centralizadas em distribuídas.

Abstract

This paper discusses the process of transforming centralized make application in a distributed tool that uses local network resources. The Distributed Make (MakeD) was implemented aiming at more efficient make process and as test case for the development of distributed application derived from its existing centralyzed.

The implementation of the Client/Server MakeD uses RPC and multithreading to ensuse distribution in the local network and multitasking within each node. The results show performance gains of 65% compared to original make. Performance gains reachs 80% when combinning distribution and multitasking.

Keywords: Distributed make, distributed aplications, threads, transforming centralized aplications in distributed ones.

1. Este trabalho foi parcialmente financiado pelo CNPq, processos número 680089/94–2 e 131786/94-6 e pela FAPESP, processo número 94/5179-6.

I - INTRODUÇÃO

As aplicações disponíveis atualmente têm oferecido progressivamente 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 das técnicas de projeto já conhecidas, 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 dos programas, o *make* [Fel79] é uma ferramenta imprescindível que, além de automatizar o processo de compilação, recompila 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* ainda concentra o esforço do desenvolvimento numa única máquina.

Este trabalho descreve a implementação da transformação da aplicação *make* numa aplicação distribuída, cliente/servidor, prática e de uso geral que soluciona limitações da ferramenta original ao acrescentar eficiência com o uso dos recursos da rede. Além disso, são discutidos aspectos do desenvolvimento de aplicações distribuídas a partir de aplicações centralizadas complexas e de grande porte.

Ao utilizar recursos da rede, o MakeD pode explorar não apenas o alto grau de paralelismo que o ambiente oferece, mas também a concorrência de execuções dentro de um mesmo processador, através dos recursos do RPC (*Remote Procedure Call*) *multithreaded* [Sun94].

O MakeD oferece diversas vantagens, principalmente no que se refere ao tempo de processamento e distribuição de tarefas entre as máquinas; viabiliza a possibilidade de se fazer bom uso da capacidade de processamento disponível na rede local, na medida em que contribui para o balanceamento da carga. Embora a aplicação seja distribuída, é completamente compatível com a ferramenta original.

A implementação de aplicações distribuídas é uma tendência em plena ascensão, visando explorar os recursos computacionais da rede (interconectados por um meio físico de alta velocidade). Essas aplicações exigem um nível de controle e complexidade muito maior em relação às aplicações centralizadas. Para seu desenvolvimento eficiente, rápido e confiável, é necessário que o ambiente de suporte à programação forneça um grau de abstração razoável.

Algumas propriedades são bastante desejáveis nesses ambientes: heterogeneidade e interoperabilidade, portabilidade, facilidade de uso, orientação a objetos, segurança, tolerância a falhas, agrupamento de objetos e transparência quanto à localização. Esses requisitos são detalhados em [Cia94], e padrões como o DCE (*Distributed Computing Environment*) [OSF90] proposto pela OSF (*Open Software Foundation*) e CORBA (*Common Object Request Broker Architecture*) [OMG91, OMG92], proposto pela OMG (*Object Management Group*), estão procurando suprir essas necessidades, com promessas no sentido de se estabelecerem como uma base para a efetiva integração de diferentes ambientes no futuro.

Ambientes de suporte à programação distribuída têm se mostrado um meio eficaz, viável e economicamente atrativo para o desenvolvimento de 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 *LegoShell* [Dru96], e ainda os padrões DCE [OSF90] e CORBA [OMG91, OMG92].

Os sistemas operacionais distribuídos são especializados [Gei93] e dificultam a portabilidade das aplicações [Cia94]. O suporte aos requisitos citados implica um forte impacto no

desempenho, um problema sério dos sistemas distribuídos [Mul90]. As linguagens para programação distribuída estão se tornando realidade devido à atenção recebida nos últimos anos [Gon94], o mesmo acontecendo com os padrões DCE e CORBA que permitem o desenvolvimento de aplicações distribuídas com expectativas crescentes de interoperabilidade num futuro próximo [Gon94].

Tratando-se de uma extensão da aplicação *make* existente, a implementação do MakeD não requer ambiente diferenciado. Seu desenvolvimento é baseado nas ferramentas disponíveis nos sistemas Unix, como NFS (*Network File System*) [Sun87], RPC *Multithreaded* [Sun94], XDR (*eXternal Data Representation*) [Sun88] e protocolos TCP/IP. O NFS garante um sistema de arquivos comum entre as estações; o RPC é responsável pelo padrão que os clientes usam para requisitar serviços aos servidores, e a linguagem XDR trata da representação dos tipos de dados que são transmitidos na rede.

Essas três ferramentas suportam amplamente a programação distribuída cliente/servidor com baixo custo, estando disponíveis em vários ambientes Unix, o que facilita a portabilidade. Embora não forneçam níveis de abstração como os dos ambientes citados anteriormente, não impedem a implementação de propriedades consideradas importantes, como por exemplo tolerância a falhas; a implementação pode ser efetuada tanto a nível de aplicação quanto a nível de sistema operacional.

Na próxima seção será caracterizado o *make*. Na seqüência, serão descritos aspectos de implementação do MakeD. A visão em contexto é então o próximo tópico, com abordagem de trabalhos na mesma linha. A seguir, serão apresentados os aspectos relevantes do processo de transformação de uma aplicação centralizada em distribuída com base na experiência adquirida. Finalmente, são apresentadas uma análise de desempenho da aplicação e as conclusões.

II - CARACTERIZAÇÃO DA FERRAMENTA MAKE

Diversas são as tarefas envolvidas no processo de viabilização de um projeto de médio ou grande porte: compilação, 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 dispendido com tarefas repetitivas, aumentando a disponibilidade dos participantes para a especificação, análise e desenvolvimento.

Uma ferramenta originalmente desenvolvida no sistema Unix, *make* permite automatizar o desenvolvimento de projetos que consistem de vários módulos, que podem ser gerados de diversas maneiras. Usos habituais do *make* incluem manutenção e atualização de programas e bibliotecas, execução de baterias de testes, instalação e distribuição. Em conjunto com o SCCS *(Source Code Control System)* o *make* provê regras especiais para a recuperação automática de arquivos fonte pertencentes a um projeto com várias versões.

Make utiliza um arquivo chamado `makefile` que contém regras descrevendo a relação de dependências entre os arquivos e as tarefas a serem realizadas. A partir desse arquivo, o utilitário determina automaticamente quais partes de um projeto precisam ser refeitas e invoca os comandos necessários.

O formato básico para as regras no arquivo `makefile` é mostrado na Figura 1, juntamente com um exemplo bastante simples. Além das regras, esse arquivo pode conter uma mistura de entradas de definição de variáveis, comentários e diretivas: *define*, condicional (*ifdef*, *ifndef*, *endif*, etc) e *include* (para inclusão de outros `makefiles`).

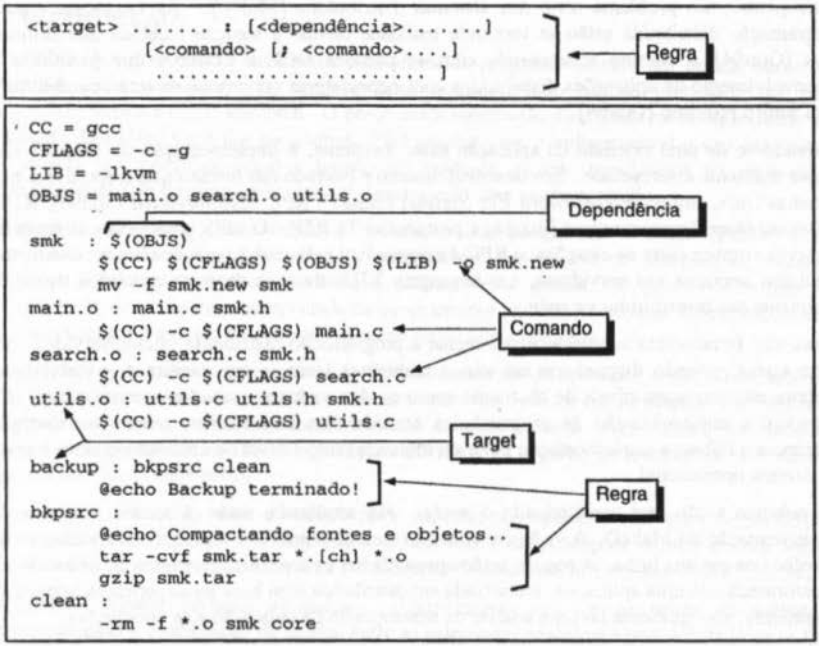

Figura 1: Formato das regras e um exemplo de `makefile`

Um *target* é geralmente um arquivo a ser gerado. Pode ser também uma ação a ser executada, tal como `clean` no exemplo acima. 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.

O `makefile` é usado apenas para construir em memória um grafo de dependências direcionado e hierárquico, onde cada nó está associado a um *target* e contém os comandos que constroem o *target* ou executa uma ação (Figura 2).

No processamento dos nós do grafo[2] é usado um algoritmo de busca em profundidade, que atualiza um nó pai somente se algum de seus filhos for mais recente e após o término da 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).

2. Processar um nó significa executar seqüencialmente seus comandos e gerar um *target* atualizado.

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 percorrer recursivamente o grafo (ou sub-grafos) várias vezes, garantindo e controlando a concorrência de processamento dos nós (geração de *targets*) através do uso de *flags* de controle contidas em cada nó. O número máximo de *jobs* executando concorrentemente é determinado pelo usuário e pela carga média do sistema. Num dado instante, só existe um comando de cada nó em processamento sendo executado, até mesmo porque um comando pode depender do resultado do anterior. E, para um dado nó é obedecida a seqüência na execução dos comandos.

Para o `makefile` da Figura 1, é gerado um grafo como o da figura abaixo. Cada c_n em um nó representa um comando da regra presente no `makefile`.

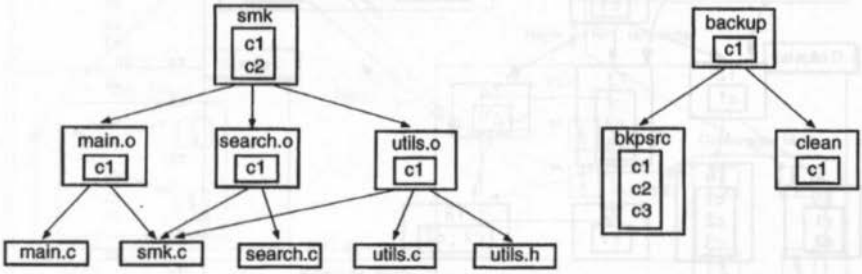

Figura 2: Grafo de dependências

O *make* é uma ferramenta de uso geral para gerar um produto final construído através de várias etapas intermediárias. A versão distribuída mantém toda essa funcionalidade e preserva as características originais da aplicação.

III - IMPLEMENTAÇÃO DO MAKE DISTRIBUÍDO

Como o *make* é uma aplicação centralizada e *singlethreaded*, os comandos nos nós do grafo são executados localmente, cada comando gerando um processo filho[3]. Nas versões de *make* suportando *jobs* concorrentes, em princípio haveria melhor aproveitamento da máquina usada. Entretanto, o usuário deve judiciosamente limitar o número máximo de *jobs* sob risco de sobrecarregar o sistema, na realidade piorando o desempenho em relação ao *make* convencional.

A Figura 3 ilustra um cenário de execução num dado instante em que três é o valor máximo para a quantidade de *jobs* concorrentes. Na figura, os nós T5, T6 e T3 estão em processamento e seus respectivos comandos c1, c2 e c3 em execução. Quando o algoritmo de busca detectar o término de c1 ou c2, o próximo comando do respectivo nó será iniciado. No caso da conclusão de c3, o *target* T3 será dado como gerado e um novo nó será escalonado.

Na implementação do MakeD, as tarefas de geração de *targets* são paralelizadas, ou seja, divididas 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

3. Alguns comandos triviais dispensam processos.

segundo as capacidades relativas das estações, consideradas em número de tarefas possíveis de serem executadas simultaneamente.

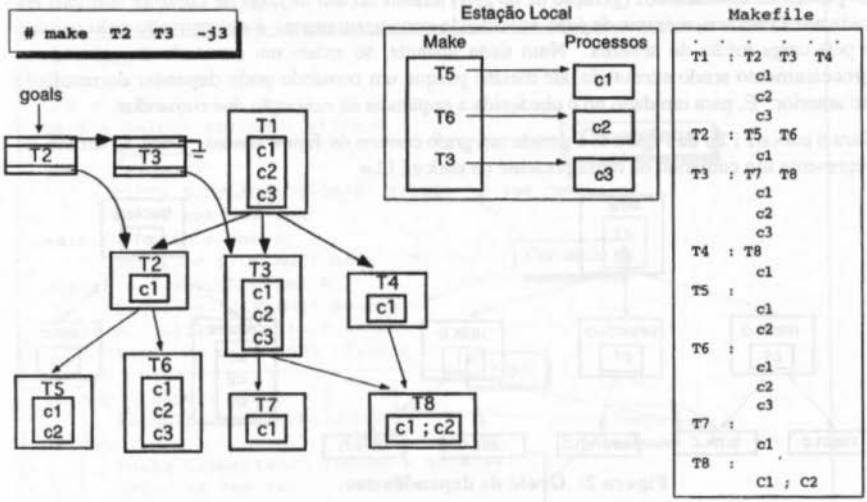

Figura 3: `Makefile`, grafo e execução de *jobs* num dado instante.

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 comandos deste e construir um *target*. Esta decisão, juntamente com a troca de processos filhos por *threads*[4] causou grandes alterações no código do *make*, inclusive na re-implementação do controle de execução de processos, antes baseado na criação e manipulação de processos filhos.

A implementação do MakeD exigiu também 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 aos servidores de processamento. Decide em qual estação a tarefa será processada com base na capacidade de processamento das estações e no balanceamento da distribuição dos serviços.
- **Servidor de Processamento (SP):** Servidor multitarefa (*multithreaded*) 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 vários pedidos ao mesmo tempo.

A Figura 4 mostra um exemplo do funcionamento do MakeD envolvendo quatro máquinas.

4. Os chamados fluxos de execução são denominados *lightweight processes* ou *threads* (termo usado neste artigo).

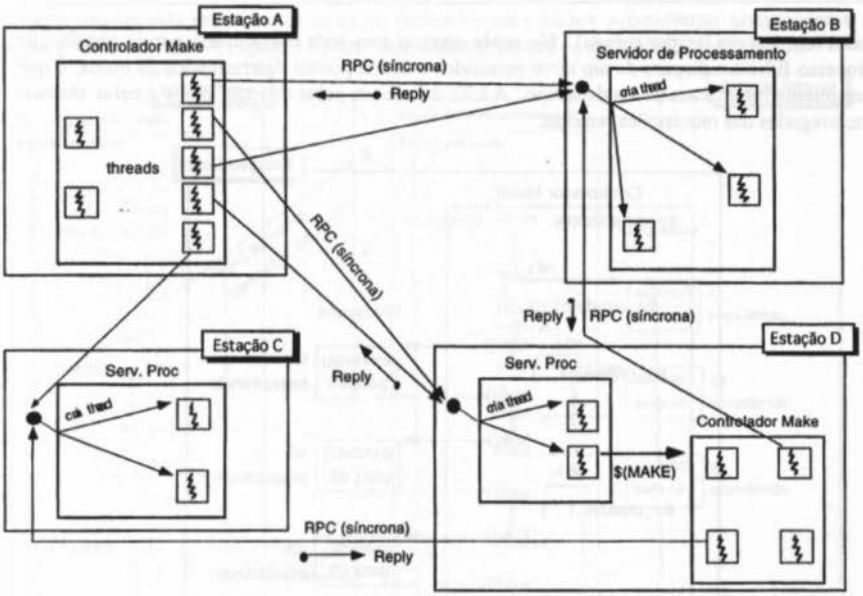

Figura 4: Execução do make distribuído usando quatro máquinas.

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, para cada nó escalonado, seus comandos 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 (Figura 5). Se o processamento for remoto, é escolhida uma estação para efetuá-lo, e a *thread* criada insere os parâmetros nas estruturas do XDR e faz a chamada RPC, a qual solicita ao SP da máquina escolhida para processar o nó (Figura 6).

A RPC é síncrona, ou seja, a *thread* bloqueia à espera dos resultados ou até que seja detectado um *timeout*[5]. 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* no algoritmo de busca.

A lista de processos em execução 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. 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

5. Intervalo de tempo configurável para espera da resposta. Pode ser usado para detectar falhas na estação remota.

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 *threads* encarregadas das requisições remotas.

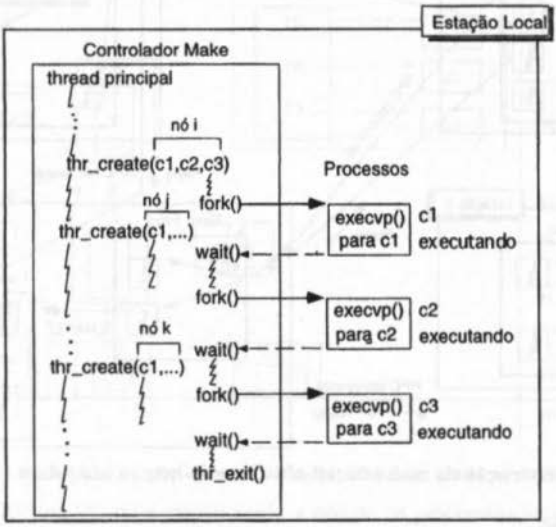

Figura 5: Execução local dos comandos de um nó.

Servidor de Processamento (SP)

O 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 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.

A quantidade máxima de tarefas simultâneas é determinada na configuração do SP e limitada pelos mecanismos *mutex* e variável condicional, e as variáveis compartilhadas são protegidas de acessos indevidos pelo *mutex*.

Cada *thread* atualiza um target criando um processo filho para cada comando do nó associado (Figura 6). Eventuais mensagens (*stdout*/*stderr*) dos comandos são redirecionadas para um arquivo temporário cujo nome, gerado automaticamente, 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.

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 e ao mesmo tempo, ou seja, possui a capacidade multi-usuário.

O uso recursivo do *make* é preservado no controlador *make* e incluído no servidor de processamento. 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 4 ilustra uma chamada recursiva do MakeD na estação D.

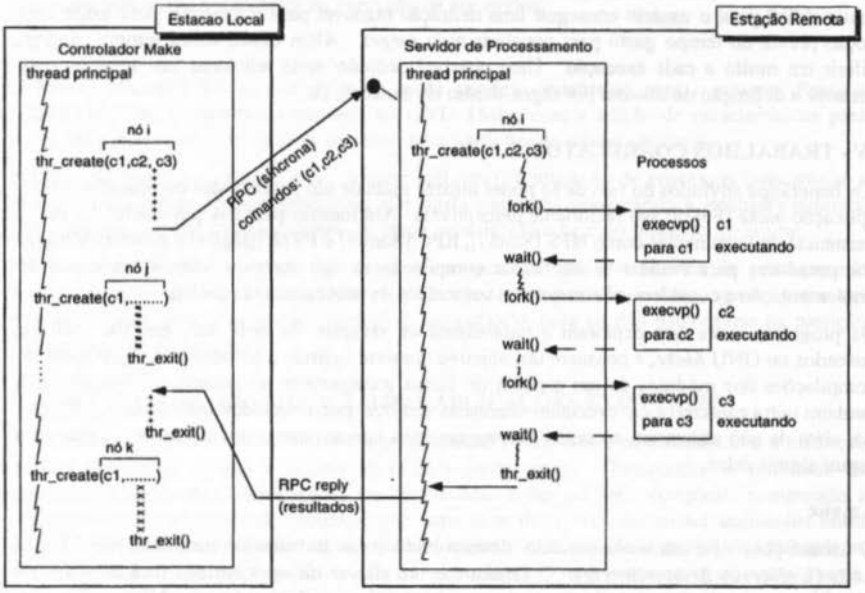

Figura 6: Execução remota dos comandos de um nó.

Considerações de Implementação

Como ponto de partida para a implementação do MakeD foi escolhido o *GNU Make* [Sta95], versão 3.74, a mais recente no início do trabalho (cerca de 20000 linhas em linguagem C); é uma das implementações mais completas e segue o padrão POSIX.2 (IEEE.2 – 1992).

A plataforma de *hardware* é constituída por um conjunto de *workstations Sun*[6]. O sistema operacional usado é o Solaris 2.5[7] (*System V*), e foi seguido o seu padrão de *threads*; porém a mudança necessária para tornar a aplicação compatível com *threads* POSIX é mínima. Toda 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 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 claramente mais compacto, elegante e fácil de entender se comparado aos processos filhos. 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 *debug* adequadas.

6. Sun Microsystem, Inc.
7. SunOS 5.5 Operating System.

Devido a preocupações com a portabilidade, o *timeout* possui um valor *default* para todas as tarefas realizadas remotamente, podendo ser redefinido via variável de ambiente. Entretanto, é muito difícil para o usuário conseguir uma definição razoável para o *timeout*, pois exige uma noção prévia do tempo gasto para construir cada *target*. Além disso, esses tempos podem diferir em muito a cada execução. Uma possível solução seria adicionar um recurso para permitir a definição do *timeout* por regra, dentro do `makefile`.

IV - TRABALHOS CORRELATOS

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 [Sun94] 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.

Os programs *make* que exploram o paralelismo ou recursos de rede são, em sua maioria, baseados no GNU *Make*, e possuem um objetivo comum: agilizar e automatizar o processo de compilações dos módulos de um sistema de forma transparente ao usuário. Compartilham também outra característica: executam chamadas remotas por comandos individuais e não por nó, além de não utilizarem recursos de programação concorrente como *threads*. Vejamos a seguir alguns deles.

Lmake

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

O Lmake é baseado no *GNU Make* versão 3.67, e dividido em duas partes: serial e 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. A fase paralela constrói o grafo de dependências através do arquivo temporário e usa PVM para enviar as mensagens (comandos) aos *daemons lmaked* ativos em cada máquina do *cluster*.

Dmake

O Dmake (*Distributed Make*) [Dwy94, Duk94] é um *make* distribuído baseado no GNU Make, versão 3.69, que exige um sistema de arquivos comum 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.

Os pedidos de compilações locais remotas são feitos via um programa cliente/servidor `rsh` (*remote shell*) melhorado. Para a compilação de um arquivo, o Dmake passa o comando para o *spew* (cliente `rsh`) e este o envia (via `rsh`) para o *rspew* (servidor `rsh`), que por sua vez recebe os dados, processa a requisição e devolve os resultados. Dessa forma, é criado dois processos filhos (*spew* e `rsh`) adicionais para a execução local e remota de comandos. Em contrapartida, o MakeD precisa somente de uma *thread* para solicitar o processamento local ou remoto de um nó por completo.

Devido ao *overhead* gerado pelo Dmake, recomenda-se usá-lo com um número reduzido de estações e poucas compilações simultâneas em cada. O principal problema é a quantidade de processos adicionais necessários na execução de comandos.

Pmake

O Pmake (*Parallel Make*) [Int94] é um *make* paralelo construído para o sistema Paragon (supercomputador paralelo) e baseado no *GNU Make*, com a adição de características para controlar a execução paralela feita nas chamadas partições do sistema Paragon.

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 cujo 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*.

V - ASPECTOS DE DISTRIBUIÇÃO DAS APLICAÇÕES EXISTENTES

A existência de ambientes de suporte à programação distribuída tem estimulado a construção de tais aplicações devido a fatores já citados neste artigo. Entretanto, o problema de transformar aplicações centralizadas em distribuídas é tão ou mais complexo comparado a aplicações originariamente distribuídas, pois uma série de fatores devem ser analisados antes da transformação propriamente dita. Fatores como a natureza da aplicação, portabilidade, eficiência, algoritmos empregados, custo da comunicação entre processadores, entre outros, são importantes durante a análise da viabilidade para a 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.

A seguir são discutidos alguns aspectos de distribuição de aplicações relacionados com a experiência prática de implementação do *make* distribuído a partir da versão centralizada.

Natureza da Aplicação

A natureza de uma aplicação é um fator decisivo para definir uma mudança de ambiente de execução. Se existe muita interação entre as partes de um programa e o tempo de comunicação é crítico, certamente um sistema multiprocessador com memória compartilhada é o ideal, e tentativas no sentido de torná-lo distribuído podem não ser satisfatórias.

O *make* é uma aplicação natural para qualquer sistema multiprocessador, onde as compilações podem ser feitas em paralelo, em processadores separados [Fle89]. Por outro lado, uma multiplicação distribuída de uma matriz realizada por [Fle89] piorou o desempenho comparada a uma centralizada. O tempo de comunicação na rede comparado com o tempo da tarefa realizada e o método empregado não justificaram a mudança no algoritmo.

Segmentação do Problema em tarefas

O primeiro passo é analisar os algoritmos na tentativa de identificar tarefas que podem ser executadas paralelamente, e escalonadas de tal forma que o tempo de execução seja minimizado. Se isto não for possível, 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*. Há vantagens em distribuir uma aplicação somente se grande parte do tempo for gasto em computações, sendo a comunicação entre processos pouco freqüente.

No caso do MakeD, decidiu-se empacotar todos 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 paralelizavéis. Por exemplo, na Figura 1, o tempo gasto pelo segundo comando que participa da geração do *target* smk, não justifica sua separação do primeiro para uma chamada remota.

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 um grande número de 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.

No MakeD, tanto o cliente quanto o servidor possuem a capacidade multi-tarefa. Entretanto, inserir essa capacidade na aplicação não é tarefa fácil. Na alteração, foi necessário a troca de processos filhos por *threads*, e uma lista de *threads* em execução substitui a de processos. Os dados passaram a ser compartilhados pela *thread* principal e descendentes, com alocação/desalocação de memória e acesso às variáveis globais protegidos pelas primitivas de sincronização *mutex* e variável condicional. O uso de *threads* no cliente 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 (sob gerência de *threads*) para executar comandos. Já no SP o uso de *threads* é para obter eficiência ao atender vários clientes ao mesmo tempo.

Uma dificuldade existente com a programação multi-tarefa é que nem todas as *system calls* usadas em programação seqüencial podem ser usadas concorrentemente. Somente as funções classificadas como MT-*safe* (*Multithread safe*) podem potencialmente ser chamadas por uma *thread*, pois elas suportam um nível razoável de concorrência. As demais 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 alto custo no desempenho.

VI - ANÁLISE DE DESEMPENHO

A implementação do MakeD visa, entre outros, a eficiência da aplicação. No entanto, é essencial verificar o seu desempenho com relação ao GNU *Make* original e a outra aplicação semelhante como o Dmake, para analisar o ganho real com a utilização da rede e dos métodos empregados na implementação.

Os testes realizados são compilações, usando máquinas semelhantes[8], de uma aplicação composta de 150 arquivos fonte. Na primeira fase, a aplicação foi construída várias vezes usando uma única estação e com o emprego do GNU *Make*, MakeD e Dmake, variando o número de tarefas concorrentes (J) a cada processo de *make*. Na segunda fase, foi utilizado o

8. Todas as estações (Sun SPARCstation 4 - 1 CPU de 110 MHZ, 32MB de RAM e Sistema Operacional Solaris 2.5) estão numa mesma rede e compartilham o mesmo sistema de arquivos, localizado num servidor presente noutra rede.

MakeD e Dmake em grupos de quantidade variável de estações e com diferentes valores de J para cada grupo, gerando a mesma aplicação diversas vezes.

A Tabela 1 apresenta os tempos (em minutos) de cada processo de *make*. O número junto ao nome do *make* indica a quantidade de máquinas utilizada, e J as tarefas concorrentes. O Gráfico 1 representa os percentuais relativos ao GNU *Make* com J=1.

J	GNUMake	MakeD1	MakeD2	MakeD3	MakeD4	MakeD5	MakeD6
1	11:37.6	11:43.0	09:13.8	06:21.1	04:55.4	04:24.5	03:58.8
2	10:46.5	10:53.2	07:27.3	05:46.5	04:08.9	03:59.4	03:20.2
4	08:41.1	09:34.5	06:50.8	04:43.0	03:24.3	03:03.0	02:33.7
6	08:02.7	08:32.3	05:35.1	04:15.0	03:13.2	02:51.4	02:19.7
8	07:54.8	08:09.9	05:34.2	04:11.0	03:05.2	02:55.5	02:17.5

J	GNUMake	Dmake1	Dmake2	Dmake3	Dmake4	Dmake5	Dmake6
1	11:37.6	16:13.0	10:55.9	06:24.7	05:46.3	04:31.1	04:03.0
2	10:46.5	12:17.9	09:29.0	04:47.9	04:34.0	03:23.9	03:04.6
4	08:41.1	10:37.4	08:15.8	04:43.1	04:12.4	03:08.8	03:25.6
6	08:02.7	10:35.8	08:12.1	04:08.5	03:50.7	02:58.6	02:48.9
8	07:54.8	10:38.6	07:59.7	04:53.0	03:11.7	03:14.9	02:48.2

Tabela 1: Tempos reais dos processos de *make*

No caso de compilações locais o GNU *Make* possui melhor desempenho para todos os valores de J, devido ao *overhead* adicional dos *makes* distribuídos. O Dmake não possui um bom desempenho pelo fato de usar a comunicação Cliente/Servidor com `rsh` (*spew*, *rspew*) dentro da própria estação. No MakeD, o *overhead* é causado principalmente pelo uso de *threads*.

Observa-se que nas execuções locais e distribuídas, o uso da concorrência é vantajoso e sempre fornece resultados melhores que em compilações seqüenciais. Entretanto, para valores de J acima de seis no MakeD, e quatro no Dmake, a relação ganho versus degradação dos sistemas não é viável. Foi observado que acima de doze tarefas simultâneas, em cada uma do grupo de seis estações, é gerado alto tráfego na rede, sobrecarga nos sistemas e excessiva demanda por leitura/escrita no servidor de arquivos.

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

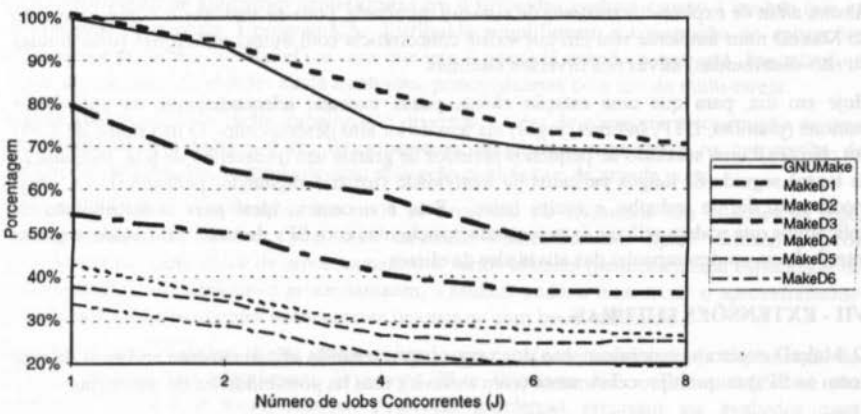

Gráfico 1: MakeD - Porcentagem Relativa ao GNU Make com J=1

exemplo, é mais vantajoso executar quatro módulos em quatro estações do que executá-los em duas (dois em cada).

O gráfico 1 mostra que o GNU *Make* consegue reduzir 1/3 do tempo através da concorrência local. Com a combinação da distribuição e concorrência no MakeD e Dmake, consegue-se ganhos de até 80% e 75%, respectivamente, usando J=8 em cada uma das seis máquinas. Em relação ao Dmake, o MakeD consegue um ganho mais uniforme com o acréscimo de J e estações. O Dmake tem sensível melhora quando empregado com duas e três estações, pois nesses casos o seu *overhead* é distribuído entre os sistemas. Entretanto, o próprio autor alerta sobre as limitações da aplicação e recomenda usá-la com poucas estações e baixo grau de concorrência. O gráfico 2 mostra que o desempenho do MakeD é, na maioria das vezes, superior ao Dmake

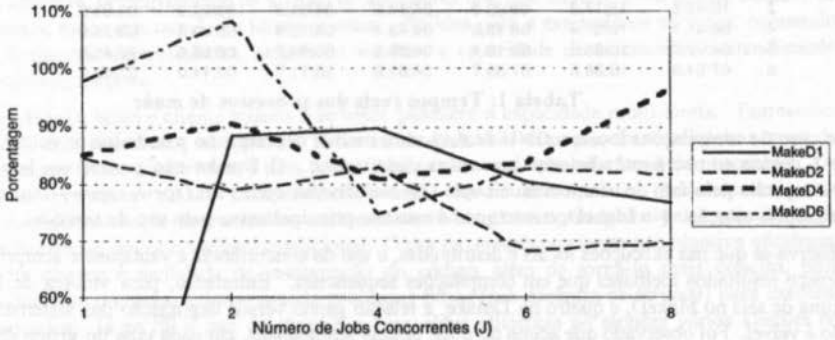

Gráfico 2: MakeD - Porcentagem relativa ao Dmake com mesmo J

Com o MakeD, foi possível mostrar o ganho real de tempo distribuíndo as tarefas entre os diversos sistemas na rede. No entanto, o desempenho foi medido considerando os recursos como sendo utilizados somente por esta aplicação. É preciso um 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.

Hoje em dia, para que uma estação cliente possa executar adequadamente os aplicativos comuns (planilha, DTP, Internet, jogos) ela tem de ter alto desempenho. O uso típico de CPU em clientes é uma sucessão de pequenos períodos de grande uso (redesenho de tela, formatação de texto) seguido de longos períodos de inatividade (usuário digitando, pensando). A carga média num dia de trabalho é muito baixa. Este é o cenário ideal para a distribuição de aplicativos que podem utilizar da ociosidade dos clientes com SPs de baixa prioridade e pouca interferência no desempenho das atividades de cliente.

VII - EXTENSÕES FUTURAS

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

- Embora os servidores 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 invocações não compartilham informações, podendo atualizar um projeto de forma não ótima.

- Para grandes projetos, esforço considerável é gasto por aplicações *make* para análise do `makefile`, montagem do grafo e verificação de dependências. Guardar essas informações de modo persistente poderia acelerar significativamente o processamento dos *targets*.
- 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 proposta já em estudos prevê abordar esses problemas incluindo no MakeD um servidor *make* multi-usuário (MakeDM), responsável por cadastrar, gerenciar e escalonar um número arbitrário de projetos definidos por `makefiles`. Usuários (possivelmente distribuídos) teriam sessões no novo *make* distribuído, cadastrando seu projeto e solicitando o processamento dos *targets* até o final da sessão.

Problemas interessantes ocorrem com usuários simultâneos do 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, adaptar versões privadas modificáveis envolve esforço considerável de reescrita. A proposta de *Shadow Objects* [Dru95] resolve o problema a nível de linguagem de programação, permitindo definir objetos com componentes públicos e privados. Componentes públicos são imediatamente compartilháveis, enquanto os privados são acessíveis se o objeto é parametrizado com um identificador (*shadow tag*). Mecanismos transparentes alocam e liberam os componentes privados dinamicamente. Clareza e economia de memória deverão compensar em muito o *overhead* da gerência (invisível) de alocação.

VIII - CONCLUSÃO

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. O MakeD mostra que isto é possível usando o que está disponível no Unix, apesar das dificuldades ainda existentes, principalmente com uso da multi-tarefa.

Os resultados positivos deste trabalho são diversos. Além de conseguir a construção de uma ferramenta de uso prático e em funcionamento, adquirimos experiência na construção de aplicações distribuídas, partindo de uma aplicação complexa e de grande porte.

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 capacidade de processameto). Distribuindo o processamento, o MakeD poderia maximizar o aproveitamento da rede utilizando os clientes normalmente ociosos ou com baixa utilização.

Com essas possibilidades, já está sendo realizado um estudo de viabilidade da implantação do MakeD também nas plataformas Windows 95 e Windows NT. Tais plataformas são populares numa rede local, mas os possíveis problemas precisam ser avaliados mais detalhadamente, principalmente os que se referem à multi-tarefa no Windows 95.

Os potenciais benefícios alcançados com o MakeD justificam a continuação da pesquisa nessa área e na linha proposta de Shadow Objects.

AGRADECIMENTOS

Gostaríamos de agradecer a Jeanne Dobgenski, Carlos Furuti, Fábio Bueno e Sheila Maceira pelas críticas e sugestões no presente artigo, e ao Fernando Silveira pela confecção das figuras.

REFERÊNCIAS

[Acc86] M. Acceta, R. Baron, et. ali. Mach: A New Kernel Foundation for UNIX Development. Tech. Rep. - Carnegie Mellon University, agosto 1986.

[Beg94] Adam Beguelin et. ali. PVM 3 User's Guide and Reference Manual, ORNL/TM-12187. Tech. Rep. - Oak Ridge National Laboratory, May 1994.

[Che88] D. R. Cheriton. The V Distributed System. *CACM, Vol. 31*, 3, 314-333, março 1988.

[Cho90] Overview of the CHORUS® Distributed Operationg Systems. Tech. Rep. Chorus Systèms CS/TR-09-25, April 1990.

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

[Dru95] R. Drummond. Shadow Objects – Compartilhando Estruturas em Programas Multi-usuário. Palestra Interna, Lab. A-HAND, IC, UNICAMP, novembro 1995.

[Dru96] R. Drummond, C. Gonçalves J. LegoShell: Linguagem Visual de Programação Distribuída. Anais do XIV SBRC, Fortaleza, CE, maio 1996.

[Duk94] D. W. Duke, et ali. Research Toward a Heterogêneous Networked Computing Cluster: The DQS Version 3.0. Tech. Rep. SCRI, Florida University, March 1994.

[Dwy94] F. Dwyer. Using the DQS Parallel Make Utility. Tech. Rep. SCRI, Florida University, January 1994.

[Fel79] S. I. Feldman. MAKE – A Program for Maintaining Computers Programs. *Software–Practice and Experience*, Vol. 9, 255-265, 1979.

[Fle89] C. J. Fleckenstein and D. Hemmendinger. Using a Global Name Space for Parallel Execution of UNIX Tools. *CACM, Volume 32*, 9, September 1989.

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

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

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

[Loi96] M. Loitz. Código Fonte do Lmake. Tech. University Braunschweig, Germany,1996.

[Mul90] S. J. Mullender, A. S. Tenenbaum, et. ali. Amoeba A Distributed Operationg System for the 1990s. *Computer*, pp. 44-53, May 1990.

[OMG91] Object Management Group, Inc. The CORBA and Specification. OMG Document Number 91.12.1, John Wiley & Sons, Inc., 1991.

[OMG92] Object Management Group, Inc. Object Management Architecture Guide. OMG TC Document 92.11.1, John Wiley & Sons, Inc., 1992.

[OSF90] Distributed Computing Environment Overview. White paper - OSF, 1990.

[Rev92] L. S. Revor. DQS User's Guide. Argonne National Laboratiory, September 1992.

[Sta95] R. M. Stallman and R. McGrath. GNU Make User's Guide. FSF, Inc., April 1995.

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

[Sun88] Sun Microsystems, XDR: External Data Representation Standard, RFC1057, june 1988.

[Sun94] Network Interfaces Programmer's Guide. Sun Microsystems, Inc., Mountain View (CA), August 1994.