UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

PARSMPL E SMPLx
MANUAL TÉCNICO – DESCRIÇÃO E ESTRUTURAS DE DADOS

ROBERTA SPOLON ULSON
REGINA H. C. SANTANA
MARCOS JOSÉ SANTANA
RENATA SPOLON

Nº 132

RELATÓRIOS TÉCNICOS

ICMC USP SÃO CARLOS

São Carlos - SP

Instituto de Ciências Matemáticas e de Computação
Biblioteca

UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
ISSN 0103-2569

Inst Ciên Mat e de Comput - ICMC/USP
30300048057

PARSMPL e SMPLx

Manual Técnico – Descrição e estruturas de Dados

Roberta Spolon Ulson
Regina H. C. Santana
Marcos José Santana
Renata Spolon

Nº 132

RELATÓRIOS TÉCNICOS

São Carlos – SP
Fev./2001

RELATÓRIO TÉCNICO

PARSMPL e SMPLx
MANUAL TÉCNICO - DESCRIÇÃO E ESTRUTURAS DE DADOS

ROBERTA SPOLON ULSON
Departamento de Computação
Universidade Estadual Paulista - UNESP
Av. Luís Edmundo Carrijo Coube, s/n
17033-360 Bauru - SP
(roberta@fc.unesp.br)

REGINA H. C. SANTANA
MARCOS JOSÉ SANTANA
Departamento de Computação e Estatística
Universidade de São Paulo - USP
Av. Dr. Carlos Botelho 1465 - C.P. 668
13560-970 São Carlos – SP
(rcs/mjs@icmc.sc.usp.br)

RENATA SPOLON
Departamento de Computação e Estatística – CCET
Universidade Federal de Mato Grosso do Sul - UFMS
Cidade Universitária, S/N – C. P. 549
79070-900 Campo Grande, MS - BRASIL
(renata@dct.ufms.br)

fevereiro de 2001

SUMÁRIO

1. Introdução 2

2. Extensão Funcional *SMPL*

2.1 Introdução 2

2.2 Estruturas de Dados 4

2.3 Primitivas Básicas 5

2.4 Exemplos 6

2.4.1 Servidor Central 6

2.4.2 Sistema Computacional Simplificado 8

3. Extensão Funcional *SMPLx*

3.1 Introdução 10

3.2 Estruturas de Dados 10

3.3 Exemplos 13

3.3.1 Servidor Central 13

3.3.2 Sistema Computacional Simplificado 15

4. Extensão Funcional *ParSMPL*

4.1 Introdução 17

4.2 Estruturas de Dados 18

4.3 Estrutura dos Processos Lógicos – Implementação do Protocolo CMB 21

4.4 Primitivas Básicas 23

4.5 Tabela de Identificação de Processos 25

4.6 Modelo de Programação 26

4.7 Tipos de Mensagens 28

4.8 Implementação dos Processos e Comunicação 29

4.9 Exemplo 32

4.9.1 Servidor Central 32

4.9.2 Sistema Computacional Simplificado 34

5. Considerações Finais 38

6. Referencias Bibliográficas 39

1. Introdução

Este relatório descreve as características e estruturas de dados das extensões funcionais *SMPLx e ParSMPL* [Ulson 1999]. *SMPLx* constitui uma extensão funcional da linguagem C para simulação discreta orientada a eventos. O desenvolvimento do *SMPLx* foi baseado na extensão funcional *SMPL* [MacDougall 1987]. P*arSMPL* é uma extensão para simulação distribuída conservativa, que utiliza o protocolo *CMB* [Misra 1986, Fujimoto 2000] para sincronização entre os processos paralelos, e foi desenvolvida com base no *SMPLx*. Assim, *ParSMPL* é uma extensão funcional da linguagem de programação C para simulação distribuída [Ulson et al. 1997, Ulson 1999].

SMPLx constitui uma versão do *SMPL* para plataformas *UNIX*. *ParSMPL* utiliza os ambientes de passagem de mensagens *PVM*[1] [Geist et al. 1994a, Geist et al. 1994b, Geist et al. 1996] e *PVMe*[2] [IBM Corporation 1996] para implementar a comunicação e sincronização entre os processos. Ambas extensões foram desenvolvidas em linguagem C, em plataformas *IBM/RS6000* e *IBM SP2* (*ParSMPLx*) utilizando o sistema operacional *AIX*.

Este relatório está dividido nas seguintes escoes: a seção 2 apresenta as características e estruturas básicas do *SMPL* e exemplos de programas de simulação; a seção 3 apresenta as alterações realizadas no *SMPL* para o desenvolvimento da extensão *SMPLx*, suas estruturas de dados e programas exemplo. Na seção 4 são apresentadas as características, estruturas de dados e programas exemplo utilizando o *ParSMPL*. A seção 5 encerra o relatório, apresentando as considerações finais.

2. Extensão Funcional *SMPL*

2.1 Introdução

Na extensão funcional *SMPL* existem três entidades básicas para a representação dos modelos: facilidades, *tokens* e eventos [MacDougall 1987]. Cada facilidade representa um recurso (centro de serviço) do sistema que está sendo modelado, como

[1] *PVM* domínio público versão 3.3.10 – *Oak Ridge National Laboratory*
[2] *IBM PVMe for AIX – IBM Corporation*

por exemplo uma *CPU* (*Central Processing Unity*) em um modelo de um sistema computacional ou um barramento em um modelo de rede de comunicação. Um sistema corresponde a uma coleção de facilidades conectadas. Com as primitivas do *SMPL* é possível definir, reservar e liberar as facilidades, bem como verificar seu estado. A interconexão das facilidades é determinada pelo roteamento dos *tokens* entre as mesmas.

Os *tokens* representam as entidades ativas do sistema, como por exemplo uma tarefa em um modelo de um sistema computacional ou um pacote de dados em um modelo de rede de comunicação. *SMPL* fornece dois tipos básicos de operações para o controle do fluxo dos *tokens* na simulação, que envolvem a reserva de uma facilidade e o escalonamento de eventos.

Se um *token* reserva uma facilidade que não tem servidores disponíveis, esse *token* é enfileirado até que algum servidor fique disponível e possa atendê-lo. O único atributo que *SMPL* disponibiliza aos *tokens* é a prioridade, o que permite a preempção das facilidades.

A mudança de estado de qualquer entidade do sistema representa um evento. Por exemplo, quando uma tarefa ou processo encerra sua execução e libera a facilidade *CPU*, dois eventos ocorrerão: o final da execução da tarefa e a liberação da *CPU*. Um evento é individualizado pelo seu número, o tempo de ocorrência e a identificação do *token* relacionado com esse evento. *SMPL* fornece rotinas para submeter e para selecionar eventos que vão ocorrer, além de ferramentas para depuração, coleta de dados, análises estatísticas e uma interface interativa para a execução de programas de simulação, denominada *mtr* [MacDougall 1987].

Os programas desenvolvidos em *SMPL* são caracterizados pela existência de um laço principal que comanda as iterações da simulação. Dentro desse laço os eventos são processados e os *tokens* são escalonados para novos eventos, e são realizadas operações para a coleta de informações estatísticas. A figura 2.1 apresenta a estrutura básica de um programa de simulação utilizando o *SMPL*.

```
#include <smpl.h>

main()
{
prepara variáveis de estado
inicia variáveis

facility()

schedule()

cause(evento, token)
{
        evento 1:

        evento 2:

        evento n:
}
report()
}
```

Figura 2.1 – Estrutura Básica de um Programa de Simulação em *SMPL*.

2.2 Estruturas de Dados

Em *SMPL* um único elemento básico de dados é utilizado para representar todas as estruturas de dados necessárias. *SMPL* mantém um *pool* de elementos que podem ser requisitados como descritores para os recursos, indicações para as filas desses recursos e indicações para os elementos da lista de eventos futuros. Cada elemento possui cinco campos conforme ilustra a figura 2.2. As estruturas de dados do *SMPL* são requisitadas a partir desse *pool* de elementos. Com esses elementos, *SMPL* cria estaticamente as listas encadeadas e as estruturas necessárias para a simulação (lista de eventos futuros, fila de eventos e descritores de recursos).

L5	L4	L3	L2	L1
float	*float*	*int*	*int*	*int*

Figura 2.2 - Elemento Básico de Dados do *SMPL*.

Os valores armazenados nos campos dos elementos dependem da estrutura que esse elemento representa. Se o elemento for requisitado como uma posição da lista de

eventos futuros, devem ser armazenadas informações sobre o tempo de ocorrência do evento, o número do *token* e o número do evento. No caso de um descritor de recurso são armazenadas informações sobre o número de servidores do recurso, o número de servidores ocupados, o nome do recurso e contadores de fila e preempção. Para um elemento da fila de um recurso, devem ser armazenados identificação e prioridade do *token*, contadores para o número de vezes em que o recurso foi liberado e período de ocupação do recurso.

2.3 Primitivas Básicas

SMPL fornece ao usuário um conjunto de primitivas que auxiliam o desenvolvimento de programas de simulação com orientação a eventos. As principais primitivas dessa extensão são descritas a seguir.

smpl(): responsável por iniciar o sistema para que a simulação possa ser executada. Atribui os valores iniciais às estruturas de dados e ao tempo de simulação (relógio);

facility(): essa primitiva cria, nomeia e retorna um descritor para cada facilidade do modelo. O descritor é utilizado para qualquer operação a ser realizada na facilidade, como a requisição ou o encerramento dos serviços. Cada facilidade possui um número *n* de servidores e uma fila de acesso. Quando a reserva da facilidade é executada, os servidores são examinados e se algum estiver disponível, esse será reservado para o *token* solicitante;

request(): essa primitiva é utilizada para requisitar o atendimento de um dos servidores de uma facilidade. Se um servidor está livre, esse servidor é reservado para o *token* solicitante. Se não existir servidor disponível, a primitiva retorna um valor indicando que a facilidade está ocupada. Neste caso, a requisição é inserida na fila da facilidade;

release(): libera um servidor de uma determinada facilidade. Após liberar o servidor, o sistema atualiza as estatísticas e decrementa o número de servidores ocupados da facilidade. Caso exista algum evento pendente na fila da facilidade, esse evento é inserido no início da lista de eventos futuros para execução imediata;

schedule(): responsável pelo escalonamento dos eventos da simulação. Todo evento a ser escalonado é inserido na lista de eventos futuros em ordem crescente do tempo de ocorrência do evento;

cause(): essa primitiva remove o elemento do início da lista de eventos futuros e avança o tempo da simulação (relógio) para o tempo de ocorrência do evento a ser executado.

O *SMPL* possui também diversas primitivas para a manipulação das filas de eventos dos recursos e da lista de eventos futuros, além de primitivas para análises estatísticas.

2.4 Exemplos

Os exemplos a seguir apresentam o código fonte para os modelos Servidor Central [MacDougall 1987] e Sistema Computacional Simplificado [Ulson 1999]. As figuras 2.3 e 2.4 apresentam o modelo de redes de filas para esses modelos.

2.4.1 Servidor Central

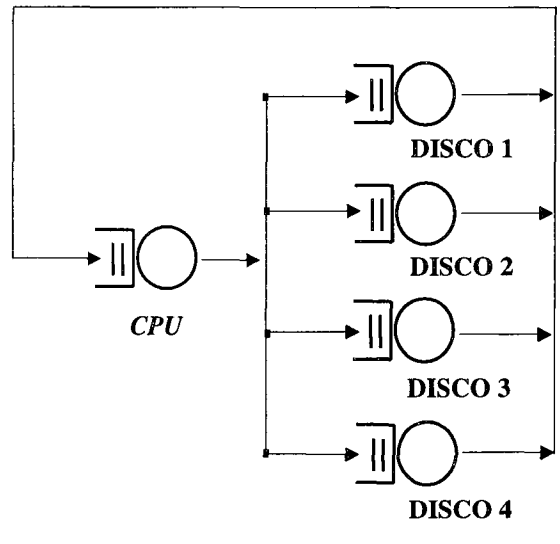

Figura 2.3: Modelo Servidor Central

```
#include "d:\Meus documentos\arquivos\smpl.c"
#include "d:\Meus documentos\arquivos\rand.c"
#include "d:\Meus documentos\arquivos\bmeans.c"

#define nd 4            /* no. of disk units  */
#define qd 1            /* queued req. return */
```

```
main()
{
int
  disk1,        /* disk facility descriptors */
  disk2,
  disk3,
  disk4,
  i, event, aux,
  cpu,          /* cpu facility descriptor   */
  nts=30000;    /* no. of tours to simulate  */

real
  tc=10.0,           /* mean cpu times  */
  td=30.0, sd=7.5;   /* disk time mean, std. dev. */


    smpl(0,"central server model");

    cpu=facility("CPU",1);

    disk1=facility("disk1",1);
    disk2=facility("disk2",1);
    disk3=facility("disk3",1);
    disk4=facility("disk4",1);

    for (i=1; i<=5; i++) schedule(1,0.0,i);

    while (nts)
      {
        cause(&event,&i);
        switch(event)
          {
            case 1:  /* begin tour */
              schedule(2,0.0,i);
            break;

            case 2:  /* request cpu */
                  if (request(cpu,i,0)!=qd) then
                schedule(3,expntl(tc),i);
            break;

            case 3:  /* release cpu, select disk */
              release(cpu,i);
              aux = random(2,5);
              schedule(2*aux,0.0,i);
            break;

            case 4:  /* request disk1 */
              if (request(disk1,i,0)!=qd) then
                schedule(5,erlang(td,sd),i);
            break;

            case 5:  /* release disk1, end tour */
                  release(disk1,i);
                  schedule(1,0.0,i); nts--;
              break;

            case 6:  /* request disk2 */
              if (request(disk2,i,0)!=qd) then
                schedule(7,erlang(td,sd),i);
            break;

            case 7:  /* release disk2, end tour */
                  release(disk2,i);
                  schedule(1,0.0,i); nts--;
              break;
```

```
            case 8:  /* request disk3 */
              if (request(disk3,i,0)!=qd) then
                schedule(9,erlang(td,sd),i);
            break;

            case 9:  /* release disk3, end tour */
                  release(disk3,i);
                  schedule(1,0.0,i); nts--;
            break;

            case 10:  /* request disk3 */
              if (request(disk4,i,0)!=qd) then
                schedule(11,erlang(td,sd),i);
            break;

            case 11:  /* release disk4, end tour */
                  release(disk4,i);
                  schedule(1,0.0,i); nts--;
            break;
        }
      }
    report();
}
```

2.4.2 Sistema Computacional Simplificado

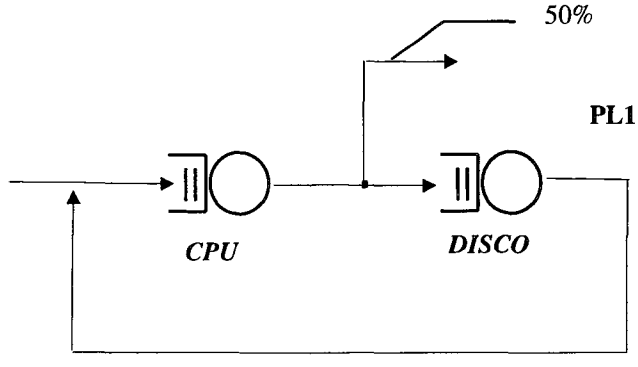

Figura 2.4: Modelo Sistema Computacional Simplificado

```
#include "d:\usr\rob\smpl\smpl.h"
#include <time.h>
#include <dos.h>

main()
{

   int
    indice,
    token = 1,      /* identificação do cliente que utiliza o recurso */
    evento_execut, /* identificação do evento a ser executado        */
    num_tours = 10000,
    rec1,
    rec2,
    i,
    option,
    gran; /* granulosidade do processo */


 double
   tarec1  = 50.0,
```

```
  tsrec1  = 10.0,
  tsrec2  = 20.0;

time_t comeco, fim;


comeco = time(NULL);

smpl(0,"Processo CPU"); /* inicia o sistema */


/* cria descritores para os recursos*/
rec1=facility("REC1",1);
rec2=facility("REC2",1);

/* escalona eventos iniciais */
schedule(1,0.0,token);

while (num_tours)
 {
  cause(&evento_execut, &token);

  switch(evento_execut)
  {
   case 1:
    schedule(1,expntl(tarec1),token);
    schedule(2,0.0,token);
    for (gran = 0; gran < 15000; gran++);
  break;

   case 2:
    if (request(rec1,token,0) == 0)
      schedule(3,expntl(tsrec1),token);
    for (gran = 0; gran < 15000; gran++);
   break;

   case 3:
    release(rec1,token);
    option = random(1,10);
    if (option < 6)
    {
      num_tours--;
   }
    else
      schedule(4,0.0,token);
    for (gran = 0; gran < 15000; gran++);
   break;

   case 4:
    if (request(rec2,token,0) == 0)
      schedule(5,expntl(tsrec2),token);
    for (gran =0; gran < 15000; gran++);
   break;

   case 5:
      release(rec2,token);
      schedule(2,0.0,token);
      for (gran =0; gran < 15000; gran++);
   break;

  }
 }
fim = time(NULL);
report();
printf("\n Tempo Execucao : %f \n",difftime(fim,comeco));
}
```

Instituto de Ciências Matemáticas e de Computação - Biblioteca

3. Extensão Funcional *SMPLX*

3.1 Introdução

A extensão *SMPLx* corresponde a uma nova versão da extensão *SMPL*. No desenvolvimento do *SMPLx* foram mantidas as mesmas primitivas do *SMPL*, porém, a forma de representação de suas estruturas de dados foram alteradas. Enquanto *SMPL* utiliza estruturas de dados estáticas; no *SMPLx* foram utilizadas estruturas dinâmicas. A sintaxe das primitivas no *SMPLx* é a mesma do *SMPL*.

3.2 Estruturas de Dados Básicas

A principal alteração efetuada para a criação do *SMPLX* ocorreu nas estruturas de dados da extensão. *SMPL* emprega estruturas de dados estáticas, requisitadas a partir de um *pool* de elementos básicos. A utilização de estruturas estáticas limita a complexidade dos modelos a serem implementados, uma vez que as dimensões das estruturas devem ser definidas em tempo de compilação.

No *SMPLX* o *pool* de elementos estático foi substituído por estruturas de dados dinâmicas, o que permite maior flexibilidade na implementação dos modelos. Cada recurso que compõe o modelo é representado por uma estrutura do tipo **Recurso,** que contém todas as informações sobre o mesmo. Para cada recurso existe uma fila com informações sobre os eventos não executados (estrutura **Fila_Eventos**) e os diversos servidores que compõem o recurso (estrutura **Lista_Servidores**). A figura 3.1 apresenta as estruturas de dados definidas para a nova versão da extensão funcional e as tabelas 3.1, 3.2 e 3.3 apresentam a descrição dessas estruturas.

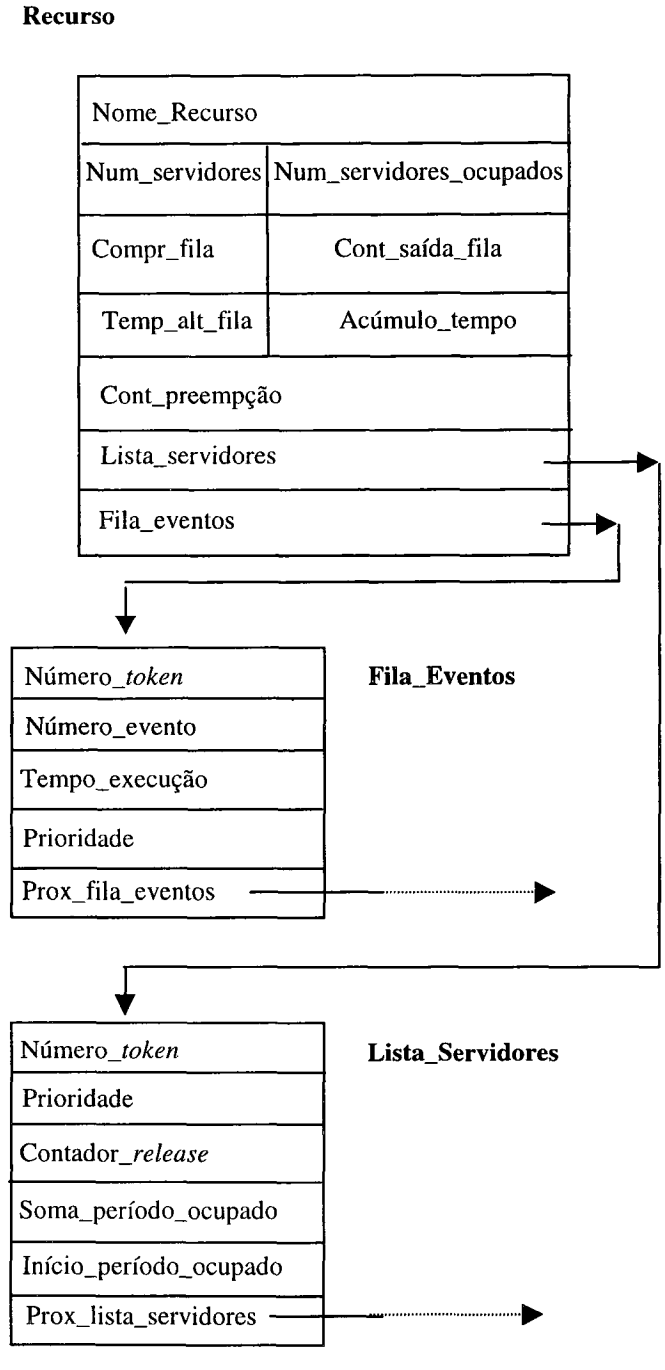

Figura 3.1 - Estruturas de Dados do *SMPLX*.

Tabela 3.1 – Descrição das Estruturas de Dados do *SMPLX* – Recurso.

Campo	Descrição	Tipo
Nome_recurso	Identificação do recurso	*char*
Cont_preempção	Contador de preempção do recurso	*int*
Num_servidores	Número de servidores disponíveis para o recurso	*int*
Num_servidores_ocupados	Número de servidores ocupados	*int*
Compr_fila	Número de *tokens* na fila do recurso	*int*
Cont_saída_fila	Contador do número de vezes que um *token* deixa a fila do recurso	*int*
Temp_alt_fila	Tempo de ocorrência da última alteração da fila	*float*
Acúmulo_tempo	Somatório dos períodos de tempo ocupado, para cálculo do comprimento médio da fila	*float*
Lista_servidores	Informações sobre os servidores que compõem o recurso	*ponteiro*
Fila_eventos	Fila de eventos que aguardam execução no recurso	*ponteiro*

Tabela 3.2 – Descrição das Estruturas de Dados do *SMPLX* - Lista_Servidores.

Campo	Descrição	Tipo
Número_*token*	Identificação do *token*	*char*
Prioridade	Prioridade do *token* (para efetuar preempção)	*int*
Contador_*release*	Número de vezes em que o servidor é liberado	*int*
Soma_período_ocupado	Somatório do período em que o servidor ficou ocupado	*float*
Início_período_ocupado	Marca de tempo em que o servidor iniciou o atendimento ao *token*	*foat*
Prox_lista_servidores	Ponteiro para próximo elemento da lista	*ponteiro*

Tabela 3.3 – Descrição das Estruturas de Dados do *SMPLX* – Fila_Eventos.

Campo	Descrição	Tipo
Número_*token*	Identificação do *token*	*char*
Número_evento	Identificação do evento	*int*
Tempo_execução	Tempo de ocorrência do evento	*float*
Prioridade	Prioridade do *token*	*int*
Prox_fila_eventos	Ponteiro para próximo elemento da fila	*ponteiro*

Além das estruturas apresentadas na figura 3.1, *SMPLX* utiliza uma estrutura de lista encadeada para a representação da lista de eventos futuros (figura 3.2). A tabela 3.4 descreve as informações armazenadas nessa lista.

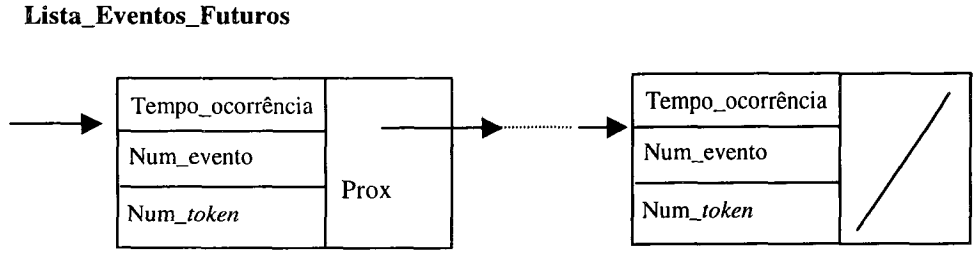

Figura 3.2 – Lista de Eventos Futuros no *SMPLX*.

Tabela 3.4 – Descrição - Lista de Eventos Futuros.

Campo	Descrição	Tipo
Tempo_ocorrência	Tempo de ocorrência do evento	*float*
Num_evento	Identificação do evento	*int*
Num_*token*	Identificação do *token*	*int*
Prox	Próximo elemento da lista	*ponteiro*

3.3 Exemplos

Os programas exemplos apresentados a seguir correspondem aos modelos Servidor Central e Sistema Computacional Simplificados, apresentados na seção 2.

3.3.1 Servidor Central

```
#include "/home/ifsc/ffi/dout/rsulson/parsmplseq.c"
#include "sys/time.h"

#define TASKS 500

main()
{
  struct timeval t1, t2;

  float dt2;

  int
    dt1,
    indice,
    token,          /* identificação do cliente que utiliza o recurso */
    evento_execut, /* identificação do evento a ser executado        */
    num_tours = 50000,
    num = num_tours;

  double
    tempo_medio_cpu = 10.0,
    td = 30.0,
    sd = 7.5;

  gettimeofday(&t1,(struct timezone*)0);

  smpl(0,"Servidor Central"); /* inicia o sistema */
  for (indice=1; indice<= TASKS; indice++)
```

```
 task[indice].ts = 0.0;

/* cria um descritor para o recurso */
facility("cpu",1);
facility("disco1",1);
facility("disco2",1);
facility("disco3",1);
facility("disco4",1);

/* escalona eventos iniciais */
for (indice=1; indice<= TASKS; indice++)
  schedule(1,0.0,indice);

while (num_tours) /* limita a execução da simulação */
{
 cause(&evento_execut, &token);

 switch(evento_execut)
 {
  case 1:
   schedule(2,0.0,token);
  break;

  case 2:
   if (request("cpu",2,token,0) == 0 )
     schedule(3,expntl(10.0),token);
  break;

  case 3:
   release("cpu",token);
   num_tours--;
   indice = randompar(2,5);
   indice = 2*indice;
   schedule(indice,0.0,token);
  break;

  case 4:
   if (request("disco1",4,token,0) == 0)
    schedule(5,erlang(30.0,7.5),token);
   break;

  case 5:
   release("disco1",token);
   schedule(1,0.0,token);
  break;

  case 6:
   if (request("disco2",6,token,0) == 0)
   schedule(7,erlang(30.0,7.5),token);
  break;

  case 7:
   release("disco2",token);
   schedule(1,0.0,token);
  break;

  case 8:
   if (request("disco3",8,token,0) == 0)
    schedule(9,erlang(30.0,7.5),token);
   break;

  case 9:
   release("disco3",token);
   schedule(1,0.0,token);
  break;

  case 10:
   if (request("disco4",10,token,0) == 0)
```

```
      schedule(11,erlang(30.0,7.5),token);
    break;

    case 11:
     release("disco4",token);
     schedule(1,0.0,token);
    break;

   }
  }

  gettimeofday(&t2,(struct timezone*)0);
  dt1 = (t2.tv_sec - t1.tv_sec)*1000000 + t2.tv_usec - t1.tv_usec;
  printf("\n num_tours %d\n",num);
  printf("\n Tempo de execucao (uSegundos): %8d\n",dt1);
  dt2 = (float)(t2.tv_sec - t1.tv_sec) + (float) (((float)(t2.tv_usec -
t1.tv_usec))/1000000);
  printf("\n Tempo de execucao ( --Segundos): %f\n",dt2);


  printf("\nCPU -- Utilizacao Recurso     %f \n",utilizacao_recurso("cpu"));
  printf("\nCPU -- Comprimento Fila       %d \n",comprimento_fila("cpu"));
  printf("\nCPU -- Comprimento Medio Fila %f
\n",comprimento_medio_fila("cpu"));
  printf("\nCPU -- Periodo Medio Ocupado  %f
\n",periodo_medio_ocupado("cpu"));
exit(0);
}
```

3.3.2 Sistema Computacional Simplificado

```
#include "/home/ifsc/ffi/dout/rsulson/smplx.c"
#include <sys/time.h>

main()
{
  float dt2;

  struct timeval t1, t2;

  int
    dt1,
    indice,
    token = 1,
    evento_execut,
    num_tours = 50000,
    num = num_tours,
    option,
    semente=1;

  double
   tarec1  = 50.0,
   tsrec1  = 10.0,
   tsrec2  = 20.0;

  gettimeofday(&t1,(struct timezone*)0);

  smpl(0,"Sistema Computacional Simpificado");

  /* cria descritores para os recursos */
  facility("REC1",1);
  facility("REC2",1);

  /* escalona eventos iniciais */
  schedule(1,0.0,token);
```

```
  while (num_tours)
  {
   cause(&evento_execut, &token);

   switch(evento_execut)
   {
    case 1:
     semente = randompar(1,15);
     stream(semente);
     schedule(1,expntl(tarec1),token);
     schedule(2,0.0,token);
    break;

    case 2:
     semente = randompar(1,15);
     stream(semente);
     if (request("REC1",2,token,0) == 0)
        schedule(3,expntl(tsrec1),token);
    break;

    case 3:
     release("REC1",token);
     option = randompar(1,10);
     if (option <= 5)
       num_tours--;
     else
       schedule(4,0.0,token);
    break;

    case 4:
     semente = randompar(1,15);
     stream(semente);
     if (request("REC2",4,token,0) == 0)
       schedule(5,expntl(tsrec2),token);
    break;

    case 5:
       release("REC2",token);
       schedule(2,0.0,token);
    break;

   }
  }

  printf("\n Utilizacao REC1 %f \n",utilizacao_recurso("REC1"));
  printf("\n Periodo M. ocupado %f\n",periodo_medio_ocupado("REC1"));
  printf("\n Comprimento Medio fila %f\n",comprimento_medio_fila("REC1"));

  printf("\n Utilizacao REC2 %f \n",utilizacao_recurso("REC2"));
  printf("\n Periodo M. ocupado %f\n",periodo_medio_ocupado("REC2"));
  printf("\n Comprimento Medio fila %f\n",comprimento_medio_fila("REC2"));

  gettimeofday(&t2,(struct timezone*)0);

  dt1 = (t2.tv_sec - t1.tv_sec)*1000000 + t2.tv_usec - t1.tv_usec;

  printf("\n Tempo de execucao (uSegundos): %8d\n",dt1);

  dt2 = (float)(t2.tv_sec - t1.tv_sec) + (float) (((float)(t2.tv_usec -
t1.tv_usec))/1000000);
  printf("\n Tempo de execucao ( Segundos): %f\n",dt2);

}
```

4. Extensão Funcional *ParSMPL*

4.1 Introdução

ParSMPL é uma ferramenta de simulação distribuída que utiliza o protocolo conservativo *CMB* para a sincronização entre os processos paralelos. Essa ferramenta é baseada no *SMPL*, uma extensão funcional da linguagem C para simulação discreta orientada a eventos [MacDougall 1987]. Assim, *ParSMPL* é uma extensão funcional da linguagem de programação C para simulação distribuída [Ulson et al. 1997, Ulson 1999, Ulson et al. 1999].

A figura 4.1 apresenta a hierarquia do *software* que compõe o ambiente *ParSMPL*. O usuário desenvolve suas aplicações em linguagem de programação de alto nível, podendo interagir também com as primitivas da extensão para simulação, com o ambiente de passagem de mensagens e com o sistema operacional. Desta forma, usuários mais experientes têm maior flexibilidade no desenvolvimento de suas aplicações.

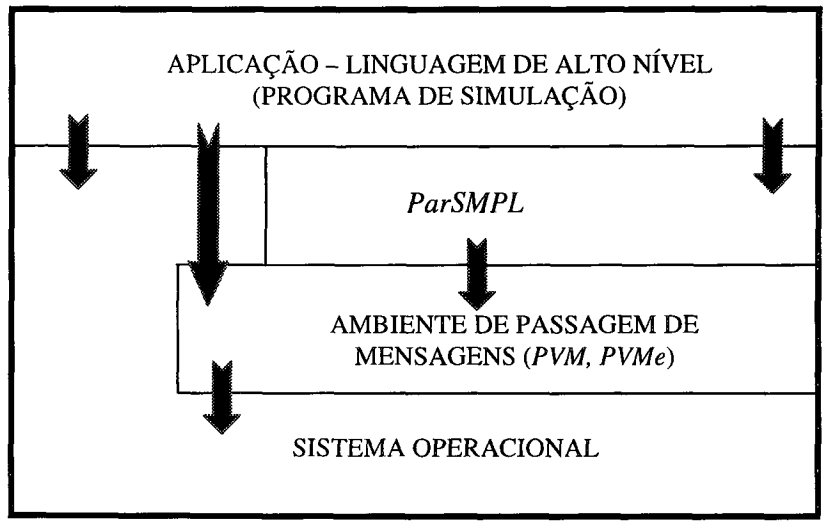

Figura 4.1 - Hierarquia dos Níveis de *Software* no *ParSMPL*.

O desenvolvimento do *ParSMPL* foi baseado nas extensões funcionais *SMPL* e *SMPLx* [Ulson 1999]. Algumas das primitivas dessas extensões foram alteradas e outras foram introduzidas para viabilizar a comunicação e a sincronização entre os processos paralelos. *ParSMPL* mantém a estrutura de dados básicas da extensão *SMPLx*. Foram introduzidas na lista de eventos futuros informações sobre a identificação dos processos para o escalonamento correto dos eventos e para a

comunicação entre esses processos. Nesta seção são descritas a estrutura e as principais características do *ParSMPL*.

4.2 Estruturas de Dados

O desenvolvimento do *ParSMPL* foi realizado tendo-se como base as extensões funcionais *SMPLX* e *SMPL*. A estrutura básica do *SMPLx* foi modificada com a introdução de novas primitivas e mecanismos que permitiram mudar seu comportamento seqüencial, possibilitando assim a comunicação e a sincronização entre os processos paralelos e o desenvolvimento do *ParSMPL*. A estrutura de dados básica do *SMPLX* também foi alterada, incorporando novos dados necessários para o desenvolvimento de programas de simulação distribuída [Ulson et al. 1997, Ulson 1999].

No *ParSMPL*, assim como no *SMPLx*, o *pool* estático de elementos foi substituído por estruturas de dados dinâmicas, o que permite maior flexibilidade na implementação dos modelos. Cada recurso que compõe o modelo é representado por uma estrutura do tipo **Recurso,** que contém todas as informações sobre o mesmo. Para cada recurso existe uma fila com informações sobre os eventos não executados (estrutura **Fila_Eventos**) e os diversos servidores que compõem o recurso (estrutura **Lista_Servidores**). A figura 4.2 apresenta as estruturas de dados definidas para a nova versão da extensão funcional e as tabelas 4.1, 4.2 e 4.3 apresentam a descrição dessas estruturas.

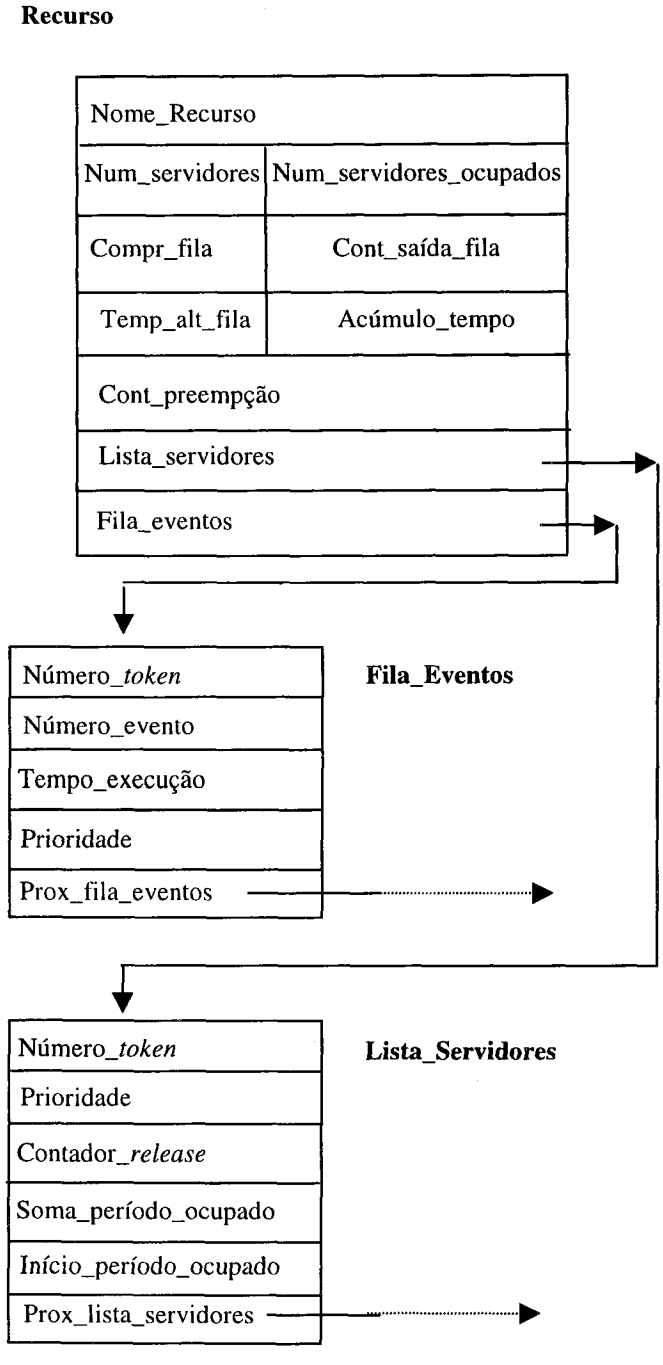

Figura 4.2 - Estruturas de Dados do *ParSMPLX*.

Instituto de Ciências Matemáticas e de Computação - Biblioteca

Tabela 4.1 – Descrição das Estruturas de Dados do *ParSMPL* – Recurso.

Campo	Descrição	Tipo
Nome_recurso	Identificação do recurso	*char*
Cont_preempção	Contador de preempção do recurso	*int*
Num_servidores	Número de servidores disponíveis para o recurso	*int*
Num_servidores_ocupados	Número de servidores ocupados	*int*
Compr_fila	Número de *tokens* na fila do recurso	*int*
Cont_saída_fila	Contador do número de vezes que um *token* deixa a fila do recurso	*int*
Temp_alt_fila	Tempo de ocorrência da última alteração da fila	*float*
Acúmulo_tempo	Somatório dos períodos de tempo ocupado, para cálculo do comprimento médio da fila	*float*
Lista_servidores	Informações sobre os servidores que compõem o recurso	*ponteiro*
Fila_eventos	Fila de eventos que aguardam execução no recurso	*ponteiro*

Tabela 4.2 – Descrição das Estruturas de Dados do *ParSMPL* - Lista_Servidores.

Campo	Descrição	Tipo
Número_*token*	Identificação do *token*	*char*
Prioridade	Prioridade do *token* (para efetuar preempção)	*int*
Contador_*release*	Número de vezes em que o servidor é liberado	*int*
Soma_período_ocupado	Somatório do período em que o servidor ficou ocupado	*float*
Início_período_ocupado	Marca de tempo em que o servidor iniciou o atendimento ao *token*	*foat*
Prox_lista_servidores	Ponteiro para próximo elemento da lista	*ponteiro*

Tabela 4.3 – Descrição das Estruturas de Dados do *ParSMPL* – Fila_Eventos.

Campo	Descrição	Tipo
Número_*token*	Identificação do *token*	*char*
Número_evento	Identificação do evento	*int*
Tempo_execução	Tempo de ocorrência do evento	*float*
Prioridade	Prioridade do *token*	*int*
Prox_fila_eventos	Ponteiro para próximo elemento da fila	*ponteiro*

Além das estruturas apresentadas na figura 4.2, *ParSMPL* utiliza uma estrutura de lista encadeada para a representação da lista de eventos futuros (figura 4.3). A tabela 4.4 descreve as informações armazenadas nessa lista.

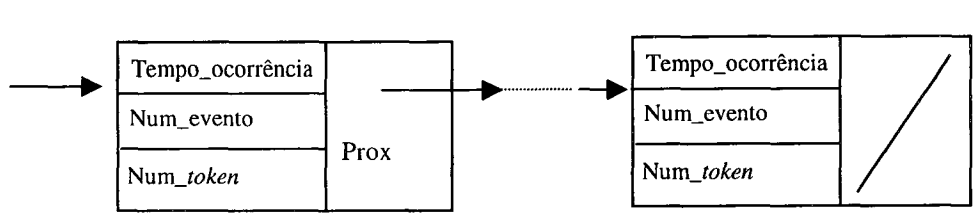

Figura 4.3 – Lista de Eventos Futuros no *SMPLX*.

Tabela 4.4 – Descrição - Lista de Eventos Futuros.

Campo	Descrição	Tipo
Tempo_ocorrência	Tempo de ocorrência do evento	*float*
Num_evento	Identificação do evento	*int*
Num_*token*	Identificação do *token*	*int*
Prox	Próximo elemento da lista	*ponteiro*

4.3 Estrutura dos Processos Lógicos – Implementação do Protocolo *CMB*

Os processos da simulação distribuída utilizando *ParSMPL* são implementados utilizando-se processos padrão *UNIX* (criados com as primitivas do *PVM*). Todos os processos são criados no início da execução da simulação e destruídos no final da mesma, ou seja, não existe a criação dinâmica de processos (característica do protocolo *CMB*). A extensão distribuída é constituída por dois módulos básicos: o núcleo, que contém as primitivas para comunicação, sincronização e prevenção do *deadlock*, e a biblioteca, que contém todas as primitivas necessárias para o desenvolvimento de um programa de simulação, além de primitivas para a coleta e análise estatística dos dados obtidos.

A implementação de uma simulação distribuída é composta por um conjunto de processos lógicos, cada um representando um processo no sistema real. Todas as interações entre os processos físicos são representadas por mensagens trocadas pelos processos lógicos correspondentes. Cada processo possui seu próprio relógio local (o mínimo entre os relógios de cada canal, como explicado no capítulo 3) que controla o seu tempo de simulação e uma lista de eventos na qual ficam armazenadas (em ordem cronológica) as mensagens recebidas de outros processos.

Cada processo tem um número determinado de canais lógicos bidirecionais através dos quais as mensagens são recebidas e enviadas. Nas plataformas utilizadas

não existe um canal físico para a comunicação entre quaisquer dois elementos de processamento, como ocorre em arquiteturas baseadas em *transputers* [Almasi; Gottlieb 1994]. Todos os elementos de processamento disputam o meio de comunicação para a transmissão de suas mensagens. Para organizar a comunicação entre os processos no *ParSMPL* foram criadas abstrações (canais lógicos ou virtuais) e para possibilitar a utilização desses canais é utilizada uma tabela de identificação, que permite aos processos reconhecerem os demais processos da simulação.

Toda a comunicação entre os processos é efetuada utilizando-se as primitivas de comunicação não bloqueantes disponíveis no *PVM*, e todos os aspectos relativos à comunicação e sincronização entre os processos paralelos são transparentes ao usuário.

Os programas desenvolvidos no *ParSMPL* têm sua estrutura semelhante à estrutura de um programa de simulação desenvolvido no *SMPL*. *ParSMPL* foi projetado para permitir ao usuário liberdade na criação de seus próprios modelos, porém o usuário deve ser responsável por garantir que novos processos não sejam criados durante a execução da simulação, já que uma das características do protocolo *CMB* é a configuração estática dos canais de comunicação entre os processos. Cada um dos processos que compõem a simulação possui a estrutura interna apresentada pela figura 4.4 [Ulson et al. 1997] e descrita na tabela 4.5. A criação de todas as estruturas de dados é executada automaticamente pelo *ParSMPL*, sendo esse processo é totalmente transparente para o usuário.

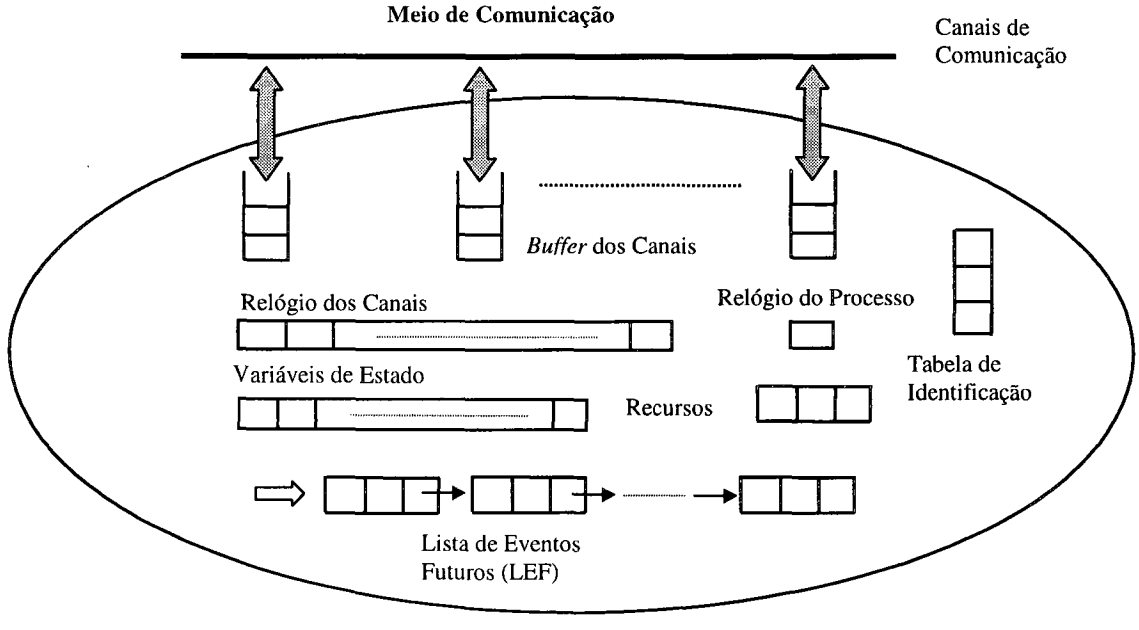

Figura 4.4 - Estrutura de Processos no *ParSMPL*

Tabela 4.5 - Estruturas de Dados dos Processos Lógicos no *ParSMPL*.

Canais de Comunicação	Permitem aos processos enviar e receber mensagens completas (que são inseridas na Lista de Eventos Futuros do processo) e mensagens nulas.
***Buffer* de Canais**	Armazena as marcas de tempo de todas as mensagens recebidas pelo processo e que ainda não foram executadas.
Relógio dos Canais	Armazena o valor da marca de tempo da primeira posição do *buffer*, que indica o relógio do respectivo canal. Se o *buffer* estiver vazio, assume o valor da última mensagem recebida pelo canal. Se não recebeu mensagem, armazena o valor zero.
Relógio do Processo	Indica o menor valor entre os relógios dos canais e corresponde ao tempo da simulação em execução.
Lista de Eventos Futuros (LEF)	Contém os eventos a serem executados pelo processo, ordenados de maneira ascendente pelo tempo de execução.
Variáveis de Estado	Armazena as informações necessárias para a execução da simulação, como identificadores de processo e informações relativas à comunicação entre os processos e coleta de estatísticas.
Recursos	Armazena todas as informações sobre os recursos pertencentes ao processo.
Tabela de Identificação	Contém os identificadores que permitem a comunicação entre os processos, utilizando as primitivas do *PVM*.

4.4 Rotinas Básicas

Algumas rotinas do *SMPLX* foram alteradas e novas rotinas foram introduzidas para permitir a execução da simulação distribuída. As alterações realizadas e as primitivas desenvolvidas são descritas a seguir.

Rotinas Alteradas

smpl(): além das variáveis existentes na versão seqüencial e que foram mantidas (como o tempo de simulação), essa rotina inicia as novas variáveis e estruturas de dados, como os descritores de recursos, o descritor da lista de eventos futuros e os contadores de mensagens nulas e completas (o número de mensagens é uma das métricas utilizadas para avaliação do protocolo *CMB* e discutida no capítulo 5);

schedule(): antes de inserir um evento na lista de eventos futuros do processo, essa rotina verifica se o mesmo deve ser escalonado localmente ou em um outro processo (escalonamento remoto). Se o evento deve ser executado localmente, o mesmo é inserido na lista de eventos futuros do processo. Se o evento deve ser executado em um outro processo, as

informações necessárias para a execução desse evento são enviadas ao processo destino;

cause(): essa rotina é acionada sempre que um evento deve ser executado e verifica se o tempo de ocorrência do próximo evento é menor ou igual ao relógio do processo. Em caso afirmativo, remove o elemento do início da lista de eventos futuros do processo e avança o tempo da simulação local para o tempo de ocorrência do evento a ser executado. Caso contrário (se o processo está bloqueado) é acionado o mecanismo de envio de mensagens nulas para todos os processos com os quais o processo se comunica. Essa rotina também verifica se o processo tem alguma mensagem a receber. Se houver mensagem, essa é recebida, identificada e tratada corretamente, de acordo do seu tipo (mensagem nula ou completa). Se for uma mensagem completa (resultado de um escalonamento remoto), o evento correspondente é inserido na lista de eventos futuros do processo. Se for uma mensagem nula, o relógio do canal é atualizado.

Nas demais rotinas foram realizadas as adaptações necessárias em função das estruturas de dados dinâmicas que foram inseridas na extensão, além de informações necessárias para a comunicação entre os processos paralelos, como a identificação dos processos origem e destino.

Novas Rotinas

ReceiveMessage(): essa rotina é chamada pela rotina *cause()* para verificar se o processo tem alguma mensagem (completa ou nula) para ser recebida, conforme descrito no capítulo 3;

SendNull(): quando o processo é bloqueado, ou seja, quando o tempo de ocorrência do elemento do início da lista de eventos futuros for maior que o relógio do processo, ou então a lista de eventos estiver vazia, mensagens nulas são enviadas para os demais processos com os quais o processo se comunica;

SendComplete(): envia uma mensagem completa para o processo destino contendo informações sobre o evento a ser escalonado, o tempo de ocorrência desse evento, a identificação do *token* e as identificações dos processos origem e destino;

InsertFEL(): insere um evento na lista de eventos futuros do processo;

InsertChannelBuf(): insere na estrutura de *buffer* de canais o tempo de ocorrência da mensagem recebida.

A hierarquia das principais rotinas no *ParSMPL* é apresentada na figura 4.5. No *ParSMPL* o usuário interage apenas com as primitivas do primeiro nível da hierarquia (destacadas por uma linha tracejada).

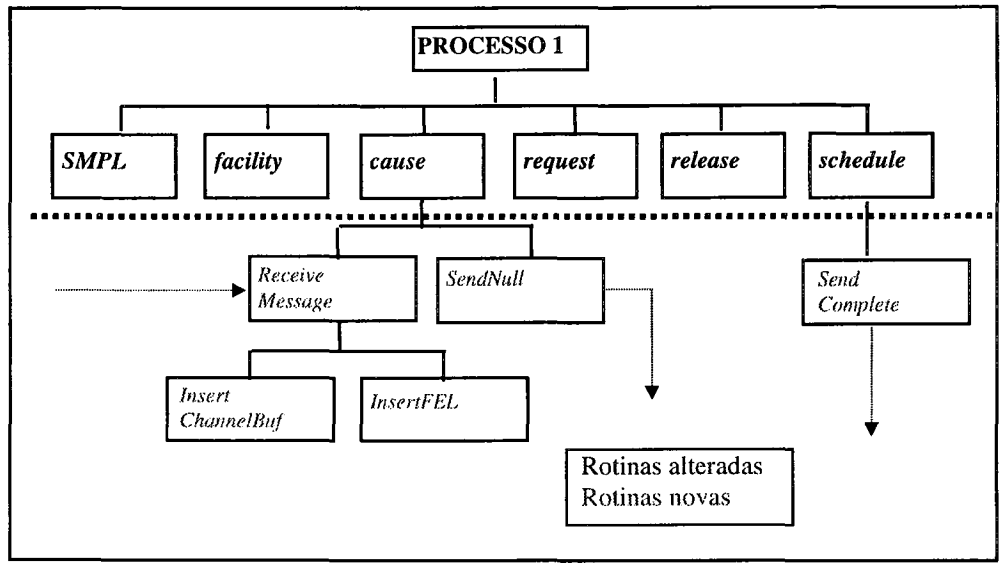

Figura 4.5 – Hierarquia das Rotinas no *ParSMPL*.

4.5 Tabela de Identificação de Processos

Como citado anteriormente, cada processo possui um número determinado de canais lógicos bidirecionais para receber e enviar mensagens. Para possibilitar a utilização desses canais é necessário preparar a tabela de identificação, que contém o identificador de todos os processos que compõem a simulação (esse identificador é um número inteiro que o *PVM* usa para a identificação das tarefas criadas no sistema).

A tabela de identificação contém uma entrada para cada processo lógico da simulação. Cada entrada é constituída por dois campos: um identificador de processo

(inserido no código dos processos, assumindo valor 0, 1, ... n-1, onde n representa o número total de processos) e um identificador que o *PVM* retorna quando os processos são criados. A tabela 4.6 apresenta a estrutura da tabela de identificação.

Tabela 4.6 – Tabela de Identificação de Processos.

Identificação Processo	Identificação *PVM*
0	Tid_processo0
1	Tid_processo1
⋮	⋮
n-1	Tid_processon-1

No *ParSMPL*, após a criação dos processos lógicos, a tabela de identificação é criada e enviada para todos os processos que compõem a simulação. A simulação tem início a partir do momento em que todos os processos receberam uma cópia dessa tabela. A criação da tabela é efetuada pelo processo mestre ou processo pai em função do modelo de programação utilizado. Esses modelos são descritos na seção 4.7.4.

4.6 O Modelo de Programação no *ParSMPL*

No desenvolvimento dos programas de simulação utilizando o *ParSMPL* é empregado o paradigma *MPMD*, ou seja, os processadores armazenam partes distintas do código, dependendo do processo a ser executado. Dentro desse paradigma, pode-se utilizar dois modelos de programação no *ParSMPL*. No modelo 1 um processo especial, denominado processo pai, é responsável por iniciar todos os demais processos e preparar a tabela de identificação (paradigma mestre/escravo). No modelo 2 um dos processos que compõem a simulação é responsável por essa tarefa. Os modelos de programação do *ParSMPL* são apresentados nas figura 4.6 e 4.7.

- **Modelo de Programação 1: Utilização do processo pai**

Nesse modelo o processo Pai não representa nenhum processo no sistema físico simulado. Sua função é controlar a execução da simulação, definindo e enviando a tabela de identificação aos demais processos lógicos que compõem a simulação e

criando os processos filhos. A figura 4.6 apresenta um exemplo com o processo pai e três processos filhos.

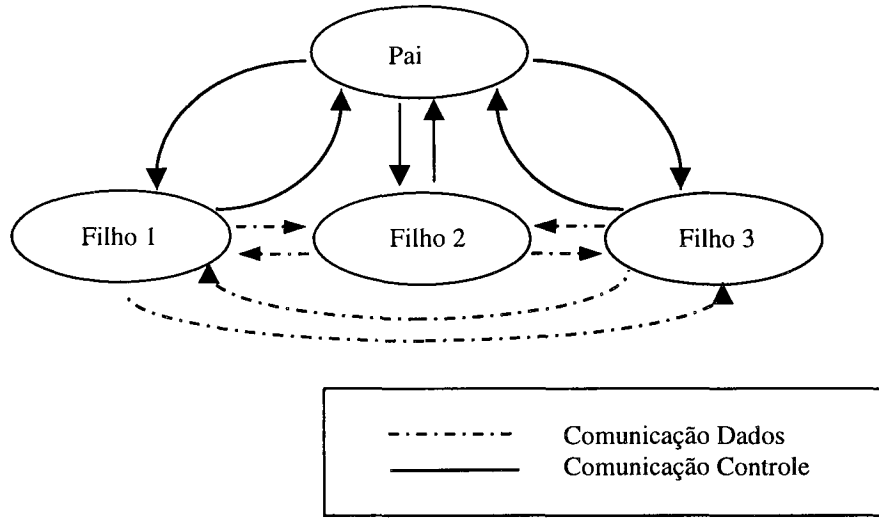

Figura 4.6 - Modelo de Programação 1 no *ParSMPL*.

- **Modelo de Programação 2: Sem a utilização do processo pai**

Como a criação do processo pai em alguns casos pode gerar uma sobrecarga desnecessária (a execução do processo pai concorrendo com a execução de um processo filho), foi criado outro modelo de programação no *ParSMPL*. Nesse modelo não existe a figura do processo pai e um dos processos que compõem a simulação (chamado processo mestre) é responsável também por iniciar os demais processos e preparar a tabela de identificação. A figura 4.7 apresenta um exemplo com três processos lógicos executados em elementos de processamento distintos.

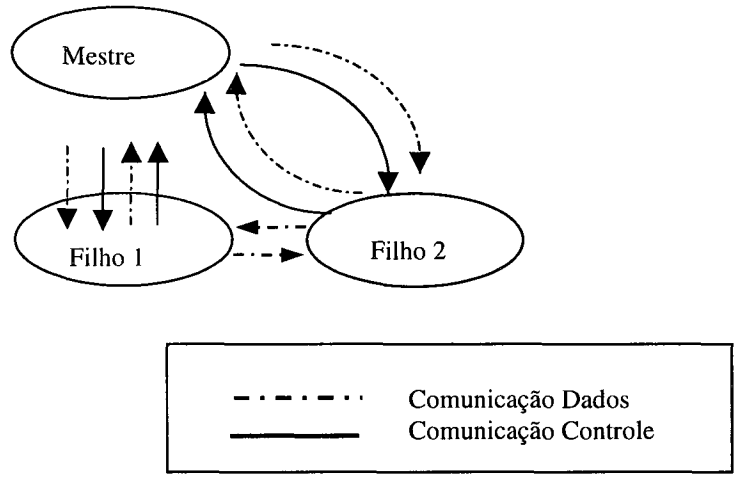

Figura 4.7 - Modelo 2 de Programação no *ParSMPL*.

4.7 Tipos de Mensagens

De acordo com as características do protocolo *CMB*, cada processo no *ParSMPL* pode enviar para outro processo dois tipos distintos de mensagens, denominadas mensagens completas e mensagens nulas.

Mensagens Completas

Os *tokens* que circulam pelo sistema de simulação são escalonados para eventos específicos, que podem ser locais ou remotos. Isso implica então em escalonamento local e escalonamento remoto de mensagens. O escalonamento local ocorre quando o evento deve ser executado no processo em questão. O escalonamento remoto ocorre quando o evento a ser executado deve ser escalonado para processamento em outro processo lógico. Neste caso é necessário enviar ao processo destino todas as informações por meio de uma mensagem completa para que ocorra o escalonamento. A figura 4.8 apresenta a estrutura de uma mensagem completa.

Te (*float*)	Num_evento(*int*)	*Token* (*int*)	Ident_Origem(*int*)	Ident_destino(*int*)

Te: Tempo de ocorrência do evento
Num_evento: Identificação do evento a ser escalonado
Token: Identificação do *token*
Ident_origem: Identificação do processo emissor da mensagem
Ident_destino: Identificação do processo receptor da mensagem

Figura 3.8 - Estrutura de uma Mensagem Completa.

Mensagens Nulas

De acordo com o protocolo *CMB*, quando um processo está bloqueado, deve enviar mensagens nulas para os demais processos com os quais se comunica, para evitar que a simulação entre em *deadlock*.

As mensagens nulas não representam qualquer evento do sistema real, apenas permitem que os processos lógicos atualizem seus relógios internos, com o objetivo de permitir o progresso da simulação e evitar a ocorrência de *deadlock*. O incremento de tempo das mensagens nulas é determinado em função do modelo que está sendo simulado. A figura 4.12 apresenta a estrutura de uma mensagem nula.

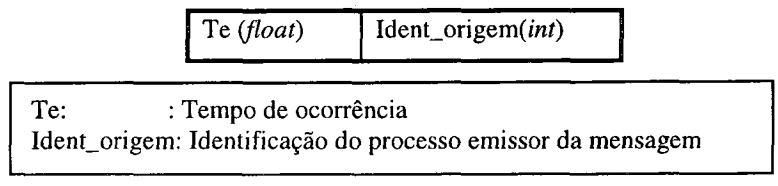

Figura 4.9 - Estrutura de uma Mensagem Nula.

4.8 Implementação dos Processos e Comunicação

Os programas de simulação desenvolvidos em *SMPL* são caracterizados pela existência de um laço principal que comanda as iterações da simulação. Dentro desse laço os eventos são executados e escalonados para ocorrência, além da execução de operações para coleta de informações estatísticas.

No *ParSMPL* os programas de simulação têm estrutura semelhante aos programas desenvolvidos em *SMPL*, tornando transparente para o usuário os mecanismos de comunicação e sincronização entre os processos. As rotinas que implementam os mecanismos de simulação no *ParSMPL* têm nomes similares às rotinas do *SMPL*, com o intuito de facilitar a utilização da extensão pelos usuários. A figura 4.10 apresenta a estrutura dos processos lógicos para o modelo de programação 2. Para o modelo 1, a estrutura dos processos filhos continua a mesma, e o processo pai (principal) contém apenas as linhas identificadas por um (*) no processo mestre.

PROCESSO MESTRE	**PROCESSO FILHO**
Inclui bibliotecas + núcleo *ParSMPL*	Inclui bibliotecas + núcleo *ParSMPL*
Inicia Processos (*)	
Prepara Tabela Identificação (*)	
Envia Tabela de Identificação (*)	
	Recebe Tabela de Identificação
Prepara Variáveis de Estado	Prepara Variáveis de Estado
Inicia variáveis	Inicia variáveis
smpl()	*smpl()*
facility()	*facility()*
schedule()	*schedule()*
cause(evento, token)	*cause(evento, token)*
Evento 1:	Evento 1:
Evento 2:	Evento 2:
Evento n:	Evento n:
Recebe Resultado (*)	Envia Resultado
Encerra Processo (*)	Encerra Processo

Figura 4.10 – Descrição de um Processo no *ParSMPL*.

A figura 4.11 [Ulson et al. 1997] apresenta a estrutura com as principais rotinas utilizadas na comunicação entre os processos no *ParSMPL*. Dentro do laço principal, no procedimento *cause()*, o processo executa *ReceiveMessage()* para saber se existe alguma mensagem pendente no *buffer* de recepção. Caso exista alguma mensagem, essa é identificada (completa ou nula) e a rotina correspondente é executada: *InsertFEL()* para inserir a mensagem completa na lista de eventos futuros e *InsertChannelBuf()* para inserir a marca de tempo da mensagem no *buffer* do respectivo canal.

Se existe evento a ser executado, *cause()* retira esse evento da lista de eventos futuros e o executa. Para escalonar um novo evento para execução, *schedule()* identifica se o evento é local ou remoto. Se for um evento local, *schedule()* insere o evento na lista de eventos futuros do processo. Se for um evento remoto *SendComplete()* é executada, enviando uma mensagem completa para o processo destino.

Caso não existam eventos a serem executados (quando a marca de tempo do evento que está no início da lista de eventos futuros é maior que o relógio do processo

ou a lista de eventos está vazia) a rotina *SendNull()* é executada para enviar mensagens nulas para os processos com os quais o processo em questão se comunica. A condição de término da simulação é determinada no processo responsável por iniciar a simulação (processo pai ou processo mestre).

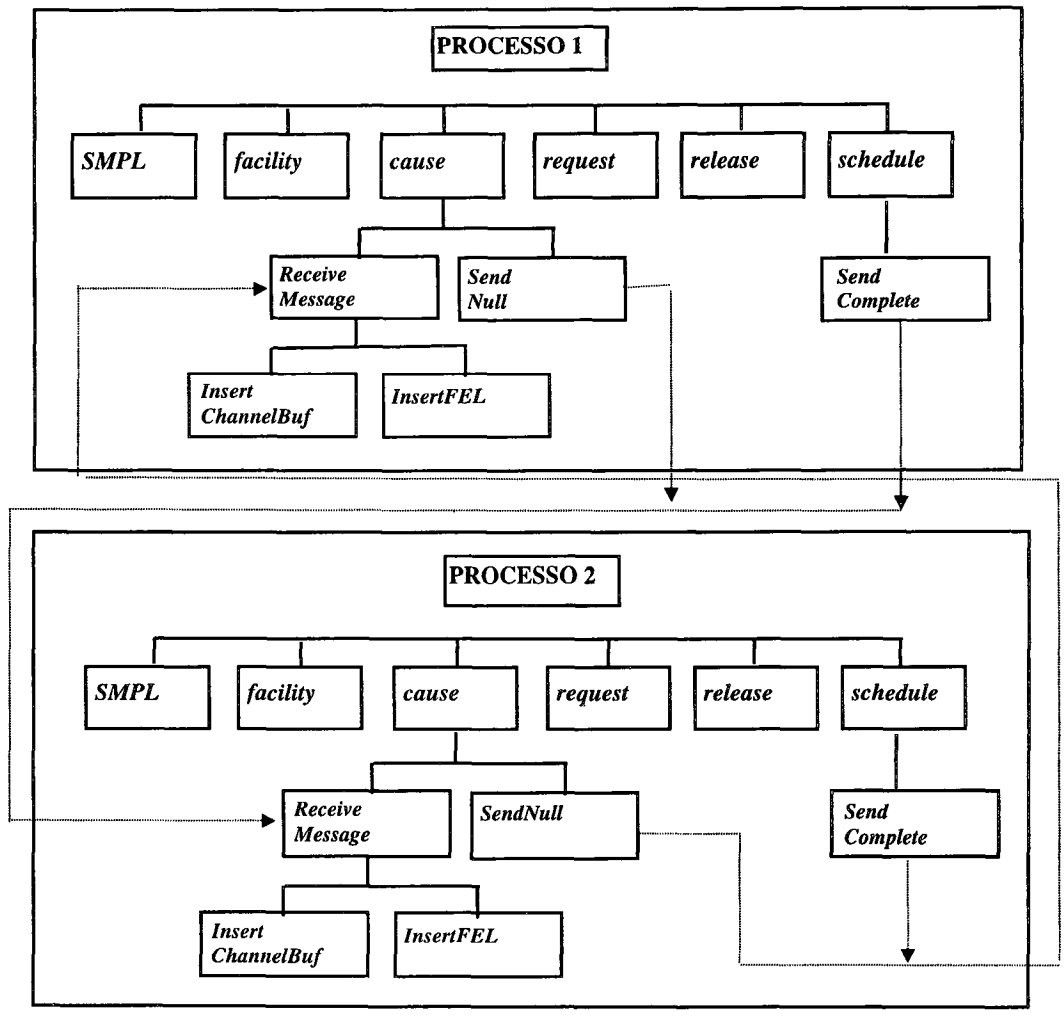

Figura 4.11 - Comunicação entre Processos no *ParSMPL*.

Cabe ressaltar que, com a utilização do *ParSMPL*, o usuário deve estar familiarizado apenas com as características de um programa de simulação orientado a eventos e ter noções sobre programação paralela para poder criar o código dos processos. O usuário não precisa ter conhecimento de programação em *PVM/PVMe*, uma vez que todas as operações de comunicação e sincronização são transparentes. Isto é, o usuário interage apenas com as rotinas da primeira camada da estrutura apresentada na figura 4.11 (algumas dessas rotinas são *SMPL()*, *facility()*, *cause()*, *request()*, *release()* e *schedule()*).

4.9 Exemplos

Os programa exemplo a seguir corresponde ao modelo Servidor Central, apresentado na seção 2.

4.9.1 Servidor Central

Nesse exemplo, o modelo foi particionado em dois processos; esses processos são identificados por CPU e DISCO_1. Considerando o desenvolvimento da simulação distribuída com o *ParSMPL*, as linhas destacadas identificam as primitivas executadas para a preparação da tabela de identificação e o envio dessa tabela para os processos filhos no processo lógico 1 (linhas 19 a 37 - figura 4.12) e recepção da tabela de identificação no processo lógico 2 (linhas 15 a 25 – figura 4.13). O usuário pode alterar o *lookahead* dos processos por meio do comando das linhas 18 (figura 4.12) e 14 (figura 4.13).

Além dessas primitivas, o usuário deve incluir no código dos processos a biblioteca *parsmpl.c* (linha 1 – figuras 4.12 e 4.13) e identificar os processos que vão constituir a simulação (linhas 2 e 3 – figuras 4.12 e 4.13). O comando das linhas 63 (figura 4.12) e 71 (figura 4.13) encerra o processo no *PVM*.

```
1.  #include "/u/rsulson/pvm3/parsmpl.c"

2.  #define CPU       0
3.  #define DISCO_1 1
4.  #define TASKS 50

5.  main()
6.  {
7.  int
8.  tscpu = 10.0,
9.  tid_processo_pai,
10. tid_cpu,
11. tid_disco1,
12. tabela_aux[10],
13. indice,
14. token,
15. evento_execut,
16. num_tours = 50000,
17. num = num_tours;

18. tempo_processo = 1.0;
19. proc_local = 0;

20. for (indice = 1; indice <10; indice++)
21. {
22. tabela_tids[indice] = 0;
23. tabela_aux[indice] =0;
24. }

25. tid_processo_pai = pvm_mytid();
26. pvm_spawn("d",(char**)0,0,"",1,&tid_disco);
27. tabela_tids[CPU] = tid_processo_pai;
28. tabela_tids[DISCO_1] = tid_disco1;
29. tabela_aux[CPU] = tid_processo_pai;
30. tabela_aux[DISCO_1] = tid_disco1;

31. for (indice = 0; indice < 10 ; indice++)
32. if (indice != DISCO_1)
clock_canais[indice] = -1.0;
33. Clock_canais[DISCO_1] = 0.0;
34. pvm_initsend(PvmDataRaw);
35. pvm_pkint(tabela_aux, 10, 1);
36. pvm_send(tid_disco1,1);

37. smpl(0,"Processo CPU");
38. for (indice=1; indice<= TASKS; indice++)
39. task[indice].ts = 0.0;

40. facility("cpu",1);

41. for (indice=1; indice<= TASKS; indice++)
42. schedule(1,0.0,indice,CPU);

43. while (num_tours)
44. {
45. cause(&evento_execut, &token);
46. switch(evento_execut)
47. {
48. case 1:
49. schedule(2,0.0,token,CPU);
50. break;

51. case 2:
52. if (request("cpu",2,token,0,CPU) == 0 )
53. schedule(3,expntl(tscpu),token,CPU);
54. break;

55. case 3:
56. release("cpu",token);
57. num_tours--;
58. indice = (2*randompar(0,3))+1;
59. schedule(indice, clock_sim(), token,DISCO_1);
60. break;
61. );

62. parsmpl_pvm_exit();
63. /* resultados*/
}
```

Figura 4.12: Processo Lógico CPU

```
1.  #include "/u/rsulson/pvm3/parsmpl.c"

2.  #define CPU    0
3.  #define DISCO_1 1

4.  main()
5.  {
6.  int tid_processo_pai,
7.  my_tid,
8.  indice,
9.  contador,
10. token,
11. evento_execut;

12. double td = 30.0,
13. sd = 7.50;

14. tempo_processo = 1.0;
15. proc_local = DISCO_1;
16. my_tid = pvm_mytid();
17. tid_processo_pai = pvm_parent();

18. pvm_recv(tid_processo_pai,-1);
19. pvm_upkint(tabela_tids,10,1);
20. for (indice = 2; indice < 10; indice++)
21. tabela_tids[indice] = 0;
22. for (indice = 0; indice < 10; indice++)
23. if (indice != CPU)
24. clock_canais[indice] = -1.0;

25. clock_canais[CPU] = 0.0;

26. smpl(0,"Processo Disco 1");

27. facility("disco1",1);
28. facility("disco2",1);
29. facility("disco3",1);
30. facility("disco4",1);

31. contador = 10000;

32. while(contador != 0)
33. {
34. cause(&evento_execut, &token);
35. switch(evento_execut)
{
```

```
36. case 1:
37. if (request("disco1",1,token,0,DISCO_1) == 0 )
38. schedule(2,erlang(td,sd),token,DISCO_1);
39. break;

40. case 2:
41. release("disco1",token);
42. schedule(1,clock_sim(),token,CPU);
43. break;

44. case 3:
45. if (request("disco2",1,token,0,DISCO_1) == 0 )
46. schedule(4,erlang(td,sd),token,DISCO_1);
47. break;

48. case 4:
49. release("disco2",token);
50. schedule(1,clock_sim(),token,CPU);
51. break;

52. case 5:
53. if (request("disco3",1,token,0,DISCO_1) == 0 )
54. schedule(6,erlang(td,sd),token,DISCO_1);
55. break;

56. case 6:
57. release("disco3",token);
58. schedule(1,clock_sim(),token,CPU);
59. break;

60. case 7:
61. if (request("disco4",1,token,0,DISCO_1) == 0 )
62. schedule(8,erlang(td,sd),token,DISCO_1);
63. break;

64. case 8:
65. release("disco4",token);
66. schedule(1,clock_sim(),token,CPU);
67. break;

68. }
69. }

70. parsmpl_pvm_exit();
}
```

Figura 4.12: Processo Lógico Disco_1

4.9.2 Sistema Computacional Simplificado

PROCESSO REC1

```
#include "/u/rsulson/pvm3/novom1/parsmpl.c"
#include <sys/time.h>

#define REC1   0
#define REC2   1
```

```
main()
{
  /* variaveis utilizadas para comunicacao com o processo pai */

  int
    tid_processo_pai,
    info,        /* para retorno dos codigos de erro das rotinas do pvm */
    tid_rec1,
    tid_rec2,
    tabela_aux[10];

  float dt2;

  struct timeval t1, t2;
  int dt1;

int
    indice,
    token = 1,     /* identificacao do cliente que utiliza o recurso */
    evento_execut, /* identificacao do evento a ser executado        */
    num_tours = 50000,
    num = num_tours,
    gran,
    option;

 double
   tarec1  = 50.0,
   tsrec1  = 100.0;

  proc_local = 0;
  tempo_processo = 5.0;

  for (indice = 1; indice <10; indice++)
  {
   tabela_tids[indice] = 0;
   tabela_aux[indice] =0;
   }

  /* identifica o processo pai */
  tid_processo_pai = pvm_mytid();

  info = pvm_spawn("rec2",(char **)0,0,"",1,&tid_rec2);

  tabela_tids[REC1] = tid_processo_pai;
  tabela_tids[REC2] = tid_rec2;
  tabela_aux[REC1] = tid_processo_pai;
  tabela_aux[REC2] = tid_rec2;

for (indice = 0; indice < 10 /* maximo_total_canais */; indice++)
   if (indice != REC2)
     clock_canais[indice] = -1.0;

  clock_canais[REC2] = 0.0;

  /* sem codificacao - maquinas homogeneas */
  info = pvm_initsend(PvmDataRaw);

  /* empacota tabela tids */
  info = pvm_pkint(tabela_aux, 10, 1);

  /* faz send da tabela de tids para disco1 */
  info = pvm_send(tid_rec2,1);

 /* registra no grupo filhos */
  info = pvm_joingroup("filhos");
```

```
  /* fica bloqueado ate todos executarem a chamada ao barrier */
  info = pvm_barrier("filhos",2);

  gettimeofday(&t1,(struct timezone*)0);

  smpl(0,"Processo REC1"); /* inicia o sistema */

  /* cria descritores para os recursos*/
  facility("REC1",1);

  /* escalona eventos iniciais */
  schedule(1,0.0,token,REC1);

  while (num_tours) /* limita a execucao da simulacao */
  {

   cause(&evento_execut, &token);

   switch(evento_execut)
   {
    case 1:
     schedule(1,expntl(tarec1),token,REC1);
     schedule(2,0.0,token,REC1);
     for (gran=0; gran < 15000; gran++);
   break;

    case 2:
     if (request("REC1",2,token,0,REC1) == 0)
        schedule(3,expntl(tsrec1),token,REC1);
      for (gran=0; gran < 15000; gran++);
    break;

    case 3:
     release("REC1",token);
     option = randompar(1,10);
     if (option <= 6)
      num_tours--;
     else
     {
     schedule(1,clock_sim(),token,REC2);
     }
     for (gran=0; gran < 15000; gran++);
    break;

    case 4:
     option = randompar(1,10);
     if (option < 5)
      num_tours--;
     else
      schedule(1,clock_sim(),token,REC2);
     for (gran=0; gran < 15000; gran++);
    break;
   }
  }


  gettimeofday(&t2,(struct timezone*)0);

  dt1 = (t2.tv_sec - t1.tv_sec)*1000000 + t2.tv_usec - t1.tv_usec;
  printf("\n Num_tours %d \n",num);
  printf("\n Tempo de execucao (uSegundos): %8d\n",dt1);

  dt2 = (float)(t2.tv_sec - t1.tv_sec) + (float) (((float)(t2.tv_usec -
t1.tv_usec))/1000000);
  printf("\n Tempo de execucao ( Segundos): %f\n",dt2);
```

```
  printf("\n mensagens completas enviadas:  %ld
\n",cont_mens_completas_enviadas);
  printf("\n mensagens nulas     enviadas:  %ld \n",cont_mens_nulas_enviadas);
  printf("\n mensagens completas recebidas: %ld
\n",cont_mens_completas_recebidas);
  printf("\n mensagens nulas     recebidas: %ld
\n",cont_mens_nulas_recebidas);
        .

  printf(" Utilizacao %f\n",utilizacao_recurso("REC1"));
  printf(" Comprimento Medio Fila %f\n",comprimento_medio_fila("REC1"));
  printf(" Periodo M. Ocupado %f\n",periodo_medio_ocupado("REC1"));

  info = pvm_lvgroup("filhos");

  pvm_kill(tid_rec2);

  parsmpl_pvm_exit();
}
```

PROCESSO REC2

```
#include "/u/rsulson/pvm3/novom1/parsmpl.c"

#define REC1 0
#define REC2 1

main()
{
  /* variaveis utilizadas para comunicacao com o processo pai */
  int mytid,
      tid_processo_pai,
      info, /* para retorno de erro das rotinas do pvm */
      indice,
      gran;

  int   contador,
        option;


int token,
     evento_execut;

  double
    tsrec2 = 20.0;

  /* seta variavel proc_local */
  proc_local = REC2;
  tempo_processo = 2.0;

  mytid = pvm_mytid();
  tid_processo_pai = pvm_parent();

/* registra no grupo filhos */
 info = pvm_joingroup("filhos");

 /* recebe tabela de tids do pai - receive bloqueante */
  info = pvm_recv(-1,-1);

 /* desempacota tabela */
  info = pvm_upkint(tabela_tids,10,1);

  /* coloca -1 nas posicoes da tabela que nao vao ser utilizadas */
  for (indice = 2; indice < 10; indice++)
   tabela_tids[indice] = 0;
```

```
  /* coloca -1 nas posicoes do vetor de clock de canais que nao estao sendo
utilizados */
  for (indice = 0; indice < 10; indice++)
   if (indice != REC1)
     clock_canais[indice] = -1.0;

  clock_canais[REC1] = 0.0;

  /* fica bloqueado ate todos executarem a chamada ao barrier */
  info = pvm_barrier("filhos",2);


   smpl(0,"Processo REC2"); /* inicia o sistema */

  /* cria um descritor para o recurso */
  facility("REC2",1);

  contador = 10000;

  while(contador != 0)
  {

   cause(&evento_execut, &token);
   switch(evento_execut)
   {

    case 1:
     if (request("REC2",1,token,0,REC2) == 0)
       schedule(2,expntl(tsrec2),token,REC2);
     for (gran =0; gran < 15000; gran++);
    break;

    case 2:
      release("REC2",token);
      schedule(2,clock_sim(),token,REC1);
      for (gran =0; gran < 15000; gran++);
    break;
   }
  }


 info = pvm_lvgroup("filhos");

 parsmpl_pvm_exit();
}
```

5. Considerações Finais

A extensão *ParSMPL* vem de encontro com essas necessidades, oferecendo ao usuário a possibilidade de utilizar todos os benefícios da simulação distribuída sem exigir conhecimentos específicos. A atenção e o esforço do usuário se restringem ao sistema a ser modelado. As principais características do ambiente que devem ser ressaltadas são: portabilidade, detalhes da comunicação e sincronização transparentes ao usuário e disponibilidade de funções estatísticas e relatório para análise dos resultados.

É importante destacar a portabilidade da extensão *ParSMPL*, que pode ser

facilmente migrada para diversos tipos de plataformas, desde *cluster* de microcomputadores e estações de trabalho até arquiteturas paralelas. Além disso, o desenvolvimento modular do ambiente permite ao usuário a realização de modificações nas características do protocolo por meio de alterações em rotinas específicas, sem a necessidade de alterações em todo o código da extensão.

Deve-se salientar também o fato de *ParSMPL* ser uma extensão funcional, que permite ao usuário o desenvolvimento de programas de simulação com orientação a eventos sem a necessidade do aprendizado de uma nova linguagem.

6. Referências Bibliográficas

FUJIMOTO, R. M. Parallel and Distributed Simulation Systems. John Willey & Sons 2000.

GEIST, A. Advanced Programming in PVM. PROCEEDINGS OF THE 3TH EUROPEAN PVM CONFERENCE, Lecture Notes in Computer Science, n. 1156, p. 1-6,1996.

GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAN, V. PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994.

GEIST, A.; BEGUELIN, A.; DONGARRA, J.; JIANG, W.; MANCHEK, R.; SUNDERAN, V. PVM 3 User's Guide and Reference Manual. Oak Ridge National Laboratory. 1994.

IBM Corporation. IBM PVMe for AIX – User's Guide and Subroutine Reference. Version 2, Release 2, 1996.

MACDOUGALL, M. H. Simulating Computing Systems - Techniques and Tools. The MIT Press, 1987.

MISRA, J. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, v.18, n.1, p.39-65, 1986.

MORSELLI Jr., J. C. M.; ULSON, R. S.; SANTANA, R. H. C.; SANTANA, M. J.. An Approach for Dynamic Swapping of Distributed Simulation Synchronisation Protocols. Proceedings of the European Simulation Symposium (ESS'2000). Hamburg, Germany, 2000.

ULSON, R. S.; SANTANA, R. H. C.; SANTANA, M. J. A Distributed Simulation Environment for Computing Systems Performance Evaluation. Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'97). In: PROCEEDINGS OF THE 1997 SUMMER COMPUTER SIMULATION CONFERENCE, p.85-90. The Society for Computer Simulation International, 1997.

ULSON, R. S.; MORSELLI Jr., J. C. M.; SANTANA, R. H. C; SANTANA, M. J. Conservative Distributed Simulation On Portability Platforms: The CMB Protocol Behavior. In:

PROCEEDINGS OF 1999 APPLIED MODELLING SIMULATION, p. 495-500. IASTED, Austrália, 1999.

ULSON, R. S. Simulação Distribuída Conservativa em Plataformas de Portabilidade: Viabilidade de Uso e Comportamento do Protocolo CMB. Tese (Doutorado), IFSC-USP, dezembro de 1999.