Programando em "Linda": Transformações de Espectro

Dorgival O. Guedes Neto
Osvaldo S. F. Carvalho
Antônio Gilberto M.Carvalho[0]
Eduardo Ferreira Loures
Silvânia de Avelar Pinto
Departamento de Ciência da Computação — UFMG
Caixa Postal 702 — 30161 Belo Horizonte - MG
{dorgival,vado,agmc,loures,silvania}@dcc.ufmg.br *

Resumo

Este artigo descreve uma experiência de implementação de um programa distribuído para transformações de espectro utilizadas em problemas de síntese de som utilizando-se a linguagem **"Linda"**. *Apresenta-se o processo de desenvolvimento de um programa em* **"Linda"** *a partir da idéia inicial abordando-se alguns métodos de estruturação dos processos envolvidos, a fim de demonstrar o poder de expressão e simplicidade de utilização da linguagem.*

Abstract

This paper describes the implementation experience of distributed program for spectrum transformations used commonly in sound sinthesis using the **"Linda"** *language. The development process of a* **"Linda"** *program is presented based on the initial idea using some different methods for structuting the component processes in order to demonstrate the language's expression power and simplicity.*

*Este trabalho conta com o apoio da Telebrás — contrato no. 416/91 e do CNPq
[0]Professor da Escola de Música da UFMG, mestrando em Ciência da Computação

1 Introdução

O desenvolvimento de programas distribuídos costuma ser uma tarefa penosa, exigindo conhecimentos e cuidados muito além dos usualmente necessários no desenvolvimento de programas seqüenciais comuns. Essa dificuldade se torna ainda maior para profissionais de outras áreas que não a Ciência da Computação, como engenheiros e físicos, talvez os maiores usuários em termos de volume de processamento exigido.

Tal dificuldade advém muitas vezes do modelo de paralelismo utilizado, exigindo que o programador se familiarize com primitivas complexas, forçando-o a utilizar construções adaptadas à arquitetura do sistema distribuído, porém distantes do seu modelo de aplicação[PB90].

Sob esse aspecto, **"Linda"** [CG89b] se apresenta como uma linguagem para desenvolvimento de sistemas distribuídos com um nível de abstração bem maior que o usual, o que simplifica sua utilização. Com base em apenas quatro primitivas básicas pode-se exprimir diversos modelos de paralelismo[GB86].

Para exemplificar a utilização da linguagem, desenvolveu-se uma aplicação para geração de seqüências de transformação de espectro para conversão de um timbre sonoro em outro.

A aplicação apresenta uma granularidade mais fina que a desejável para implementação em um sistema distribuído sobre uma rede local de estações de trabalho, porém considerou-se interessante o trabalho do ponto de vista de um maior conhecimento do processo de desenvolvimento de uma aplicação distribuída com **"Linda"**.

Nas seções a seguir apresenta-se o modelo básico do sistema utilizado, a linguagem **"Linda"**, um maior detalhamento do algoritmo a ser abordado, os passos seguidos durante a implementação e outras possibilidades de estruturação do problema também consideradas.

2 O problema proposto

Nos últimos tempos vem se tornando um recurso usual em animações para cinema e televisão as "metamorfoses de imagens", onde o observador vê uma imagem transformar-se em outra em um processo contínuo. Um problema semelhante existe no campo da música: dados dois sons de timbres bem definidos, como uma voz humana e um saxofone, gerar uma sequência sonora em que um som se transforme gradualmente no outro, sem que o ouvinte seja capaz de perceber mudanças abruptas.

No caso de imagens, é sabido que não basta uma interpolação linear pixel a pixel entre as duas imagens. Isso resultaria no efeito de "shading", onde uma imagem vai desaparecendo enquanto a outra surge, sem que haja uma continuidade nos contornos dos objetos envolvidos. Para o efeito desejado é necessário estabelecer correlações entre áreas das duas figuras, indicando para o computador que a área dos olhos de um chimpanzé deve ser relacionada à área dos olhos de uma pessoa na outra imagem, mesmo que ocupem regiões diferentes nas imagens respectivas.

Algo semelhante ocorre no campo da música: não basta que se tome as formas de onda de dois timbres diferentes e se gerem quadros com formas de onda obtidas a partir da interpolação linear entre os elementos de cada uma. O efeito para o ouvinte seria como se o volume de uma fonte de som fosse continuamente abaixado enquanto se aumentasse o volume da outra, sem uma continuidade entre ambos. Para obter-se um resultado agradável é necessário que se trabalhe no domínio das freqüências componentes dos sons:

- Dados os dois sons que se deseja "integrar", deve-se primeiramente determinar o conteúdo harmônico de ambos, o que pode ser feito com um algoritmo como a transformada rápida de Fourier (FFT[Bri74]).

- Deve-se então proceder à interpolação linear das amplitudes e fases de cada harmônico, determinando um espectro completo interpolado para cada quadro.
- Finalmente, de posse do conteúdo espectral de cada forma de onda intermediária deve-se proceder à geração dessas formas de onda no domínio do tempo a fim se produzir uma saída audível.

Tal processo aceita várias estruturações possíveis, tanto em relação aos algoritmos disponíveis quanto na ordenação das atividades envolvidas. Antes porém de se discutir essas possibilidades, deve-se avaliar com mais detalhe o sistema a ser utilizado na implementação.

3 "Linda"

"Linda" foi proposta em 1982[GB82] e hoje já existem implementações para diversas arquiteturas e linguagens base. No presente caso utilizou-se a implementação desenvolvida na UFMG com algumas alterações em relação ao modelo original[GC92].

3.1 Elementos básicos

"Linda" se baseia na manipulação de conjuntos ordenados de campos denominados tuplas, tendo cada campo um tipo e valor associado. Completando a tupla destaca-se um campo do tipo seqüência de caracteres denominado chave da tupla, utilizado na sua manipulação. Na implementação em linguagem C uma tupla é identificada por um `struct` contendo os campos, e um vetor de `char` de 16 posições para a chave.

As tuplas são mantidas em espaços de tuplas (ET) tipados, podendo cada aplicação definir tantos ET quantos sejam necessários. Em aplicações normais, basta que se declare cada espaço de tuplas com uma macro adequada como a seguir, para que ele seja criado e aberto para cada processo componente da aplicação quando da sua carga. Uma vez criado e aberto, cada ET passa a ser referenciado por um descritor.

```
#include <lindaAPI.h>
typedef struct {
  /* ... */                                  /* Descricao dos campos da tupla    */
} tuple_t;
int     tsd;                                 /* Descritor de um espaco de tuplas */
BEGIN_TS_DECL                                /* Definicao de um ET               */
  TS_INIT( tsd, tuple_t, "TS Access Key" );
END_TS_DECL
```

Cada aplicação tem um ET base aberto automaticamente, identificado pelo descritor 0 e associado a tuplas do tipo `int`.

A operação básica executada sobre tuplas é o "casamento". Uma tupla casa-se com um padrão (um modelo de tupla) em um espaço de tuplas se e somente se a tupla e o padrão possuem a mesma chave e ambos possuem campos do tipo definido para o ET.

3.2 As quatro operações

Uma vez definidas as condições para o casamento de tuplas, as quatro operações que caracterizam **"Linda"** podem ser então definidas. Para isto, deve-se considerar além das declarações anteriores as seguintes no programa em **"C-Linda"**:

```
int     eval_function( int );
tuple_t tuple;

BEGIN_EVAL_DECL                          /* Definicao de um processo linda  */
  eval_function;
END_EVAL_DECL
```

As operações são então:

`out(tsd,"Chave",&tuple)` deposita no espaço de tuplas identificado por tsd uma tupla com chave `"Chave"` e demais campos contidos na variável `tuple`. O processo que a executa não é bloqueado.

`in(tsd,"Acesso",&tuple)` uma tupla identificada por uma chave `"Acesso"` é retirada do espaço de tuplas. Os demais campos da tupla retirada são copiados para a variável `tuple`. Se não há uma tupla disponível com a chave indicada o processo é suspenso até que uma surja, quando o processo é liberado. Se há várias tuplas com a chave especificada, a escolha é não determinística.

`rd(tsd,"Ola'!",&tuple)` semelhante ao `in()`, porém a tupla não é removida do espaço de tuplas após o casamento, mas apenas tem seus campos copiados para a variável `tuple`.

`eval(tsd,"Resultado",eval_function,param)` gera uma "tupla viva" que leva à criação de um novo processo, o qual executará as operações definidas na função `eval_function` com um parâmetro inteiro `param`. Ao término da execução uma tupla com chave `"Resultado"` deverá ser depositada no ET identificado por `tsd` com um único campo (inteiro) contendo o valor de retorno da função. O processo pai, que executou o `eval()`, prossegue sem esperar por seus filhos.

Com base nestas operações, pode-se verificar que o modelo permite acesso concorrente a dados para leitura, porém não há nenhuma operação que permita a alteração de dados enquanto no espaço de tuplas. Uma alteração de uma tupla só é possível através de sua remoção do espaço e sua substituição por outra com novo valor.

3.3 O processo de desenvolvimento e execução

Uma aplicação **"Linda"** é desenvolvida como um programa em linguagem C único, sendo cada processo definido como uma função que pode receber e/ou retornar um inteiro ao final. Para se ter acesso às funções que compõem as primitivas basta que se inclua o arquivo lindaAPI.h nos arquivos que as utilizem, e "linkar" a biblioteca adequada.

O ambiente de execução é configurado pelo usuário por um arquivo de especificação com o seguinte formato:

- Uma linha indicando o processo iniciador da aplicação, o qual será o único a executar a função `lmain` — substituta da função `main` usual da linguagem C — e seus parâmetros de entrada.
- Uma linha para cada "servidor de `eval()`" a ser disparado para a aplicação. Cada servidor é capaz de executar um `eval()` por vez. O programa disparado é o mesmo utilizado como iniciador, cabendo ao sistema de tempo de execução identificar o papel de cada novo processo disparado.

4 Estruturação da aplicação

O problema de transformação de timbres pode ser expresso da seguinte forma: dados como entrada dois ou mais timbres definidos por seus conteúdos espectrais, deseja-se obter como saída uma seqüência de quadros contendo ondas no domínio do tempo que componham uma transformação gradual de um timbre no seguinte.

Há dois passos bem definidos nesse processo: para cada quadro deve-se definir seu conteúdo espectral por interpolação com os extremos e em seguida gerar a onda no domínio do tempo. O processo de interpolação segue métodos convencionais, sem maiores problemas. O processo de geração da onda no domínio do tempo em função do espectro porém pode receber vários tratamentos, como a Transformada Rápida de Fourier, sem dúvida o método mais aconselhado do ponto de vista de ordem de complexidade do algoritmo.

Tendo-se em vista que o principal objetivo deste trabalho é analisar o poder de expressão de paralelismo da linguagem, optou-se pelo método de geração direta através da aplicação do conceito de séries de Fourier convencional: a onda é gerada como uma soma de senóides de freqüências crescentes. Tal processo permite que se explore mais facilmente o paralelismo na geração de um mesmo quadro, enquanto, devido ao algoritmo adotado, a FFT não oferece um modelo de paralelismo interno adequado a um sistema paralelo de grão grosso.

Pelo método adotado, uma onda no domínio do tempo pode ser obtida a partir de um espectro dado com as fases e amplitudes de cada componente de freqüência envolvido por uma expressão do tipo:

```
for ( tempo=0; tempo<PERIODO; tempo++ )
{
  onda[tempo] = 0.0;
  for ( frequencia=0; frequencia<FREQ_MAX; frequencia++ )
    onda[tempo] += espectro[frequencia].amplitude *
                   sin( frequencia * tempo + espectro[frequencia].angulo );
}
```

Uma das formas de se paralelizar esse processo seria gerar completamente a forma de onda de cada freqüência em paralelo, para somá-las em seguida gerando o quadro. Um estudo mais detalhado permite aumentar ainda mais o paralelismo, reduzindo ao mesmo tempo o processamento envolvido. Segundo a proposta anterior, para cada quadro, todos os pontos de um harmônico de uma dada freqüência serão recalculados, envolvendo custosas operações de cálculo do valor do seno para cada ponto. Para os vários quadros, as alterações sobre a senóide básica que representa um harmônico são uma multiplicação para se determinar a amplitude, e um deslocamento para se levar em conta as variações do ângulo de fase.

O processo pode ser então dividido em três partes:

- Geração das formas de ondas dos harmônicos "normalizados", isto é, com amplitude unitária e ângulo de fase nulo.
- Geração dos harmônicos "ajustados" para cada quadro, obtendo como entrada os valores de amplitude e deslocamento do harmônico e a forma de onda normalizada, e produzindo a nova forma de onda ajustada.
- Geração final do quadro, pela obtenção e soma de todos os harmônicos componentes de uma onda.

Optou-se então por gerar vários processos encarregados de cada uma destas tarefas, onde todos os processos de mesmo tipo atualizam um seqüenciador para se definir qual o próximo harmônico ou quadro a ser tratado.

É necessário destacar que a aplicação, em qualquer forma de paralelização que venha a ser adotada, apresenta uma alta taxa de comunicação em relação ao volume de processamento envolvido, não sendo esperado assim um grande "speed-up" em um sistema de grão grosso como o utilizado. Não obstante, o trabalho foi movido pelo desejo de se investigar as características da linguagem na expressão do paralelismo mais que na busca de resultados de alto desempenho nessa primeira aplicação.

4.1 Espaços de tuplas necessários

Pode-se identificar então as estruturas de dados necessárias para a comunicação entre processos:

4.1.1 Seqüenciadores

Serão necessários três contadores para ordenação das tarefas. Cada contador pode ser implementado por uma tupla, tendo como campo apenas um inteiro contendo o valor corrente do contador. Cada contador pode ser reconhecido pela sua chave:

SEQ_NORM_HARM para controlar a geração dos harmônicos normalizados. Deve variar de zero até o número máximo de harmônicos.

SEQ_ADJ_HARM para fazer com que os programas "ajustadores" gerem os harmônicos ajustados em seqüência. Variará de zero até o número de quadros.

SEQ_FRAME para garantir que os somadores gerem um quadro por vez em seqüência. Variará de zero até o número de quadros especificado.

Como serão todos os seqüenciadores inteiros, pode-se utilizar o ET base da aplicação para manipulá-los, não sendo necessário que se declare um novo espaço.

4.1.2 Ambiente de execução

Deseja-se que a aplicação seja configurável com relação ao número de harmônicos a serem utilizados na geração das ondas, número de quadros a serem gerados, número de pontos por quadro, etc. Como esses valores não são conhecidos em tempo de compilação, é necessária uma maneira de comunicá-los aos processos disparados por eval. Para isso, cria-se um **struct** contendo todos os parâmetros de configuração da aplicação.

4.1.3 Fatores de normalização

Não se deve esquecer que na primeira fase da aplicação são calculados por interpolação os valores de amplitude e ângulo de fase para cada harmônico em cada quadro. Os processos "ajustadores" devem obter essa informação para cada harmônico a ser gerado. Para isso utilizam-se tuplas contendo três campos: número do harmônico, amplitude e deslocamento angular, e tendo por chave o número do quadro afetado. A definição do tipo das tuplas e do ET seria da forma:

```
typedef struct {
          int    harmonic;
          double amplitude;
          double angular_d;
        } factor_t;
TS_INIT( harm_factor_tsd, "HARM_FACTOR", factor_t )
```

4.1.4 Formas de onda

Deve-se definir um tipo de tupla que armazene a forma de onda de cada harmônico normalizado, cada harmônico já ajustado para um quadro, e cada quadro completamente gerado. Tais tuplas devem conter um vetor com o valor da amplitude da onda para cada ponto do intervalo considerado.

Os três tipos de ondas citados poderiam ser manipulados em um único ET, por serem do mesmo tipo. Porém, uma divisão em três espaços distintos permite uma melhor organização da aplicação, simplificando a escolha das chaves de acesso: harmônicos normalizados podem ter por chave o número do mesmo, enquanto harmônicos ajustados e quadros completos podem utilizar o número do quadro.

Note-se que não há necessidade de se identificar na chave para tuplas de harmônicos ajustados o número do harmônico, uma vez que todos receberão o mesmo tratamento, sendo simplesmente somados.

Nas definições a seguir, N_INTERV indica um valor limite para o número de pontos utilizados na representação da onda:

```
typedef double  wave_t[N_INTERV];
TS_INIT( norm_harm_tsd, "NORMALIZED_HARM", wave_t )
TS_INIT( adj_harm_tsd,    "ADJUSTED_HARM", wave_t )
TS_INIT( frames_tsd,      "FRAMES",        wave_t )
```

4.2 Gerador de harmônicos normalizados

Os processos encarregados de gerar as formas de onda dos harmônicos normalizados devem seguir a seguinte seqüência:

- Determinar o harmônico a ser gerado com base no seqüenciador.
- Preencher um vetor com os valores de amplitude para cada intervalo.
- Depositar o vetor normalizado no ET apropriado.

O código final utilizado na implementação é apresentado a seguir:

```
01 int norm_harm()
02 {
03   int    i, harmon, n_harmon, n_interv, work_done=0;
04   double angular_inc, angle;
05   wave_t normalized_wave;
06   /* Inicialmente, obtem o ambiente da aplicacao */
07   in( environ_tsd, "ENVIRONMENT", &environment );
08   n_interv = environment.n_interv; n_harmon = environment.n_harmon;
09   angular_inc = 2*M_PI/(n_interv-1);
10   for (;;) {
11     in(  0, "SEQ_NORM_HARM", &harmon );
12     harmon++;
13     out( 0, "SEQ_NORM_HARM", &harmon );
14     if ( harmon > n_harm ) break;
15     for ( angle=0.0,i=0; i<n_interv; i++, angle += angular_inc )
16       normalized_wave[i] = sin( harmon * angle );
17     out( norm_harm_tsd, itokey(harmon-1), normalized_wave );
18     work_done++;
19   }
20   return work_done;
21 }
```

A função `itokey()` é oferecida pelo sistema **"Linda"** para geração da chave da tupla a partir de um inteiro. Funções semelhantes existem para cada tipo básico da linguagem C.

A variável `work_done` é incrementada a cada tarefa executada pelo processo, e seu valor definido como valor de retorno. Isso significa que, ao terminar, cada processo normalizador gerará uma tupla contendo o número de tarefas executadas. Tal recurso foi utilizado para se analisar a distribuição de tarefas entre os vários nodos do sistema.

O `in()` na linha 07 permite que se obtenha a variável de definição do ambiente, necessária nesse caso para determinar a resolução desejada para as ondas no domínio do tempo (`n_interv`) e o número de harmônicos a serem gerados (`n_harmon`).

O centro do programa é implementado por um `for` sem parâmetros. A condição de terminação do processo é dada na linha 14. Quando o seqüenciador obtido excede ao número de harmônicos desejado, a tupla é novamente depositada no ET antes do término do processo. Isso garante que a tupla permaneça disponível para ser utilizada pelos demais normalizadores.

4.3 Escalador

Os processos responsáveis pela geração da forma de onda de cada harmônico componente de um dado quadro o fazem aplicando um fator de escala e um deslocamento, se necessário, às formas de onda geradas pelos normalizadores, daí o nome de escaladores, ou ajustadores.

Para cada harmônico a ser escalado corresponderá uma tupla no ET descrito por `harm_factor_tsd`. Cada escalador deverá inicialmente obter uma dessas tuplas, definindo assim sua próxima tarefa. Com esses dados, ele deve executar um `rd()` para a forma de onda normalizada do harmônico correspondente, e aplicar sobre cada elemento o fator de escala correspondente e um deslocamento, gerando um novo vetor de forma de onda que é depositado no ET de harmônicos ajustados.

```
00 int adjust_harm()
01 {
02   int      i, harmon, sequencer, frame, theta, theta_compl;
03   wave_t   normalized_wave, adjusted_wave;
04   factor_t factor;
05   int      n_interv, work_done = 0;
06   double   angular_d, aux;
07   in( environ_tsd, "ENVIRONMENT", &environment );
08   n_interv = environment.n_interv;
09   for (;;) {
10     in(  0, "SEQ_ADJ_HARM", &sequencer );
11     frame = (sequencer++)/n_harm;
12     out( 0, "SEQ_ADJ_HARM", &sequencer );
13     if ( frame >=  n_frames ) break;
14     in( harm_factor_tsd, itokey(frame), &factor );
15     rd( norm_harm_tsd,   itokey(factor.harmonic), normalized_wave );
16     theta       = (factor.angular_d/(2*M_PI))*(n_interv-1);
17     if (theta<0) theta += (n_interv-1);
18     theta_compl = (n_interv-1) - theta;
19     for (i=0;i<theta_compl;i++)
20       adjusted_wave[i] = factor.amplitude * normalized_wave[i+theta];
21     for (i=theta_compl;i<n_interv;i++)
22       adjusted_wave[i] = factor.amplitude * normalized_wave[i-theta_compl];
23     out( adj_harm_tsd, itokey(frame), adjusted_wave );
24     work_done++;
25   }
26   return work_done;
27 }
```

As linhas 10 a 13 implementam o controle de alocação de tarefas e a terminação em função do número de harmônicos total. Desse valor obtém-se o número do quadro a ser tratado. O número do harmônico dentro do quadro não é necessário, uma vez que o descritor da tarefa já fornece essa informação (linha 14). Uma vez de posse do número do harmônico, basta então obter sua forma de onda normalizada (linha 15).

As linhas 16 a 18 determinam o deslocamento desejado e as linhas 19 a 22 realizam o ajuste de escala e o deslocamento. Finalmente, na linha 23 a forma de onda é depositada no ET apropriado.

4.4 Somador

Uma vez geradas as formas de onda de cada harmônico de um quadro, resta a tarefa de coletar e somar tais ondas. Os processos somadores devem então definir uma tarefa, isto é, obter o número de um quadro ainda não gerado e passar a obter todas as formas de onda dos espectros nele contidos. Como no caso dos escaladores, não é necessário se especificar o número de cada harmônico que se deseja somar. Basta que se garante que um número de harmônicos igual ao especificado foi removido do ET tendo por chave a identificação do quadro em questão.

Obtidas todas as formas de onda dos harmônicos, basta depositar a forma de onda do quadro gerado no ET de resultados.

```
00 int add_harms()
01 {
02   int    i, harmon, frame_no, n_harm, n_interv, work_done=0;
03   wave_t adjusted_harm, frame;
04   in( environ_tsd, "ENVIRONMENT", &environment );
05   n_harm = environment.n_harm; n_interv = environment.n_interv;
06   for (;;) {
07     in( 0, "SEQ_FRAME", &frame_no );
08     frame_no++;
09     out( 0, "SEQ_FRAME", &frame_no );
10     if ((frame_no--)>n_frames) break
11     for (i=0;i<n_interv;i++)
12       frame[i] = 0.0;
13     for (harmon=0;harmon<n_harm;harmon++) {
14       in( adj_harm_tsd, itokey(frame_no), adjusted_harm );
15       for (i=0;i<n_interv;i++)
16         frame[i] += adjusted_harm[i];
17     }
18     out( frames_tsd, itokey(frame_no), frame );
19     work_done++;
20   }
21   return work_done;
22 }
```

Como observado, as linhas 14 a 16 realizam a obtenção de cada forma de onda e sua adição ao quadro que está sendo formado. Como nos normalizadores, apenas o número do quadro é utilizado para identificar os harmônicos ajustados, uma vez que o tratamento a eles dispensado independe completamente de seu número.

4.5 Processo "gerente"

O processo gerente tem por função inicializar os dados para a aplicação e controlar a execução dos demais processos. É ele o procedimento executado no programa inicial da aplicação. Com base nos dados obtidos de um arquivo de configuração contendo o número de ondas básicas a se considerar, número de quadros a ser gerado entre cada par de ondas básicas, número de

harmônicos desejado no processamento, número de pontos a serem gerados para as ondas no domínio do tempo, amplitude dos harmônicos normalizados e seqüência de ondas desejada, inicia-se o processamento. Como parâmetros iniciais o processo gerente recebe o número de processos de cada tipo que devem ser disparados.

A primeira operação é então iniciar os normalizadores. Para isso deve-se depositar no ET base o seqüenciador por eles utilizado, bem como aqueles que virão a ser utilizados pelos demais tipos de processos. Em seguida deve-se gerar uma tupla com a descrição do ambiente para cada trabalhador e finalmente executar o `eval()` que os disparará.

Enquanto os normalizadores fazem seu trabalho deve-se gerar as tuplas que definirão as tarefas dos escaladores. Para isso o processo gerente lê de arquivos gerados anteriormente o conteúdo espectral de cada par de ondas base especificado. Para cada par procede-se então à interpolação linear das amplitudes e ângulos de fase para cada harmônico considerado. Para cada quadro deposita-se no ET de fatores de escala o fator correspondente para cada harmônico.

Isso feito, passa-se então ao disparo dos normalizadores, também com uma tupla de ambiente para cada processo. Pode-se nesse instante disparar também os somadores, para que iniciem seu processamento tão logo tuplas contendo ondas normalizadas se tornem disponíveis.

O processo principal pode então passar a coletar as tuplas contendo os quadros gerados e armazená-las em arquivo. Para evitar o aumento de complexidade do código devido às operações de entrada e saída para manipulação de arquivos, apresenta-se a seguir apenas um esboço da estrutura do programa principal:

```
00 int lmain( argc, argv )
01     int   argc;
02     char* argv[];
03 {
04   /* Declaracao de variaveis */
05   /* ... */
06   /* Obtencao dos valores de configuracao: */
07   /*   n_transf: numero de transformacoes ( numero de ondas base - 1 ) */
08   /*   n_harmon: numero de harmonicos a serem considerados             */
09   /*   n_interv: numero de pontos no dominio do tempo                  */
10   /*   fr_per_tr: numero de quadros para cada transformacao            */
11   /*   n_normalizers: numero de normalizadore a serem disparados       */
12   /*   n_adjustets:   numero de escaladores                            */
13   /*   n_adders:      numero de somadores                              */
14   /* ... */
15   /* Inicializacao dos sequenciadores */
16   sequencer = 0;
17   out ( 0, "SEQ_NORM_HARM", &i );
18   out ( 0, "SEQ_ADJ_HARM",  &i );
19   out ( 0, "SEQ_FRAME",     &i );
20   /* Disparo dos normalizadores */
21   for ( i=0; i<n_normalizers; i++)
22   {
23     out( environ_tsd, "ENVIRONMENT", &environment );
24     eval( 0, "NORMALIZER", normal_harm, 0 );
25   }
```

```
26  /* Geracao dos fatores de escala */
27  read_spectrum( wave1 )
28  frame = 0;
29  for( i=0; i<n_transf; i++) {
30    read_spectrum( wave2 )
31    for( j=0; j<fr_per_tr; j++ )
32      for( k=0; k<n_harm; k++) {
33        factor.harmonic = k;
34        factor.amplitude = interpolate_amp( wave1, wave2, j, k );
35        factor.angular_d = interpolate_ang( wave1, wave2, j, k );
36        out( harm_factor_tsd, itokey(frame), &tuple );
37      }
38    copy_spectrum( wave1, wave2 );
39  }
40  /* Disparo dos escaladores */
41  for (i=0;i<n_adjusters;i++)
42  {
43    out( environ_tsd, "ENVIRONMENT", &environment );
44    eval( 0, "ADJUSTER", adjust_harm, 0 );
45  }
46  /* Disparo dos somadores */
47  for (i=0;i<adds;i++)
48  {
49    out( environ_tsd, "ENVIRONMENT", &environment );
50    eval( 0, "ADDER", add_harms, 0 );
51  }
52  /* Coleta dos quadros gerados */
53  fd = open("sequencia.dat",O_RDWR|O_CREAT,0600);
54  for (i=0;i<n_frames;i++)
55  {
56    in( frames_tsd, itokey(i), frame );
57    write( fd, frame, sizeof(wave_t) );
58  }
59  close(fd);
60 }
```

4.6 Distribuição dos processos

Pela descrição do procedimento de disparo de processos fica evidente o baixo nível de acoplamento espacial e temporal entre os processos da aplicação, uma característica básica de "**Linda**". Não há exigências ou restrições inerentes à linguagem sobre o instante de início de cada processo envolvido em comunicações por um dado espaço de tuplas. Pela aplicação fica claro que deseja-se que os somadores não sejam disparados antes dos escaladores, e que estes não precedam os normalizadores. Porém pode-se optar por exemplo por disparar processos de todos os tipos ao mesmo tempo, ou todos os processos de um tipo de cada vez. Essas opções permitem várias organizações para os processos ao longo do espaço e tempo.

Para n máquinas disponíveis, duas alternativas básicas seriam:

- n/3 processos de cada tipo. Nesse caso, os três tipos de processos executariam em paralelo, sendo que tuplas produzidas seriam consumidas quase imediatamente pelos processos que delas necessitassem. Porém, por haver apenas uma parte dos processadores alocada a cada tipo de processo, cada processo teria trabalho por mais tempo.

- n processos de cada tipo. Primeiro seriam disparados n normalizadores, que quando terminassem seriam substituídos por n escaladores, e finalmente n somadores. Esse caso mostra bem o desacoplamento permitido por "**Linda**": os dados gerados por um processo só são

entregues a um outro processo iniciado após o término do primeiro. Nesse caso, cada etapa seria distribuída a vários processos idênticos.

Várias outras soluções intermediárias podem ser utilizadas, inclusive com números desiguais de processos para cada etapa. Cada tarefa de um normalizador é de baixo custo computacional e todos eles devem terminar antes que o primeiro quadro seja gerado. Assim sendo, uma possibilidade seria disparar poucos normalizadores, ocupando o restante das máquinas com escaladores. Quando os primeiros terminassem, seriam substituídos por igual número de somadores.

Tais variações se enquadram dentro do problema de balanceamento e alocação de processadores para aplicações distribuídas, e merecem maiores estudos posteriores.

5 Outras possibilidades de implementação

A divisão de processos adotada pode ser definida como um modelo híbrido, de pipeline de "bags of tasks", isso é, o processamento se divide em três operações encadeadas, sendo que cada operação é executada por um conjunto de trabalhadores replicados que vão continuamente obtendo novas tarefas a serem processadas. **"Linda"** é extremamente flexível quanto à forma de expressão do paralelismo, e várias outras opções poderiam ser utilizadas:

5.1 Tuplas vivas

Ao invés de se utilizar processos que executem continuamente um conjunto de instruções para várias tarefas obtidas através de um `in()` uma opção seria definir processos que executassem uma única tarefa em função de seu parâmetro de entrada. Nese caso, a tupla seqüenciadora seria eliminada, e tantos `eval()` quantas fossem as tarefas definidas seriam executados, tendo como parâmetro o identificador de cada tarefa. Esse modelo de divisão de trabalho recebe o nome de "estruturas de dados vivas", uma vez que cada processo termina por produzir uma única tupla como resultado de sua execução[CG89a].

5.2 "Pipeline" puro

Seria o caso quando apenas um processo de cada tipo fosse disponível, executando todos ao mesmo tempo. Como exaustivamente estudado, esse tipo de paralelismo só se aplica bem a problemas que possam ser facilmente divididos em várias etapas consecutivas. Há sempre o problema, entretanto, de limitar-se o grau de paralelismo ao número de etapas identificadas.

5.3 "Bag of tasks" puro

Nesse caso há apenas um processo controlador e todos os demais processos são de mesmo tipo. Um ET desempenha o papel de "bag of tasks", mantendo tuplas cujo conteúdo indica cada um uma tarefa a ser executada. Tal paralelismo apresenta como grande vantagem o fato de se adaptar facilmente ao número de processadores existetente, uma vez que basta disparar-se um novo trabalhador. Uma implementação segundo esse modelo para o problema apresentado contaria com processos trabalhadores que, dado o número de um quadro (tarefa) executariam todo o processamento até a produção do mesmo. A implementação apresentada, como discutido anteriormente, aproveita o paralelismo interno de cada tarefa, definindo conjuntos de trabalhadores para cada etapa.

5.4 FFT

Apesar de não se tratar de uma alteração da divisão do algoritmo, mas sim de um outro algoritmo completamente diferente, algumas considerações devem ser feitas:

É fato que o algoritmo da transformada rápida de Fourier (FFT) apresenta uma complexidade computacional inferior ao utilizado[1]. O principal objetivo desse trabalho, entretanto, foi analisar a utilização de **"Linda"** na expressão de paralelismo entre vários processos diferentes. A paralelização do algoritmo da transformada rápida usualmente resulta em um grão muito fino, com altas taxas de comunicação a cada passo do algoritmo, não sendo adequado para sistemas distribuídos como uma rede local. A única implementação possível seria a nível de "bag of tasks", com processos que executassem cada um uma transformada completa, produzindo um quadro por tarefa.

6 Resultados

A figura 1 apresenta alguns resultados de desempenho para o sistema. As medições foram feitas na rede local do Departamento de Ciência da Computação da UFMG, utilizando-se 9 estações de trabalho Sun Sparcstation SLC. Para reduzir ao máximo a influência de outras tarefas as medições foram realizadas em períodos sem usuários "logados" ou aplicações em "batch".

Foram feitos vários testes variando-se o número de quadros e harmônicos envolvidos, trabalhando-se sempre com n processos idênticos de cada tipo. As legendas indicam por pares QQxHH (p.ex. 30x20) o número de quadros e harmônicos envolvidos.

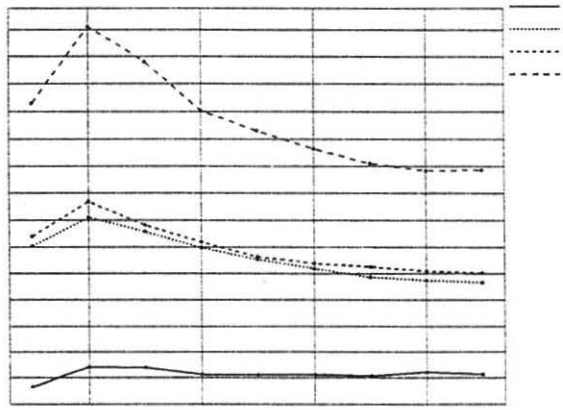

Figura 1: Tempos para várias configurações

Como esperado, os resultados de "speed-up" não foram muito sensíveis devido à natureza da aplicação. Alguns dados interessantes porém podem ser observados.

O tamanho de uma aplicação pode ser relacionado ao produto do número de quadros pelo de harmônicos, uma vez que esse será sempre o número de tarefas dos escaladores. Os resultados de

[1] considerando-se o número de harmônicos igual ao de pontos no domínio do tempo, a complexidade da FFT é $O(n \log n)$, contra $O(n^2)$ do algoritmo utilizado

"speed-up" vão se tornando melhores com o aumento do tamanho do problema. Para pequenas configurações, o ganho é quase nulo, enquanto a situação vai melhorando para aplicações maiores.

Com o aumento do número de processadores envolvidos o tempo cresce nos primeiros pontos, devido ao aumento das necessidades de comunicação do sistema, ainda não compensados pelo grau de paralelismo disponível. Com a entrada de mais processadores, o custo de comunicação passa a ser compensado pelo ganho em paralelismo.

Para dois problemas de mesmo tamanho (90x20 e 30x60) pode-se observar tempos bastante próximos, com pequena vantagem para o primeiro caso, devido ao menor número de tarefas para os normalizadores, reduzindo seu tempo dentro do tempo total da aplicação.

7 Conclusões

"Linda" é uma linguagem extremamente poderosa e simples para o desenvolvimento de aplicações distribuídas. Seu conjunto reduzido de primitivas oferece várias opções de utilização permitindo a expressão fácil de vários tipos de paralelismo. A aplicação apresentada ilustra bem essa simplicidade de desenvolvimento. O algoritmo utilizado foi escolhido devido á possibilidade de obtenção de vários níveis de paralelismo, apesar de problemas de desempenho seqüencial.

Várias opções de estruturação foram discutidas, considerando-se suas particularidades e a adequação da linguagem para cada caso. A opção utilizada na implementação foi discutida em detalhes.

Os resultados de desempenho permitem que se perceba os efeitos do tamanho do problema sobre o "speed-up" alcançado, com resultados mais significativos para problemas maiores.

Referências

[Bri74] E. Oran Brigham. *The Fast Fourier Transform*. Prentice-Hall, Inc, 1974.

[CG89a] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[CG89b] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[GB82] D. Gelernter and A. Bernstein. Distributed communication via global buffer. In *Proceedings Symposium on Principles of Distributed Computing*, pages 10–18. ACM, August 1982.

[GB86] D. Gelernter and A. Bernstein. Distributed data structures in linda. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 236–242. ACM, January 1986.

[GC92] Dorgival O. Guedes and Osvaldo S. F. Carvalho. Um núcleo "linda" para o desenvolvimento de aplicações distribuídas em uma rede unix. In *Anais do X Simpósio Brasileiro de Redes de Computadores*, pages 574–585. SBC, April 1992.

[PB90] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmer? *IEEE Computer*, pages 13–23, December 1990.