UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANDERSON DA COSTA MORO

Paralelização de uma Aplicação de Transferência Eletrônica de Fundos

Trabalho de Conclusão apresentado como
requisito parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Prof. Marcelo Johann
Orientador

Porto Alegre, Julho de 2012

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Moro, Anderson da Costa

Paralelização de uma Aplicação de Transferência Eletrônica de Fundos / Anderson da Costa Moro. – Porto Alegre: PPGC da UFRGS, 2012.

53 f.: il.

Trabalho de Conclusão (graduação) – Universidade Federal do Rio Grande do Sul. Curso de Ciência da Computação, Porto Alegre, BR–RS, 2012. Orientador: Marcelo Johann.

1. Paralelização. 2. Programação paralela. 3. POSIX. 4. Thread. 5. TEF. 6. POS. I. Johann, Marcelo. II. Título.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

POS	Point of Sale
TEF	Transferência Eletrônica de Fundos
PTHREADS	Posix Threads
GPRS	General Packet Radio Service
Wi-Fi	Wireless Fidelity
POSIX	Portable Operating System Interface
IEEE	Institute of Electrical and Electronics Engineers
VCL	Visual Components Library
TCP	Transmission Control Protocol
LAN	Local Area Network
TPS	Transações por Segundo
DLL	Dynamic Link Lybrary
API	Application Programming Interface
SPC	Serviço de Proteção ao Crédito
PDV	Ponto de Venda
ATM	Asynchronous Transfer Mode
RENPAC	Rede Nacional de Pacotes
CPU	Central Processing Unit
HD	Hard Disk
BIN	Bank Identification Number

LISTA DE FIGURAS

RESUMO

O objetivo desse trabalho de graduação é a paralelização de um aplicativo comercial com a biblioteca *Pthreads* e a análise dos resultados através de um comparativo de desempenho entre o programa sequencial e o programa paralelo.

Espera-se, com o uso da programação paralela, aumentar a eficiência do sistema executado em máquinas *multicore*, verificando um maior número de tarefas processadas por unidade de tempo e redução dos tempos de espera e de processamento de cada tarefa.

Palavras-chave: Paralelização, programação paralela, POSIX, thread, TEF, POS.

ABSTRACT

The objective of this graduate work is the parallelization of a commercial application with the Pthreads library and analysis of results through a comparison of performance between the sequential program and parallel program.

It is expected with the use of parallel programming, to increase the efficiency of the system implemented on multicore machines, achieving a larger number of tasks processed per unit time and reducing the waiting time and processing time of each task.

Keywords: Paralelização, programação paralela, POSIX, thread, TEF, POS.

1 INTRODUÇÃO

Computadores com processadores *multicore* são mais comuns nos dias de hoje, por isso um ganho de desempenho de uma máquina está mais vinculado ao número de *cores* do que ao *clock* desse processador.

Entretanto, não adianta termos máquinas *multicore* se as aplicações executam apenas em uma *thread*. O intuito de criarmos aplicações *multithreads*, ou seja, paralelizarmos as aplicações, é para que essas possam usufruir dos vários núcleos dos processadores, buscando assim um melhor desempenho na execução das suas atividades.

O aplicativo em questão trata da Tranferência Eletrônica de Fundos (TEF), que é um serviço que permite aos clientes efetuarem pagamentos a estabelecimentos comerciais através de transações realizadas com instituições financeiras, utilizando-se de cartões magnéticos. Esse sistema trabalha de forma sequencial, ou seja, os dados recebidos são armazenados em uma lista e processados um após o outro, fazendo com que a aplicação, conforme a quantidade de transações, não aproveite o potencial das máquinas atuais, já que o código da aplicação é serial.

O objetivo desse trabalho é paralelizar uma aplicação com a *API Posix Threads*, objetivando uma melhor *performance* executando-a em máquinas *multicore*.

2 MOTIVAÇÃO

Verificou-se em uma empresa da área da informática que muitas de suas aplicações poderiam ter uma melhora no seu código, o que contribuiria para um melhor aproveitamento do *hardware* atual. Em um dos sistemas dessa empresa, não há vantagem na utilização de máquinas *multicores*, devido ao código fonte das aplicações ser escrito com programação sequencial. Foi analisado que, para alguns clientes, em dias de grande volume de dados, o processamento de uma *CPU* ficava comprometido, alcançando quase 100% de uso. Para tratar esse volume de transações, seria necessário a duplicação do *hardware* e do sistema. Isso acarreta em uma descentralização dos dados, ou seja, as informações para pesquisas, como relatórios, arquivos de manuntenção, entre outros, estarão em máquinas diferentes, o que prejudica a captação desses dados. Além disso, também é necessário um balanceamento das transações que chegam na primeira ou na segunda máquina, o que é uma tarefa dispendiosa.

Com base nesses dados, este trabalho propõe a paralelização desse sistema, de forma a melhorar o desempenho em máquinas *multicore* e trazer uma facilidade ao acesso dos dados da aplicação devido a sua centralização.

Através do contato com colegas da faculdade e de outras empresas da área de informática, percebe-se que muitas aplicações ainda mantém um código sequencial e outras ainda são programadas de forma sequencial, o que torna esse trabalho interessante para programadores que tenham interesse na paralelização de aplicações.

3 FUNCIONAMENTO DA APLICAÇÃO

3.1 Aplicação de Transferência Eletrônica de Fundos

A Transferência Eletrônica de Fundos (TEF) é um serviço oferecido por empresas de informática que permitem a clientes efetuarem pagamentos a um estabelecimento comercial através de transações realizadas com instituições financeiras, utilizando-se de cartões magnéticos ou ainda cartões com *chip*. Basicamente, pode-se realizar os pagamentos através de cartões de crédito, cartões de débito e cartões de alimentação. Além dessas possibilidades, pode-se realizar consultas cadastrais ao SPC e ao Serasa, pagamento de contas, saque e outras transações bancárias.

O processo de pagamento utilizando TEF é mais seguro e prático do que as outras formas de pagamento, pois não envolve valores em moeda corrente na loja, isto é, os valores são transferidos eletronicamente da conta do cliente para a conta do lojista.

A TEF necessita basicamente de 4 aplicativos: *Checkout*, *Conversor Client*, TEF e Rede Autorizadora.

3.1.1 *Checkout*

Trata-se de um aplicativo que inicia uma transação TEF através do portador do cartão que interage com essa aplicação. Podemos ter vários tipos de *checkouts*, conforme o tipo de negócio do estabelecimento: *Client PDV*, *Client Móvel*, *Client Telemarketing* e *Client ATM*. Nesse trabalho, o estudo foi concentrado no aplicativo *Client Móvel*, que tem como funções:

- obter a operação desejada, identificando se é uma operação com cartão de crédito, débito, etc, bem como o tipo de transação (compra, consulta, pagamento de contas, entre outras);
- realizar a leitura do cartão magnético ou *chip* do portador, quando necessário;
- realizar a leitura da senha do portador, quando necessário;
- imprimir comprovante da operação TEF, quando necessário;
- realizar a comunicação com o *Conversor Client*.

3.1.1.1 Client Móvel

Essa solução possibilita realizar uma TEF através de um *POS* (Point of Sale) sem fio e com comunicação *GPRS*, o que possibilita total mobilidade na transação, visto que utiliza o serviço de dados das operadoras de telefonia celular. Também podemos ter um *POS* sem fio com comunicação *Wi-Fi*.

3.1.2 *Conversor Client*

Aplicativo que centraliza as transações dos *Checkouts* para preparar as mensagens conforme a especificação de cada Rede Autorizadora. Esse componente também pode ser encontrado pelo nome de *Conversor POS* e tem como funções:

- receber as mensagens dos diversos *checkouts*;
- rotear as mensagens originadas dos diversos *checkouts*;
- controlar o fluxo de mensagens de solicitação e de resposta entre os *checkouts* e o TEF.

3.1.3 TEF Dedicado

Aplicativo que formata e roteia as transações para cada Rede Autorizadora com conexão dedicada. Além disso, fornece todo o controle para consultas, tratamento de pendências e conciliação financeira, possuindo como funções:

- receber as parametrizações necessárias;
- receber as mensagens dos diversos *checkouts* através do *Conversor Client*;
- formatar as mensagens para a rede autorizadora;
- estabelecer o *link* de comunicação junto à RENPAC (ou rede similar);
- conectar-se à rede autorizadora;
- controlar o fluxo de mensagens de solicitação e resposta entre o *Conversor Client* e a rede.

3.1.4 Rede Autorizadora

Aplicativo final que aprova ou nega as transações de cada estabelecimento. É ele que aprova o crédito do portador, emite consultas e realiza todas as transações financeiras.

3.1.5 Topologia da Aplicação

A figura 3.1 mostra a topologia da aplicação, ou seja, a transação partindo do *POS* até a Rede Autorizadora e o seu retorno até o *POS*.

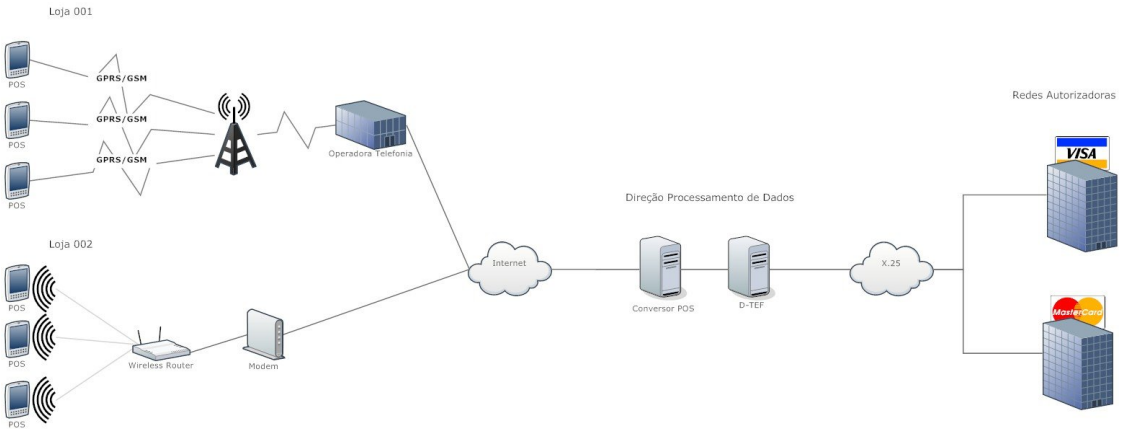

Figura 3.1: Topologia da Aplicação

3.1.6 Motivos da Paralelização

O aplicativo *Conversor POS* recebe, através de conexões com *sockets*, os dados de cada *POS* e/ou redes autorizadoras, de modo que esses dados sejam processados de forma sequencial. Ou seja, os dados recebidos são armazenados em uma lista e processados um após o outro, fazendo com que a aplicação, conforme a quantidade de transações, não tenha uma boa performance. Em inúmeras dessas transações realizadas o aplicativo final (*checkout POS*) não recebe resposta, finalizando por *timeout* ou recebendo uma resposta após muitos segundos. Isso implicará em filas e, consequentemente, no descontentamento do cliente final.

Pode-se dimensionar esse problema descrevendo um cliente como a Petrobrás, que possui cerca de 6000 postos em todo o Brasil e com média de 2 *Checkouts* por posto. Em horário de "pico", o servidor do aplicativo *Conversor POS* pode receber cerca de 150 transações por segundo. Uma máquina com um processador de apenas 1 core consegue receber cerca de 50 transações por segundo, o que é pouco, fazendo com que seja necessário aumentar o número de máquinas e, por consequência disso, descentralizar os dados. Com a paralelização da aplicação, pode-se utilizar apenas uma máquina com um processador e mais *cores*, melhorando a performance da aplicação.

3.1.7 Fluxo Transacional da Aplicação

A figura 3.2 mostra todo fluxo transacional, ou seja, o fluxo das transações de solicitação, resposta e confirmação. A transação de solicitação é iniciada com a coleta do cartão e o envio de uma transação de consulta. O aplicativo ConversorPOS recebe essa consulta e, através de consultas a tabelas locais que possuem parâmetros de configuração de cada BIN (seis primeiros digitos do cartão), envia uma resposta ao POS. Essa resposta possui fluxos de captura de alguns dados que devem ser coletados pelo operador do POS. Após a coleta dessas informações, o POS envia uma nova transação que será encaminhada à rede autorizadora. Após verificar a validade do cartão e outras informações, a rede autorizadora envia a resposta para o sistema TEF que redireciona a mensagem para o POS. Após o recebimento da transação, o POS imprime um comprovante para o cliente final e envia uma transação de confirmação à rede autorizadora.

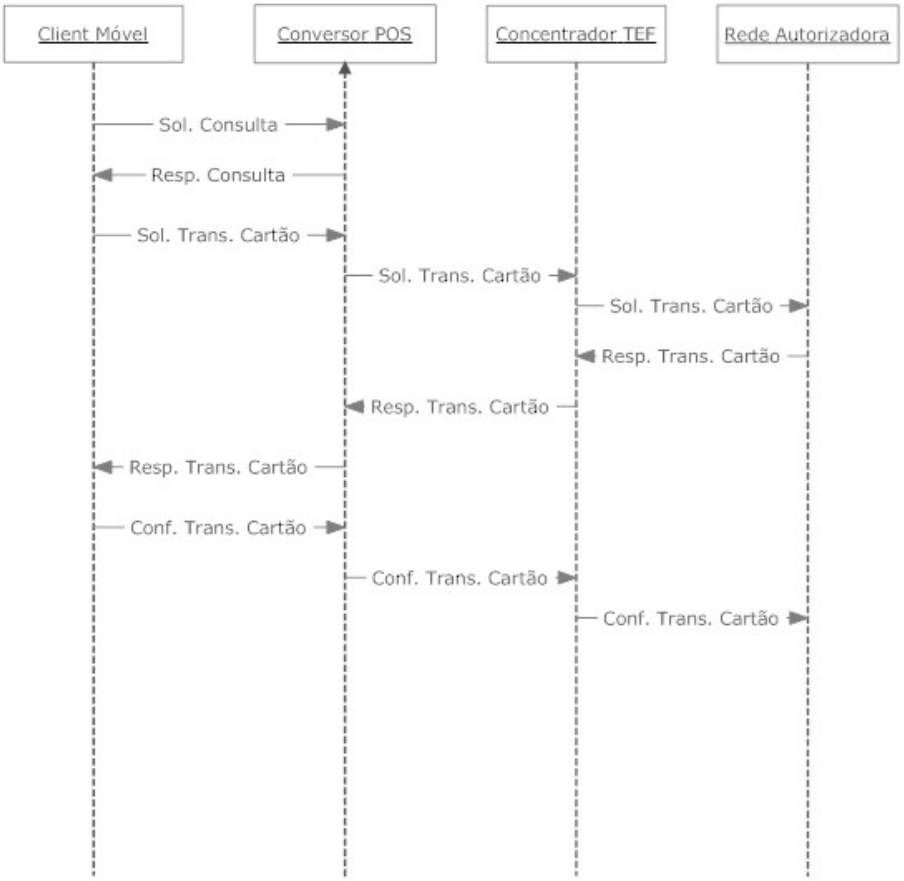

Figura 3.2: Fluxo Transacional

4 FUNDAMENTAÇÃO TEÓRICA

4.1 Programação Paralela em Outros Trabalhos

O crescimento do uso de computadores com vários processadores incentivou o desenvolvimento de programas com processamento paralelo, com capacidade de processamento muito maior.

A programação tradicional, ou em série, trata as tarefas de forma sequencial, ou seja, resolvendo tarefa após tarefa. Já na computação em paralelo são realizadas várias tarefas em simultâneo, uma por processador, aproveitando os vários processadores disponíveis nas arquiteturas atuais.

Sendo assim, a paralelização tornou-se um assunto comum entre os estudantes de informática que buscam nela soluções para problemas de desempenho enfrentados com programas sequenciais. O uso de *threads* é a opção mais direta para o aproveitamento de computadores com múltiplas *CPUs* e memória compartilhada, pois seu modelo coincide exatamente com o modelo da arquitetura de *hardware* destas máquinas, com memória compartilhada (MORALES, 2009).

Muitos trabalhos nessa área são realizados e podemos destacar alguns que paralelizam algoritmos (BRINKHUS, 2009) e outros que analisam o desempenho da programação paralela (PILLA, 2009).

4.2 *Threads*

Uma *thread* é um fluxo de execução dentro de um processo. Esse processo pode se "dividir" em vários fluxos que podem ser executados concorrentemente, sendo chamado de *multithread* quando essa divisão acontece, possibilitando, assim, executar várias tarefas de forma independente.

As *threads* podem ser implementadas de duas formas. A primeira são as *threads* de nível de usuário, que são criadas e manipuladas pelo *software*. A segunda são as *threads* a nível de núcleo (*kernel*), que são utilizadas pelo Sistema Operacional.

4.2.1 Gerenciamento de *Threads*

É necessário conhecermos algumas operações básicas para o gerenciamento das *threads*: criação, término, junção e suspensão.

- Criação: os processos normalmente iniciam com apenas uma *thread* e essa *thread* tem a capacidade de criar novas *threads*. Não é possível especificar o espaço de endereço da nova *thread*, pois ela executa no endereço da *thread* criadora. À *thread* criadora é retornada um identificador da *thread* em criação. Após sua criação uma

thread pode ocupar um desses quatro estados: pronto, executando, bloqueado e terminado. (CARISSIMI, 2004)

- Término: após o término do seu trabalho, a *thread* pode ser encerrada. Isso não serve para todos os casos, mas geralmente é chamado um procedimento da biblioteca de *threads* que encerra sua execução, por exemplo, *thread_exit*. Além disso, é necessário ressaltar que, geralmente, se a *thread* criadora terminar, todas as *threads* filhos também serão terminadas.
- Junção: é destinado ao caso em que uma *thread* criadora precisa esperar a conclusão da execução de uma ou algumas das *threads* filhas.
- Suspensão: permite que uma *thread* desista voluntariamente da *CPU* para deixar outra *thread* executar.

4.2.2 Sincronização

Frequentemente *threads*, assim como processos, precisam trocar informações entre si. Pode-se destacar dois tópicos em relação a isso: como as *threads* não invadem umas as outras quando envolvidas em atividades críticas e também a relação de dependência entre as *threads*.

4.2.2.1 Condições de Disputa

Em alguns sistemas operacionais, processos que trabalham juntos podem compartilhar algum armazenamento comum, a partir do qual cada um é capaz de ler e escrever. O armazenamento compartilhado pode estar na memória principal (possivelmente em uma estrutura de dados do núcleo) ou em um arquivo compartilhado; o local da memória compartilhada não altera a natureza da comunicação ou dos problemas que surgem. Situações como a em que dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende das informações de quem e quando executa precisamente são chamadas de condições de disputa (*race conditions*). A depuração de programas que contenham condições de disputa não é nada divertida. Os resultados da maioria dos testes não apresentam problemas, mas uma hora, em um momento raro, algo estranho e inexplicável acontece. (TANENBAUM, 2003)

4.2.2.2 Regiões Críticas

Para evitar condições de disputa aqui e em muitas outras situações que envolvam memória compartilhada e arquivos compartilhados deve-se encontrar algum modo de impedir que mais de um processo leia e escreva ao mesmo tempo na memória compartilhada. Em outras palavras, precisamos de exclusão mútua (*mutual exclusion*), isto é, assegurar que outros processos sejam impedidos de usar uma variável ou um arquivo compartilhado que já estiver em uso por um processo.

O problema de evitar condições de disputa pode também ser formulado de um modo abstrato. Durante uma parte do tempo, um processo está ocupado fazendo computações internas e outras coisas que não acarretam condições de disputa. Contudo, algumas vezes, um processo precisa ter acesso à memória ou a arquivos compartilhados ou tem de fazer outras coisas críticas que podem ocasionar disputas. Aquela parte do programa em que há acesso à memória compartilhada é chamada de região crítica (*critical region*) ou seção crítica (*critical section*). Se pudéssemos gerenciar o processamento de modo que nunca dois processos estivessem em suas regiões críticas ao mesmo tempo, as disputas seriam evitadas (TANENBAUM, 2003).

4.2.2.3 Semáforos

Um semáforo é um recurso oferecido pelo sistema operacional que consiste em um número inteiro e em uma fila que armazena descritores de tarefas. O conceito de semáforos consiste na colocação de proteções (guardas) em torno do código que acessa esta estrutura para oferecer acesso limitado aos dados (SEBESTA, 2000). Em geral, a estrutura de dados é uma fila, funcionando em regime de primeiro-a-entrar/primeiro-a-sair (GHEZZI, 1991).

4.2.2.4 Mutexes

Quando não é preciso usar a capacidade do semáforo de contar, lança-se mão de uma versão simplificada de semáforo, chamado *mutex*. *Mutexes* são adequados apenas para gerenciar a exclusão mútua de algum recurso ou parte de código compartilhada. São fáceis de implementar e eficientes, que os torna especialmente úteis em pacotes de *threads* implementados totalmente no espaço do usuário.

Um *mutex*, abreviação de *mutual exclusion*, é uma variável que pode estar em um dos dois estados seguintes: desimpedido ou impedido. Consequentemente, somente 1 *bit* é necessário para representá-lo, mas, na prática, muitas vezes se usa um inteiro, com 0 para desimpedido e qualquer outro valor para impedido. Dois procedimentos são usados com *mutexes*. Quando uma *thread*, ou processo, precisa ter acesso a uma região crítica, ele chama *mutex_lock*. Se o *mutex* estiver desimpedido, indicando que a região crítica está disponível, a chamada prosseguirá e a *thread* que chamou *mutex_lock* ficará livre para entrar na região crítica.

Por outro lado, se o *mutex* já estiver impedido, a *thread* que chamou *mutex_lock* permanecerá bloqueada até que a *thread* na região crítica termine e chame *mutex_unlock*. Se múltiplas *threads* estiverem bloqueadas sobre o *mutex*, uma delas será escolhida aleatoriamente e liberada para adquirir o impedimento (TANENBAUM, 2003).

4.3 Problemas convertendo Código *Monothread* em Código *Multithread*

Geralmente, quando há um transbordo da pilha de um processo, o núcleo apenas garante que o processo ganhará mais espaço na pilha. Quando temos múltiplas *threads*, um processo também deve ter múltiplas pilhas. Se o núcleo não souber dessas múltiplas pilhas ele não poderá garantir mais pilhas.

Outro problema é que muitas bibliotecas não são reentrantes, ou seja, não estão preparadas para mais de uma chamada aos seus procedimentos ao mesmo tempo.

Normalmente, o código de uma *thread* executa vários procedimentos que podem possuir muitas variáveis locais, parâmetros de procedimentos e variáveis globais. As duas primeiras não geram problemas, mas as variáveis globais sim. O problema geralmente acontece quando mais de uma *thread* utiliza essa variável ao mesmo tempo. Como exemplo podemos citar uma variável global X que é modificada por uma *thread* A e que deveria ser lida no final de um procedimento dessa *thread*, mas, antes dessa leitura, uma *thread* B ganha o controle da *CPU* e modifica o valor de X, causando assim um problema quando a *thread* A retomar o controle e for ler o valor da variável X.

4.4 *POSIX Threads*

POSIX é o nome de uma coleção de padrões relacionados e especificados pela *IEEE*, que define diversas interfaces de programação e utilização de sistemas operacionais. Um desses padrões é o *POSIX Threads*, ou *Pthreads*.

A biblioteca *Pthreads*, portável de *threads*, é a interface padrão no *Linux* e também é usada na maioria das plataformas *Unix*. Existe uma versão de código aberto disponível para *Windows* (PTHREADSWIN32, 2012), a qual usaremos para a paralelização da aplicação a que se refere esse trabalho.

4.4.1 Criação e Junção de *Threads* com a Biblioteca *Pthreads*

A criação de *threads* na biblioteca *Pthreads* é feita através da função *pthread_create*:

```
pthread_create(&id, atributos, nome_da_funcao, param);
```

O primeiro argumento é uma variável do tipo *pthread_t* à qual é atribuído o identificador da *thread*. O segundo argumento especifica os atributos da *thread*. O terceiro argumento é o nome da função que será executada ao iniciar a *thread*. O quarto argumento é usado para passar parâmetros para a função do terceiro argumento.

A operação *join* é realizada pela função *pthread_join*:

```
pthread_join(id_da_thread, retval);
```

O primeiro argumento é uma variável do tipo *pthread_t*, que representa o identificador da *thread* que deve ser esperada. Ao executar esta função, o Sistema Operacional faz com que a *thread* atual fique bloqueada até que a *thread id_da_thread* termine sua execução. O segundo argumento é um *buffer* do tipo *void***, onde é colocado o valor de retorno da função.

4.4.2 Sincronização

Será exemplificado o uso de um *mutex* da biblioteca *Pthreads*. A biblioteca utiliza as funções *pthread_mutex_lock* e *pthread_mutex_unlock*. É passado como parâmetro uma variável do tipo *pthread_mutex_t* para as duas funções que bloqueiam e desbloqueiam o *mutex*, respectivamente.

O *mutex* é útil para resolver problemas clássicos de programação concorrente como o do produtor e consumidor.

5 IMPLEMENTAÇÃO DO CÓDIGO *MULTITHREAD*

Esse capítulo descreve as dificuldades de portar um código *monothread* para *multithread* e detalha as modificações realizadas no código fonte da aplicação.

5.1 Arquitetura do Aplicativo *Conversor POS*

O aplicativo foi desenvolvido na *IDE Borland C++ Builder 6.0* e baseia-se em "eventos", ou seja, para cada mensagem que o aplicativo recebe de um *Checkout POS* ou de uma rede autorizadora, um evento é gerado e adicionado em uma lista de eventos. Sequencialmente, os eventos são lidos dessa lista e processados, conforme o código abaixo:

```
void __fastcall CConversorPOS::tmTimerTimer(TObject *Sender)
{
  CEvento   *oEvento;
  bool bFim = false;
  AnsiString sMensagemErro;
  AnsiString sMensagem;

  tmTimer->Enabled = false;
  try
  {
    while (!bFim)
    {
      if (Application != NULL)
        Application->ProcessMessages();
      else if (Service != NULL)
        Service->ServiceThread->ProcessRequests(false);
      oEvento = NULL;
      oEvento = BuscaEvento();
      if (oEvento == NULL)
        break;
      // processa o evento. Se for evento de finalização ou erro,
      //sai do loop
      if (ProcessaEvento(oEvento) != 0)
        bFim = true;
      delete oEvento;
    }
  }
```

```
  catch (Exception &e)
  {
    sMensagem = "Exceção: " + e.Message;
    GravaErro(__FUNC__, sMensagem);
    oPrintDeb->DumpStackTrace();
    if(oEvento == NULL)
      delete oEvento;
  }
  if (!bFim)
    tmTimer->Enabled = true;
}
```

A figura 5.1 representa a arquitetura da aplicação *Conversor POS*.

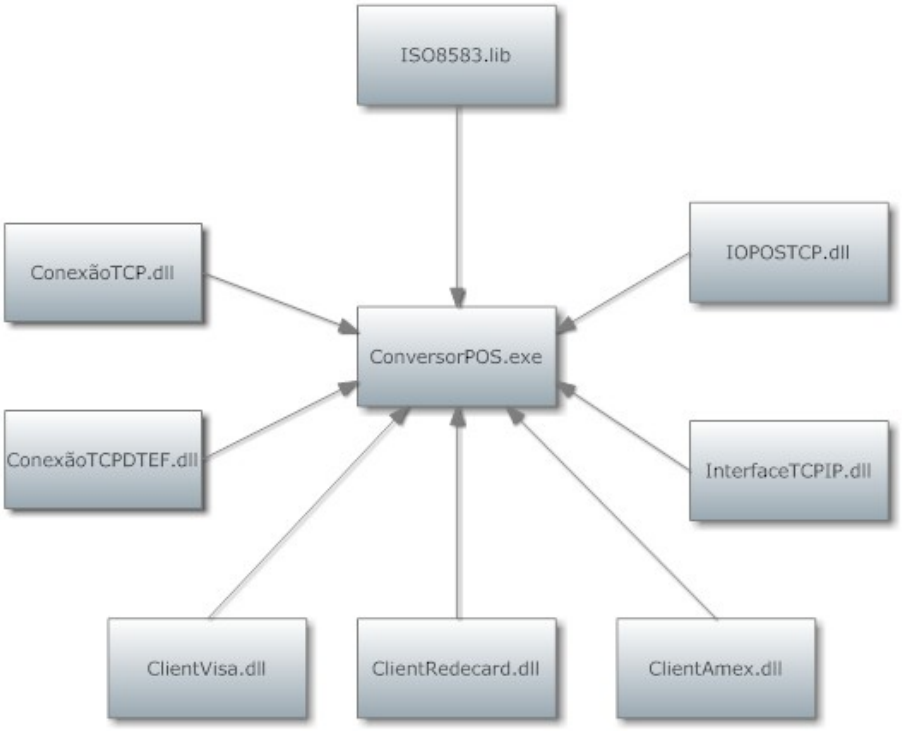

Figura 5.1: Arquitetura da Aplicação

5.2 Análise para Implementação

Analisando a aplicação, verifica-se a possibilidade da criação de *threads* para o processamento da fila de eventos, que hoje é processada sequencialmente. Por exemplo, se o aplicativo receber 4 transações ao mesmo tempo, em vez de processá-las uma a uma, pode-se criar uma rotina como a abaixo para "consumir" ou "processar" os eventos presentes na lista.

```
void CConversorPOS::CriaObjeto(void *pHandle)
{
...
    pthread_create(&m_threadId[i], NULL, NovaThread, this);
...
```

```
}

void *CConversorPOS::NovaThread(void *arg)
{
  CEvento   *oEvento;
bool bFinalizaThread = false;

  while(bFinalizaThread == false)
  {
    oEvento = ((CConversorPOS *)arg)->BuscaEvento();
    if( oEvento != NULL )
    {
      if (((CConversorPOS *)arg)->ProcessaEvento(oEvento) != 0)
        bFinalizaThread = true;

        delete oEvento;
        oEvento = NULL;
    }
  }
  pthread_exit((void *)0);
}
```

Para realizar essa alteração é necessário tratar alguns dos problemas citados no capítulo de conversão de código *monothread* para *multithread*. Analisando a figura 5.1, verifica-se a utilização de algumas bibliotecas que, após testes iniciais, com as modificações do código acima, fizeram com que o programa travasse. Isso ocorreu por essas bibliotecas não serem reentrantes, como explicado no capítulo 4.

Outro problema é que o próprio objeto criado da classe *CConversorPOS* é passado como parâmetro do método *NovaThread* e isso implica na modificação das variáveis compartilhadas por procedimentos utilizados dentro da *thread*.

5.3 Atividades

5.3.1 Criação da *DLL pthread* para *Windows*

Não existia a biblioteca *pthreads* compatível com o compilador da *Borland*. Com os fontes e o *Makefile* disponíveis foi possível gerá-la através dos passos listados abaixo.

- Instalação do *Cygwin* (CYGWIN, 2012)
- Execução do comando *"make -fBmakefile"* que gerou o erro *''Error version.rc 33 11: Cannot open file: winver.h"*
- Modificação do arquivo *Bmakefile* incluindo o *path* do arquivo *winver.h*
- Execução do comando *make -fBmakefile* que gerou os arquivos *"pthreadBC2.dll"* e *"pthreadBC2.lib"*

5.3.2 Adicionando Biblioteca no Projeto *Conversor POS*

Após a geração da *DLL pthreadBC2* é necessário a inclusão dessa biblioteca no projeto do *Conversor POS* e a cópia dos arquivos *"headers"* disponibilizados pela (PTHREADSWIN32, 2012) para o diretório de instalação do *C++ Builder*.

As figuras 5.2, 5.3, 5.4, 5.5 e 5.6, respectivamente nas páginas 22, 22, 23, 23 e 23, ilustram esses procedimentos.

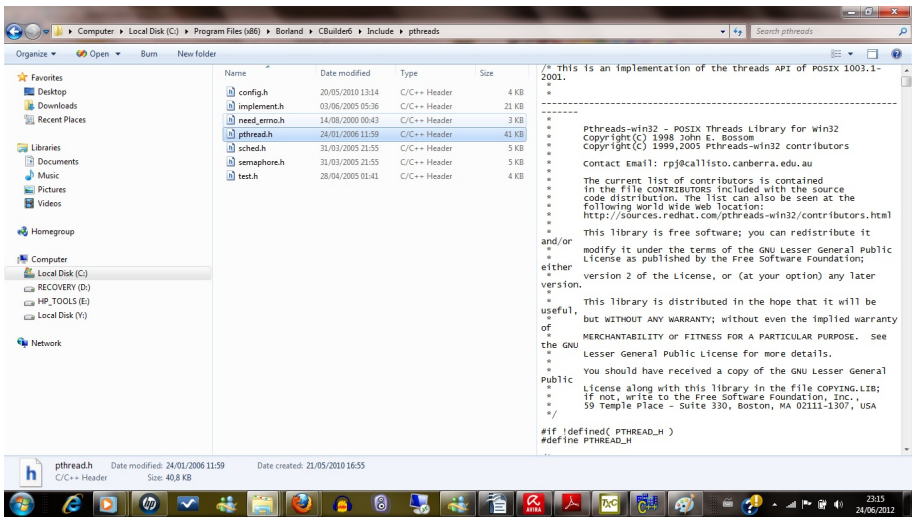

Figura 5.2: Copiando Arquivos

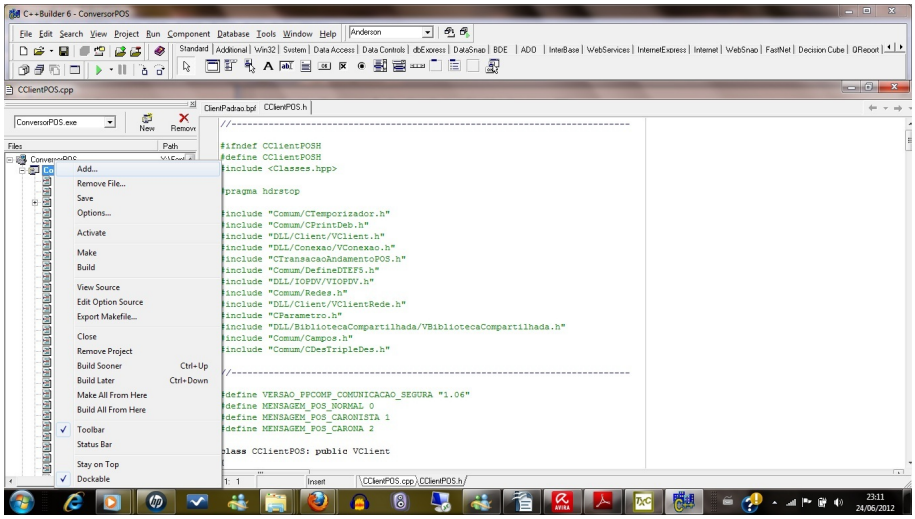

Figura 5.3: Adicionando Biblioteca ao Projeto

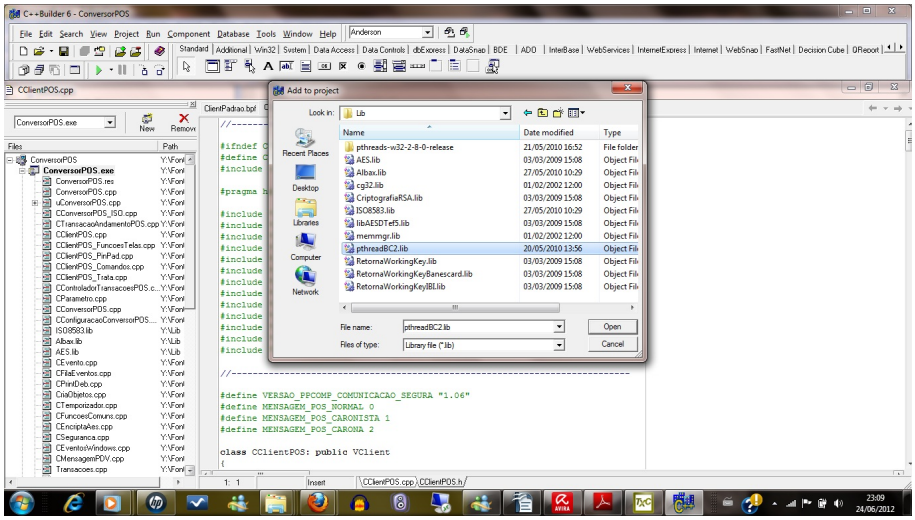

Figura 5.4: Adicionando Biblioteca ao Projeto

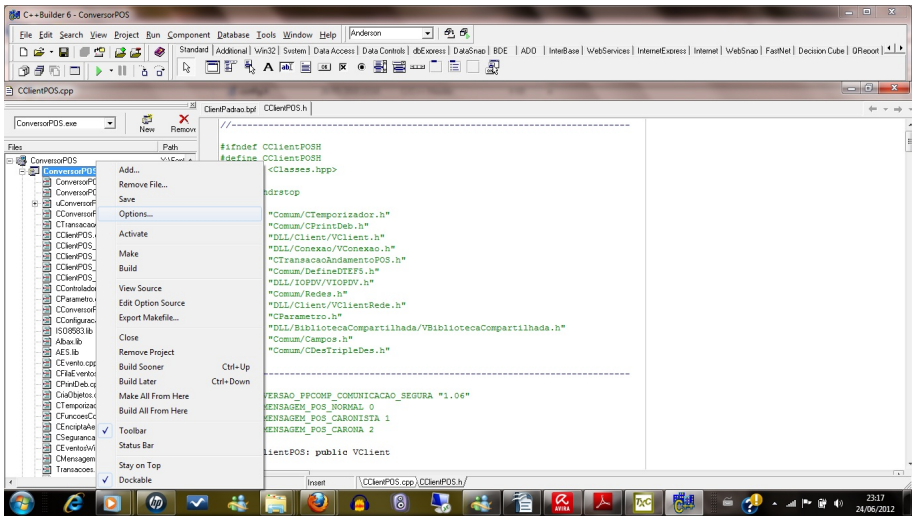

Figura 5.5: Configurando Projeto para uso da Biblioteca

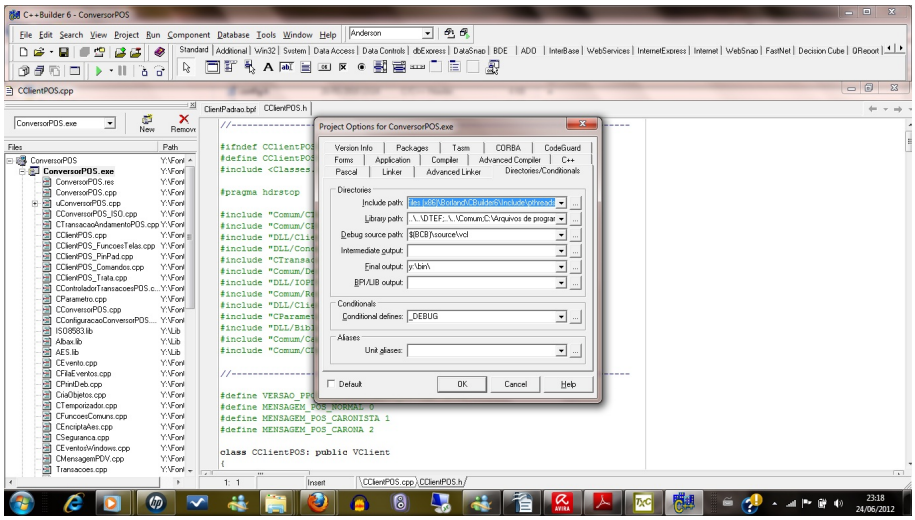

Figura 5.6: Configurando Projeto para uso da Biblioteca

5.3.3 *Pthreads* x *VCL*

Após a paralelização do código e alguns testes, verificou-se o travamento da aplicação. A figura 5.7 mostra parte do guia do usuário do *C++ Builder*. Ele diz que alguns componentes da *VCL* não são *thread-safe*, ou seja, não garantem que várias *threads* executarão simultaneamente de maneira correta. Ele apresenta uma solução que seria utilizar o método *Synchronize* para garantir *thread-safe*, mas esse método não existe na biblioteca *pthreads*. Foi necessário então modificar todas as *dlls* que utilizavam algum compononente da biblioteca *VCL*.

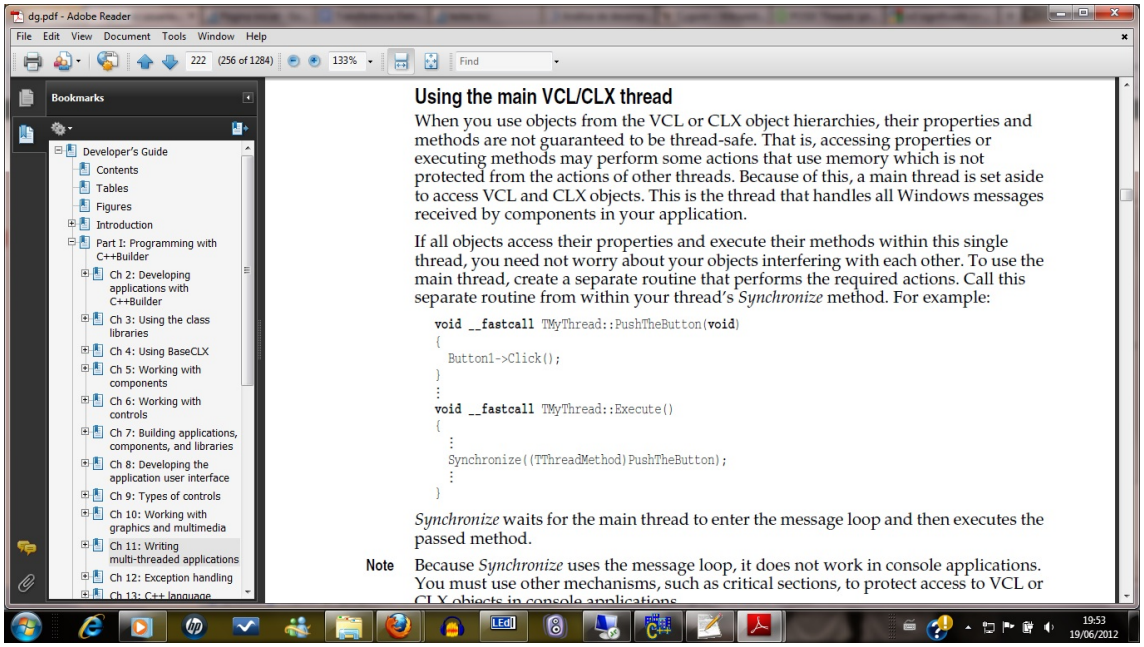

Figura 5.7: *Threads* x *VCL*

5.3.4 Análise e Modificações da Biblioteca *conexaotcp.dll*

A *dll conexaotcp.dll* utilizava o componente *TForm* da *VCL*, conforme pode ser visto na figura 5.8. Pela análise realizada, o uso do *TForm* era apenas para a criação do componente *TTimer*, mas pode-se criá-lo dinamicamente através da chamada *new*.

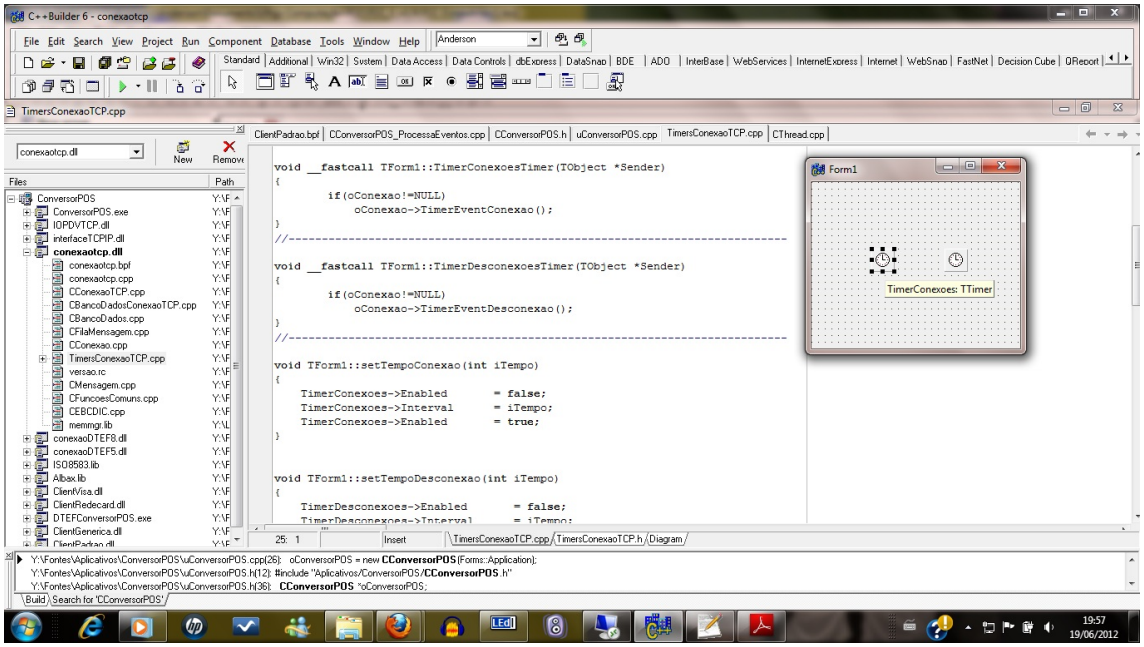

Figura 5.8: *DLL ConexãoTCP*

Foi modificada a classe *CConexao* retirando a declaração da variável do tipo *TForm1* e adicionando a declaração de duas variáveis do tipo *TTimer*, conforme o código abaixo.

```
class CConexaoTCP : public CConexao
{
  private:
    CBancoDadosConexaoTCP *oBancoDadosConexaoTCP;
    void*   HandleOriginal;
    //código comentado para não utilizarmos a VCL
    //TForm1  *FormTimers;
    //código adcionado para não utilizarmos a VCL
    //criando timers que eram gerados diretamente no Form
    TTimer *TimerConexoes;
    TTimer *TimerDesconexoes;
    void __fastcall TimerConexoesTimer(TObject *Sender);
    void __fastcall TimerDesconexoesTimer(TObject *Sender);

  public:
        CConexaoTCP();
        ~CConexaoTCP();

        int Inicializa(void *Handle, VFilaEventos* oFilaEventos,
        VInterface* oInterface, VPoolConexoesBD *oPoolConexoesBD,
        VPrintDeb *oPrintDeb, AnsiString &sMensagemErro);
        int     Finaliza();
        int     Reconfigura(VConexao *oConexao);
        int     InicialistenPDV();
        int     getStatusConexao();
        //não é mais necessário, pois não criaremos Form
```

```
        //void    CriaForm(void* Handle);
        //void    DestroiForm();
        void    setTempoConexao(int iTempo);
        void    setTempoDesConexao(int iTempo);
        AnsiString getStringConexao();
        void    getTabelasParaAuditoria(TStringList *stlTabelas);
};
```

No método construtor da classe foi criado dinamicamente o objeto *TTimer*.

```
CConexaoTCP::CConexaoTCP() : CConexao()
{
  setInterface((VInterface*) NULL);
  CFilaMensagem  *oFilaMensagem;
  oFilaMensagem=new  CFilaMensagem;
  setFilaMensagem(oFilaMensagem);
  HandleOriginal=NULL;
  oBancoDadosConexaoTCP = NULL;
  setEndereco("");
  setPorta(0);

  TimerConexoes = new TTimer(NULL);
  TimerConexoes->OnTimer = TimerConexoesTimer;
  TimerConexoes->Enabled = false;
  TimerConexoes->Interval = 3000;

  TimerDesconexoes = new TTimer(NULL);
  TimerDesconexoes->OnTimer = TimerDesconexoesTimer;
  TimerDesconexoes->Enabled = false;
  TimerDesconexoes->Interval = 3000;
}
```

5.3.5 Análise e Modificação da Classe *CPrintDeb*

A classe *CPrintDeb* cria e atualiza todos os dados necessários em um arquivo de auditoria. Esse arquivo serve como base de consulta para o programador, que consegue analisar através dele o funcionamento do aplicativo, podendo encontrar toda a sequência de execução da aplicação.

A classe *CPrintDeb* é instanciada apenas uma vez e essa instância é quem controla toda a gravação. Como nossas *threads* poderiam utilizar esse objeto de forma simultânea, utilizou-se o mecanismo *mutex* para que a gravação das informações no arquivo de auditoria fosse acessada exclusivamente por cada *thread*.

Parte do código fonte que cria a variável *mutex* encontra-se a seguir.

```
class CPrintDeb : public VPrintDeb
{
  private:
    TDateTime   dtDataAtual;
    AnsiString  sDiretorio;
```

```
    int         iDiasManutencaoArquivos;
    bool        bNaoCriptografa;
    bool        bCriptografiaForte;
    AnsiString  sBaseDebug;
    AnsiString  sArquivoDebug;
    AnsiString  sArquivoErro;
    AnsiString  sArquivoSQL;
    FILE        *fd_debug;
    FILE        *fd_debug_erro;
    FILE        *fd_debug_sql;
    CEncriptaAes *oEncriptaAes;

    pthread_mutex_t mutex;
  public:
    CPrintDeb(AnsiString sBaseDebug);
    ~CPrintDeb();

    int         Inicializa(int iNivel, AnsiString sDiretorio, int
iDiasManutencaoArquivos, bool bNaoCriptografa);
    void        Finaliza();
    void        Grava(int iNivelChamada, const char *fmt, ... );
    AnsiString  getDiretorio();
};
```

Parte do código fonte em que se inicializa a variável *mutex* no construtor da classe *CPrintDeb* verifica-se abaixo.

```
CPrintDeb::CPrintDeb(AnsiString sBaseDebug)
{
  ...
this->sBaseDebug = sBaseDebug;
  if (Trim(this->sBaseDebug) == "")
    this->sBaseDebug = "POS";

  fd_debug = NULL;
  fd_debug_erro = NULL;
  fd_debug_sql = NULL;

  //inicializando a variável mutex
  pthread_mutex_init(&mutex, NULL);
...
}
```

A seguir, encontra-se parte do código fonte em que se protege a região crítica através das chamadas *pthread_mutex_lock* e *pthread_mutex_unlock*, para que apenas uma *thread* acesse os dados.

```
void CPrintDeb::Grava(int iNivelChamada, const char *fmt, ...)
{
  char pcDataHora[200];
```

```
  char Chave[20] = "1234567890123456";
  int iTamBufferOut;
  AnsiString s;

  //trava mutex
  pthread_mutex_lock(&mutex);
  if (iNivelChamada <= iNivel)
  {
    TrocaData(false);

    if (fd_debug == NULL)
      fd_debug = fopen(sArquivoDebug.c_str(), "ab");

    if (fd_debug != NULL)
    {
      sprintf(pcDataHora, "%s",
FormatDateTime("dd/mm/yyyy hh:nn:ss:zzz", Now()).c_str());

      va_list ap;
      va_start(ap, fmt);
      s.vprintf(fmt, ap);
      va_end(ap);

      s = AnsiString(pcDataHora) + s;

      if (!bNaoCriptografa) // se falso , criptografa
      {
        // encripta buffer
        char *pBufferOut = new char[2 * s.Length() + 64];

        oEncriptaAes->CriptografaBufferN(s.Length(), s.c_str(),
Chave, &iTamBufferOut, pBufferOut);
        fwrite(pBufferOut, iTamBufferOut,1,fd_debug);
        delete [] pBufferOut;
      }
      else   // se verdadeiro , nao criptografa
      {
        fwrite(s.c_str(), s.Length(), 1, fd_debug);
      }

      Descarrega();

      if (!bManterArquivoAberto)
      {
        fclose(fd_debug);
        fd_debug = NULL;
      }
    }
```

```
  }
  //destrava mutex
  pthread_mutex_unlock(&mutex);
}
```

5.3.6 Análise e Modificação da Classe *CFilaEventos*

O aplicativo *Conversor POS*, como dito anteriormente, baseia-se em "eventos", ou seja, para cada mensagem que o aplicativo recebe de um *Checkout POS* ou de uma Rede Autorizadora, um evento é gerado e adicionado a uma lista de eventos. Essa lista de eventos possui algumas regiões críticas em que se adicionam os eventos e realiza-se a leitura/remoção deles. Devido a isso é necessária a criação de *mutex* para que cada *thread* acesse esses dados de forma exclusiva.

Abaixo segue o código fonte da classe *CFilaEventos* com a criação do *mutex* através do tipo de variável *pthread_mutex_t*.

```
class CFilaEventos: public VFilaEventos
{
  private:
    TList *oFila;
    pthread_mutex_t mutex;

  public:
    CFilaEventos();
    ~CFilaEventos();
    void InsereEvento(int iCodigo, int iOrigem, int iTamanho,
                  void pDados);
    CEvento *Remove();
    int ElementosDaFila();
};
```

No método construtor da classe inicializa-se a varável *mutex*.

```
CFilaEventos::CFilaEventos()
{
  oFila = new TList();
  //inicializando a variável mutex
  pthread_mutex_init(&mutex, NULL);

}
```

No método de inserção e leitura/remoção foi protegido a lista de eventos com as funções *pthread_mutex_lock(&mutex)* e *pthread_mutex_unlock(&mutex)*.

```
void CFilaEventos::InsereEvento(int iCodigo, int iOrigem,
                                int iTamanho, void *pDados)
{
  CEvento *oEvento;

  oEvento = new CEvento(iTamanho);
```

```
  memcpy(oEvento->oDados, pDados, iTamanho);
  oEvento->iCodigo = iCodigo;
  oEvento->iOrigem = iOrigem;

  pthread_mutex_lock(&mutex);
  oFila->Add(oEvento);
  pthread_mutex_unlock(&mutex);
}

CEvento* CFilaEventos::Remove()
{
  CEvento *oEvento;

  pthread_mutex_lock(&mutex);

  if (oFila->Count > 0)
  {
    oEvento = (CEvento *) oFila->Items[0];
    oFila->Delete(0);
    pthread_mutex_unlock(&mutex);
    return oEvento;
  }

  pthread_mutex_unlock(&mutex);
  return NULL;
}
```

5.3.7 Análise e Modificação das *DLLs* das Redes Autorizadoras

O aplicativo criava apenas uma instância das *DLLs* das Redes Autorizadoras, o que causava travamentos dentro dessas *DLLs*, pois as *threads* acessavam o mesmo objeto ao mesmo tempo. Esse problema foi resolvido criando uma instância para cada *thread* que acessa a *DLL*. No código fonte antigo, a *DLL* era instanciada na criação dos objetos da classe *CConversorPOS*. Apenas um objeto era criado, como mostrado abaixo em parte do código.

```
void CConversorPOS::CriaObjetos(void *pHandle)
{
  oControladorTransacoes = new CControladorTransacoesPOS();

  oFilaEventos = new CFilaEventos();
  oConfiguracaoConversorPOS = new CConfiguracaoConversorPOS();
  oPrintDeb = new CPrintDeb("conversorpos");
  oTemporizador = new CTemporizador(oFilaEventos, pHandle);
  oIOPDV = (VIoPDV *) NewObjeto("IoPdvTcp.dll");
  oInterfacePDV = (VInterface *) NewObjeto("InterfaceTCPIP.dll");
  oConexaoPDV = (VConexao *) NewObjeto("ConexaoTCP.dll");
  oConexaoPDV->setCodigoConexao(CONEXAOPDVTCP);
```

```
  oClientRedeVisa = (VClientRede *)NewObjeto("ClientVisa.dll");
  oClientRedeRedecard = (VClientRede *)NewObjeto("ClientRedecard.dll");
  oClientRedeGenerica = (VClientRede *)NewObjeto("ClientGenerica.dll");
  oClientRedePadrao = (VClientRede *)NewObjeto("ClientPadrao.dll");
  oClientRedeGetNet = (VClientRede *)NewObjeto("ClientGetNet.dll");
}
```

Agora os objetos são instanciados a cada transação criada, como mostrado em parte do código fonte abaixo da classe *CClientPOS*.

```
void CClientPOS::Inicializa(void)
{
  oTransacaoAndamento = new CTransacaoAndamentoPOS();
  oListParametroSolicitacao  = new TList();
  oListParametroSolicitacaoAdvice  = new TList();

  oListParametroConfirmacao  = new TList();
  oListIdentificadorSolicitacao  = new TList();
  oListParametroResposta  = new TList();

  oListTipoAbastecimento = new TList();
  oListValorLitro = new TList();
  oListValorServico = new TList();

  oClientRedeVisa = (VClientRede *)NewObjeto("ClientVisa.dll");
  oClientRedeRedecard = (VClientRede *)NewObjeto("ClientRedecard.dll");
  oClientRedeGenerica = (VClientRede *)NewObjeto("ClientGenerica.dll");
  oClientRedePadrao = (VClientRede *)NewObjeto("ClientPadrao.dll");
  oClientRedeGetNet = (VClientRede *)NewObjeto("ClientGetNet.dll");
}
```

5.3.8 Análise e Modificação da Classe *CEncriptaAes*

A classe *CEncriptAes* possui métodos para a criptografia de dados da aplicação.

O problema encontrado com a paralelização foi que essa classe possuía funções estáticas que eram utilizadas sem instanciá-la.

```
class CEncriptaAes
{
  private:
    DLLCriptografa CriptografaDll;
    DLLDesCriptografa DesCriptografaDll;
    DLLCriptografaBufferDados CriptografaBufferDadosDll;
    DLLDesCriptografaBufferDados DesCriptografaBufferDadosDll;

  public:
    bool bDLLCarregada;
    HINSTANCE_DLL hinstDLL;
    int iNumeroChamadas;
```

```
    CEncriptaAes();
    ~CEncriptaAes();

    int InicializaN();
    int FinalizaN();
    int CriptografaBufferN(int iTamBuffer, char *pBuffer, ...);
    int DesCriptografaBufferN(int iTamBufferCripografado, ...);
    static int Inicializa();
    static int Finaliza();
    static int CriptografaDadosNumericos(char *Chave, ...);
    static int DesCriptografaDadosNumericos(char *Chave, ...);
    static int CriptografaDadosAlfaNumericos(char *Chave,...);
    static int DesCriptografaDadosAlfaNumericos( ...);
    static int CriptografaCartao(int NroBit, char *Chave,...);
    static int DesCriptografaDadosBinarios(char *Chave, ...);
    static int CriptografaDadosBinarios(char *Chave, ...);
    static int CriptografaBuffer(int iTamBuffer, ...);
    static int DesCriptografaBuffer(int iTam, ...);
};
```

Um exemplo de como era a chamada dos métodos dessa classe pode ser visto abaixo em parte do código fonte.

```
int CClientPOS::GravaComprovante(AnsiString sNomeArquivo, AnsiString
sComprovante, bool bGravarUltimoComprovante, int iTipoTransacao)
{
...
  sChave="7466214";
  // montagem da chave de criptografia em duas partes
  sChave=sChave+AnsiString("983672106");
  ptrDescript = new char[(sComprovante.Length())+512];

  CEncriptaAes::CriptografaBuffer(sComprovante.Length(),
sComprovante.c_str(), sChave.c_str(),
  &iTamBuffer,ptrDescript);

  sComprovanteOriginal = AnsiString(ptrDescript,iTamBuffer);
  delete [] ptrDescript;
...
}
```

Como mostrado acima, a classe não foi instanciada e foi utilizado um dos métodos declarados como estático.

A modificação foi retirar os métodos estáticos declarados na classe e também instanciar um objeto da classe para o uso dos métodos, como podemos ver abaixo em parte do código fonte modificado.

```
int CClientPOS::GravaComprovante(AnsiString sNomeArquivo, AnsiString
sComprovante, bool bGravarUltimoComprovante, int iTipoTransacao)
{
```

```
.
.
.
  CEncriptaAes *oEncriptaAes;

  oEncriptaAes =new CEncriptaAes();
  oEncriptaAes->Inicializa();

  sChave="7466214";
  // montagem da chave de criptografia em duas partes
  sChave=sChave+AnsiString("983672106");
  ptrDescript = new char[(sComprovante.Length())+512];

  oEncriptaAes->oCriptografaBuffer(sComprovante.Length(),
sComprovante.c_str(), sChave.c_str(),
  &iTamBuffer, ptrDescript);

  sComprovanteOriginal = AnsiString(ptrDescript,iTamBuffer);
  oEncriptaAes->Finaliza();

  delete oEncriptaAes;
  delete [] ptrDescript;
...
}
```

5.3.9 Aplicativo para Testes do Paralelismo

Foi desenvolvido um aplicativo para testes de estresse que pode simular diversos *POS* enviando transações simultâneas. Ele servirá para criar uma base de dados para uma melhor visualização dos resultados obtidos com a paralelização.

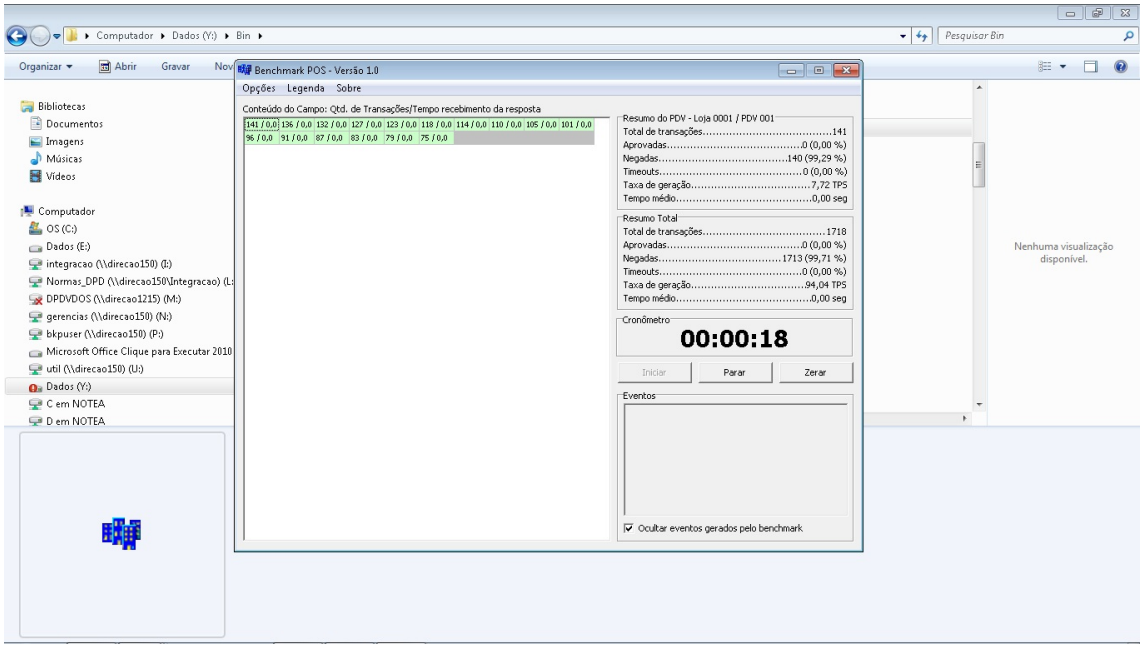

Figura 5.9: *Benchmark POS*

5.4 Análise da Implementação

Paralelizar essa aplicação com a biblioteca *pthreads* foi uma tarefa complicada, pois não há métodos que simplificam essa execução. Primeiramente, foi necessário analisar todo o código do programa para implementar as novas *threads*. Em seguida, foram analisadas as DLLs que eram instanciadas apenas uma vez e acessadas ao mesmo tempo pelas threads. Isso gerou os problemas das DLLs não reentrantes, conforme citado na seção 4.3. Foi adotado como solução a criação de várias instâncias das DLLs, uma para cada thread. Também foram analisados os pontos em que existiam dados compartilhados para essas threads e, utilizando mecanismos de sicronização como semáforos, evitou-se o acesso simultâneo dessas regiões críticas.

Mesmo com essas análises e modificações, após executar o aplicativo ocorreram *deadlocks* e, para encontrá-los, teve-se que depurar a aplicação. Um dos erros verificados com a depuração foi o acesso simultâneo dos métodos estáticos pelas threads, como mostrado na subseção 5.3.8. Também foram encontrados, após a depuração, os problemas com o uso dos componentes da VCL, conforme citado na subseção 5.3.3.

Após muitos testes, pôde-se sanar a maioria dos erros que ocorriam, mas esse método de tentativa e erro não é o mais adequado, já que não é eficiente quanto ao tempo e também não garante que a aplicação esteja corretamente paralelizada, pois seria necessário muito tempo para realizar todos os testes.

6 RESULTADOS OBTIDOS

6.1 Ambiente e Metodologia de Testes

Nos testes foram utilizados dois computadores pessoais da fabricante *Dell*, modelo *Optiplex 990*, sistema operacional *Windows 7 Professional*, processador *Intel(R) Core(TM) i5-2400*, *CPU* de 3.10 GHz e memória RAM de 4 GB. A partir da necessidade de os aplicativos comunicarem-se através de conexões *TCP*, os computadores foram ligados em uma rede local de 100 *Gigabits*. Para uma melhor compreensão, iremos referenciar os dois programas que foram utilizados nos testes como **monothread** e **multithread** que, respectivamente, possuem programação sequencial e paralela. Em um dos computadores foi instalado o aplicativo *benchmark* que simula diversos *POS* e no outro computador foram instalados os programas *monothread* e *multithread*. Configurando o aplicativo *benchmark* para 1, 10, 30, 60, 150, 300 e 600 *POS*, foi realizada uma bateria de testes individuais de 5 minutos com os aplicativos *monothread* e *multithread*. Utilizou-se o aplicativo Monitor de Recursos do próprio *Windows* para analisar o uso da *CPU* e dos outros recursos das máquinas. Mediu-se, também, a quantidade de transações por segundo em tomadas de tempo.

6.2 Testes e Resultados

Essa seção apresenta e compara os gráficos e figuras obtidos a partir das tomadas de tempo realizadas com os programas *monothread* e *multithread*.

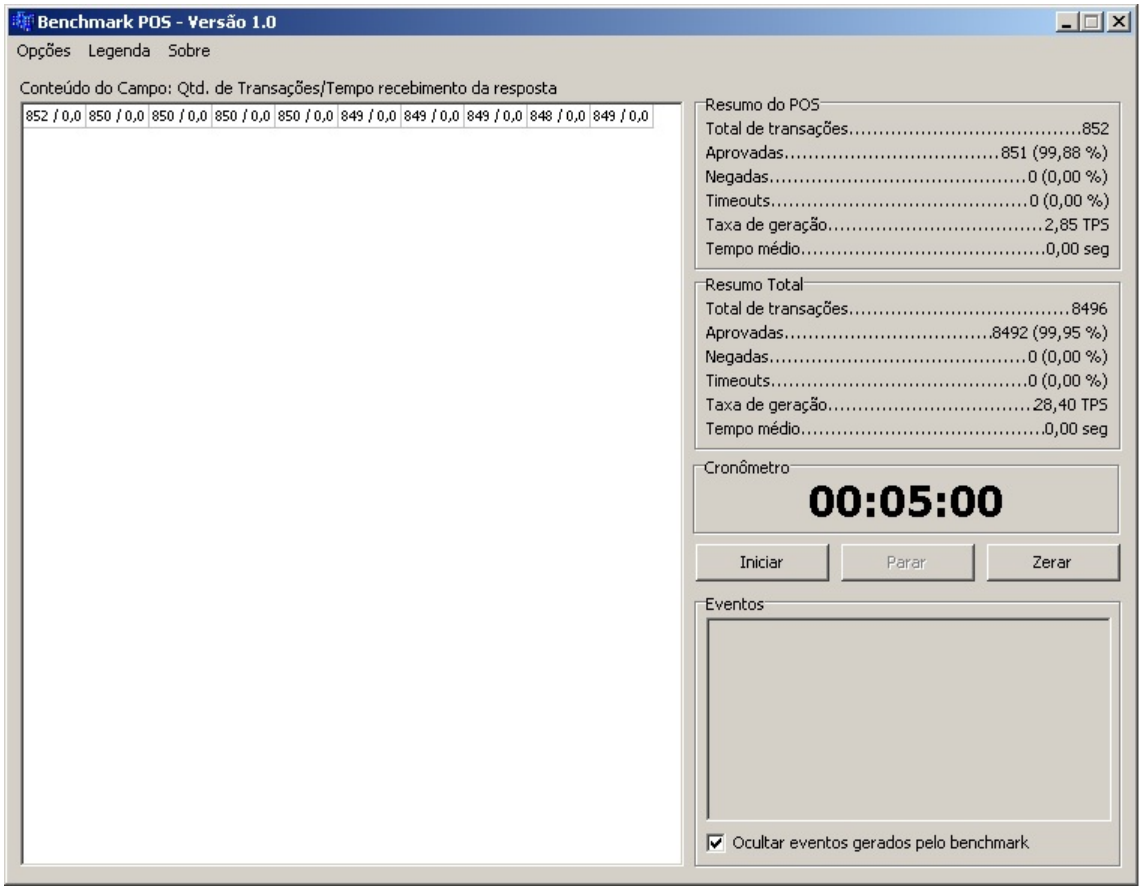

Figura 6.1: *Benchmark* Monothread *10 POS*

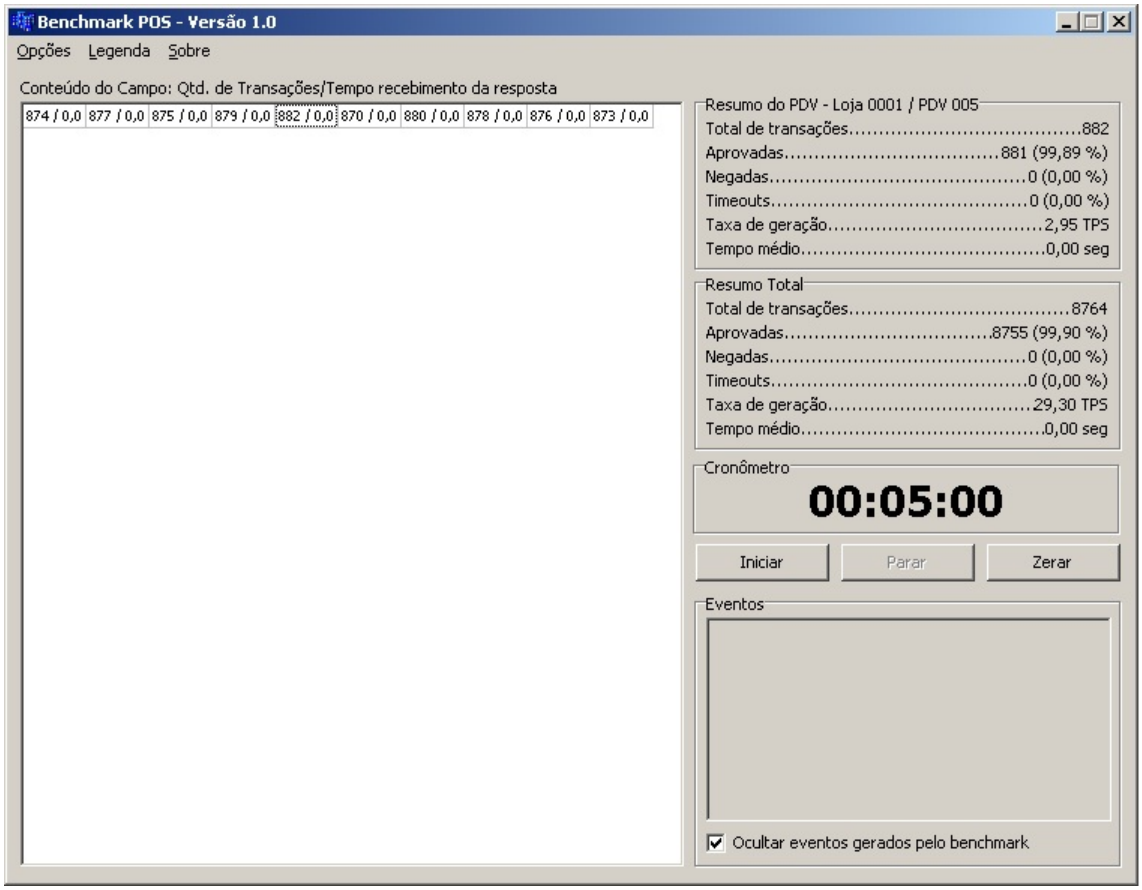

Figura 6.2: *Benchmark* Multithread *10 POS*

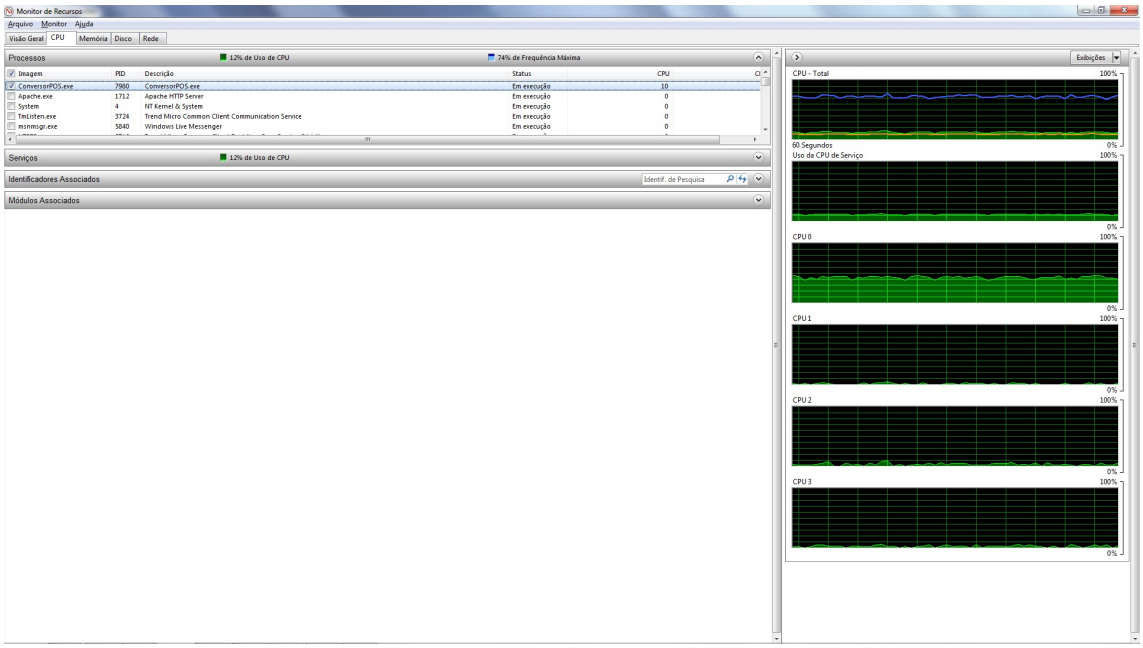

Figura 6.3: *CPU* - Aplicativo *Monothread* recebendo transações de 10 *POS*

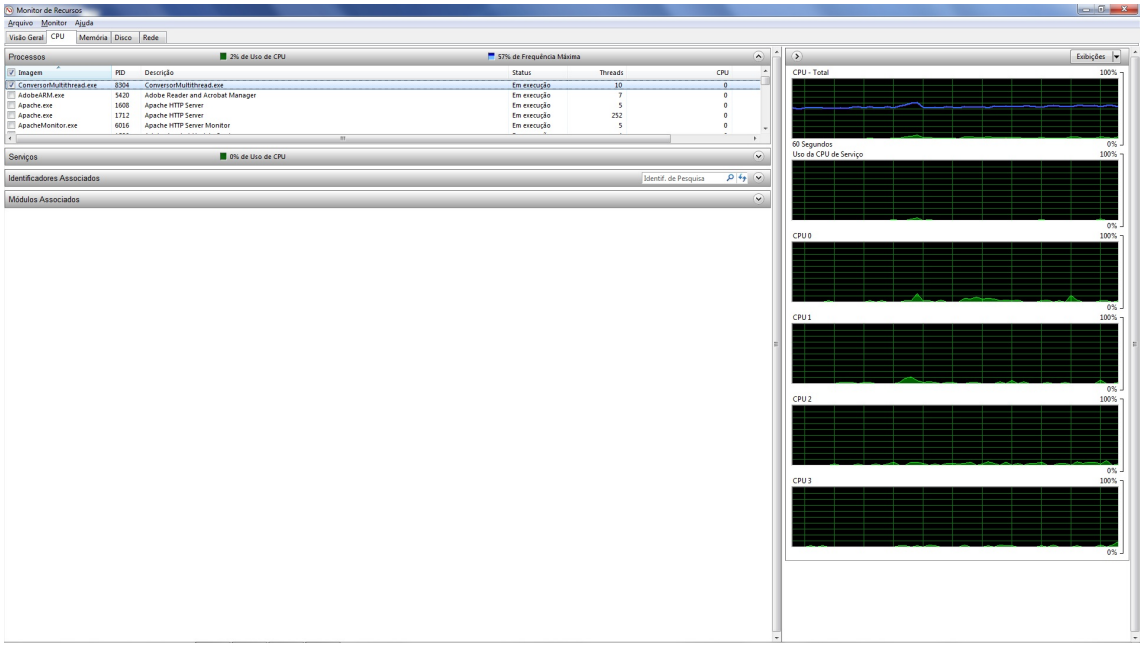

Figura 6.4: *CPU* - Aplicativo *Multithread* recebendo transações de 10 *POS*

Os testes realizados simulando 10 *POS* tiveram resultados parecidos em termos de cálculo de *throughput* entre os aplicativos *monothread* e *multithread*. Analisando os gráficos de uso da *CPU* percebe-se que nenhuma das aplicações utilizou toda a *CPU* disponível. Assim, pode-se concluir que quando não temos 100% do uso da *CPU* pelo aplicativo *monothread* o ganho de performance do aplicativo *multithread* é pequeno em relação ao *monothread*, como pode ser visto nas figuras 6.1 e 6.2.

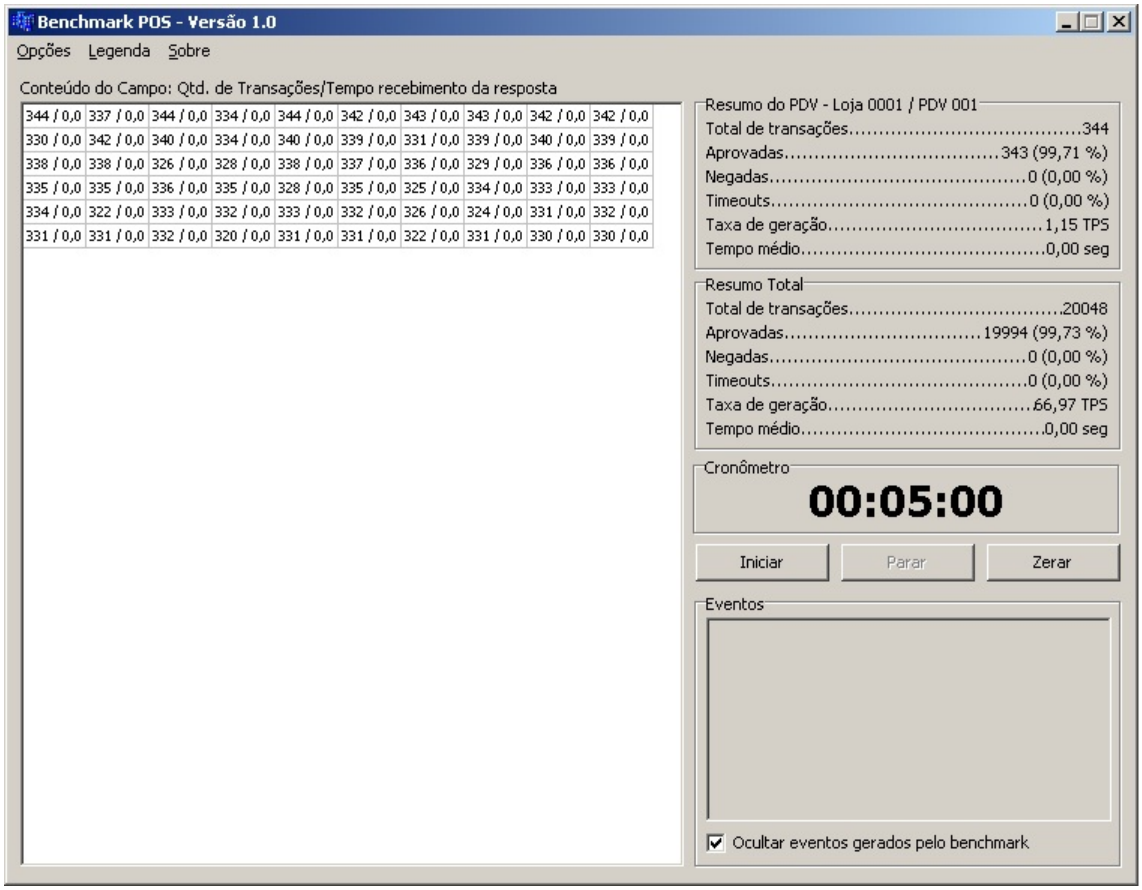

Figura 6.5: *Benchmark* Monothread *60 POS*

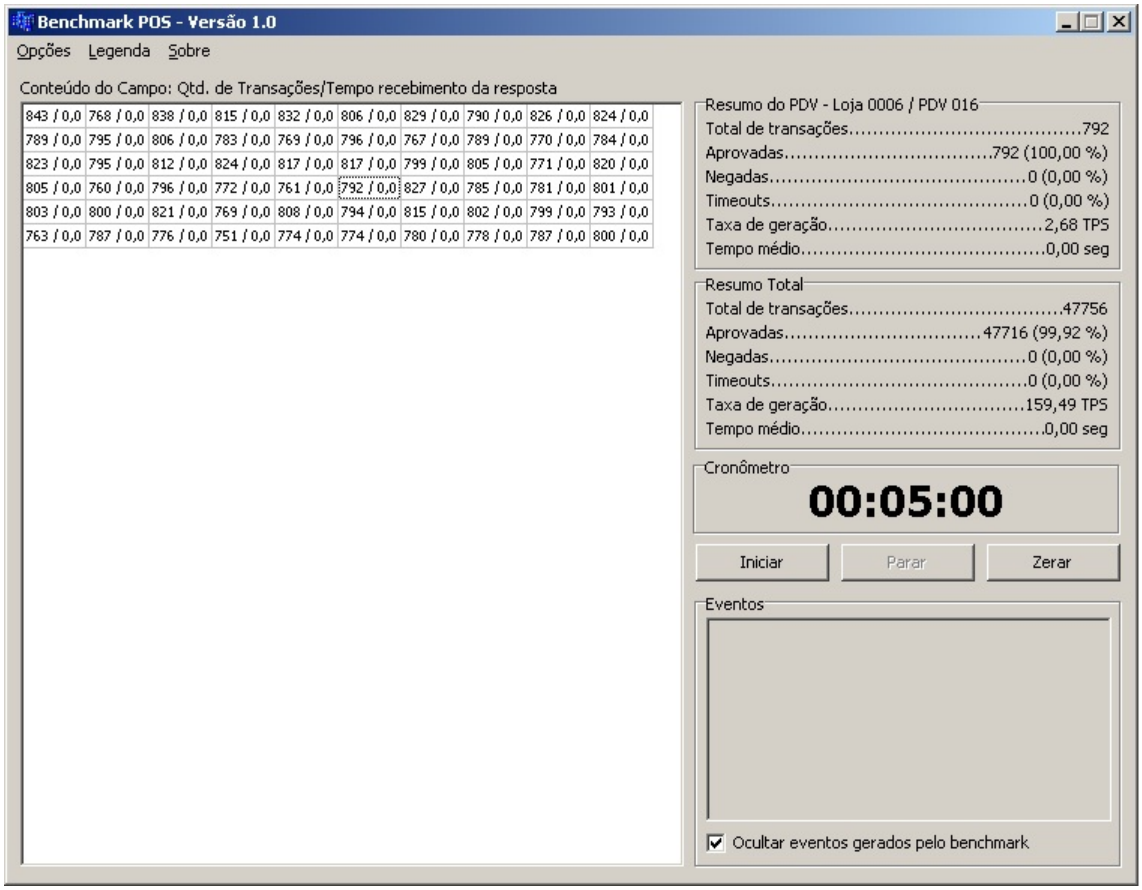

Figura 6.6: *Benchmark* Multithread *60 POS*

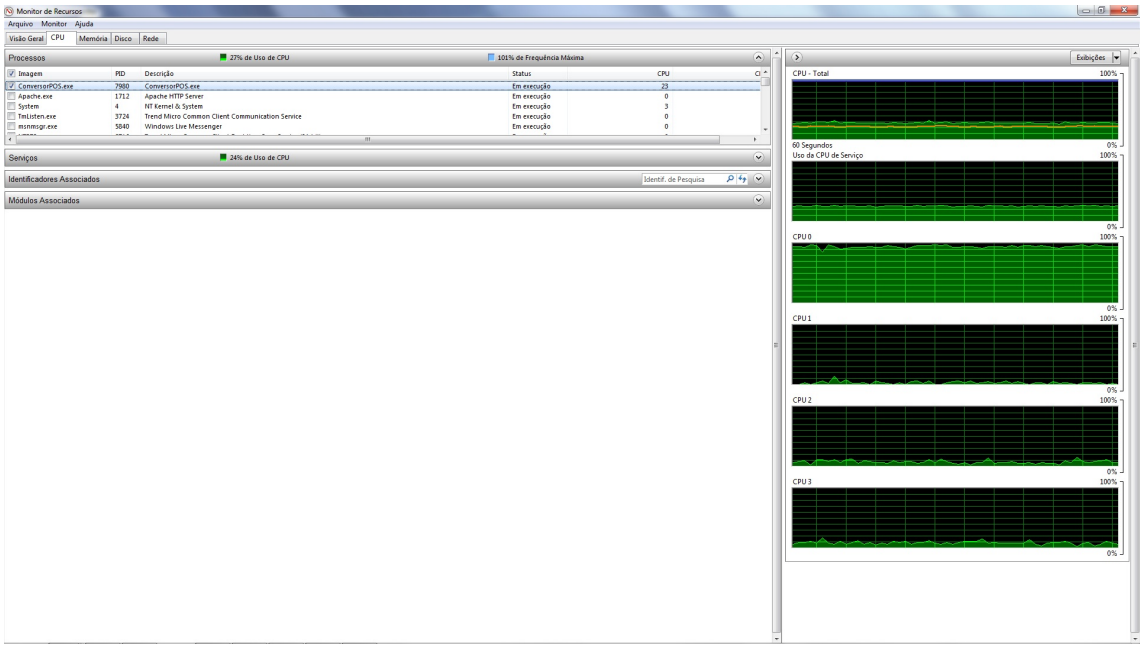

Figura 6.7: *CPU* - Aplicativo *Monothread* recebendo transações de 60 *POS*

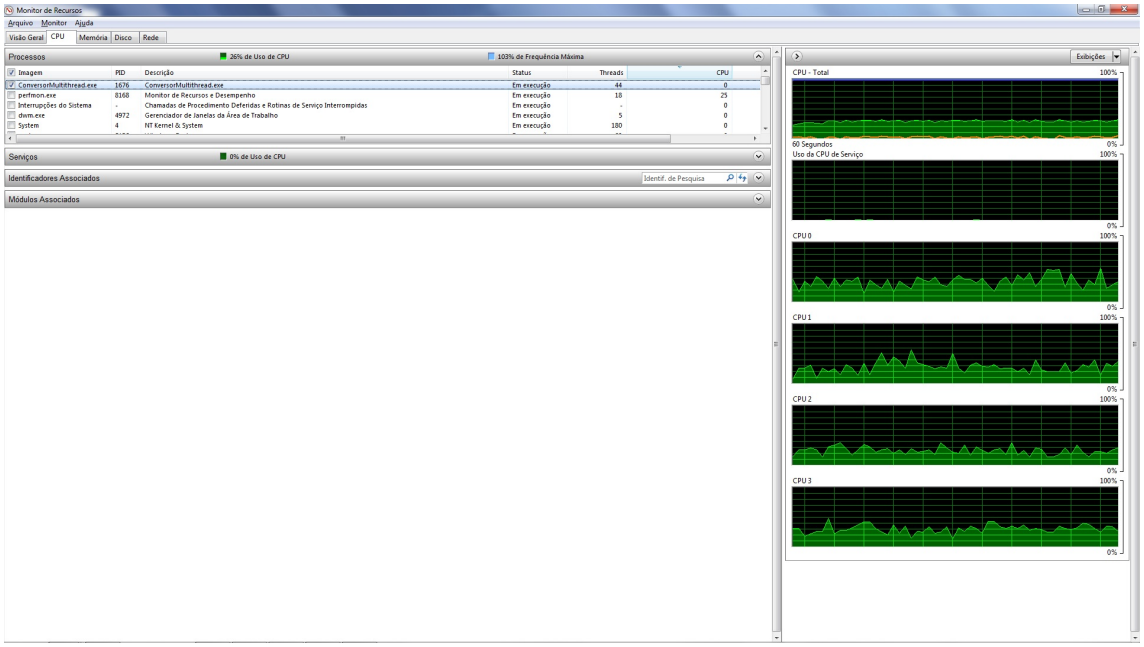

Figura 6.8: *CPU* - Aplicativo *Multithread* recebendo transações de 60 *POS*

Os testes realizados com 60 *POS* tiveram resultados bem distintos. O aplicativo *monothread* teve um desempenho de 66,97 TPS e o *multithread* de 159,49 TPS. Nesse teste consegue-se ter uma visualização melhor do desempenho da programação paralela, que utiliza todas as *CPUs* disponíveis para a obtenção de uma melhor performance, conforme pode ser visto na figura 6.8. Visualiza-se, também, que a *CPU* utilizada pelo programa *monothread* está em 100% do uso e pode-se verificar nos próximos testes que o desempenho dessa aplicação não terá uma melhora.

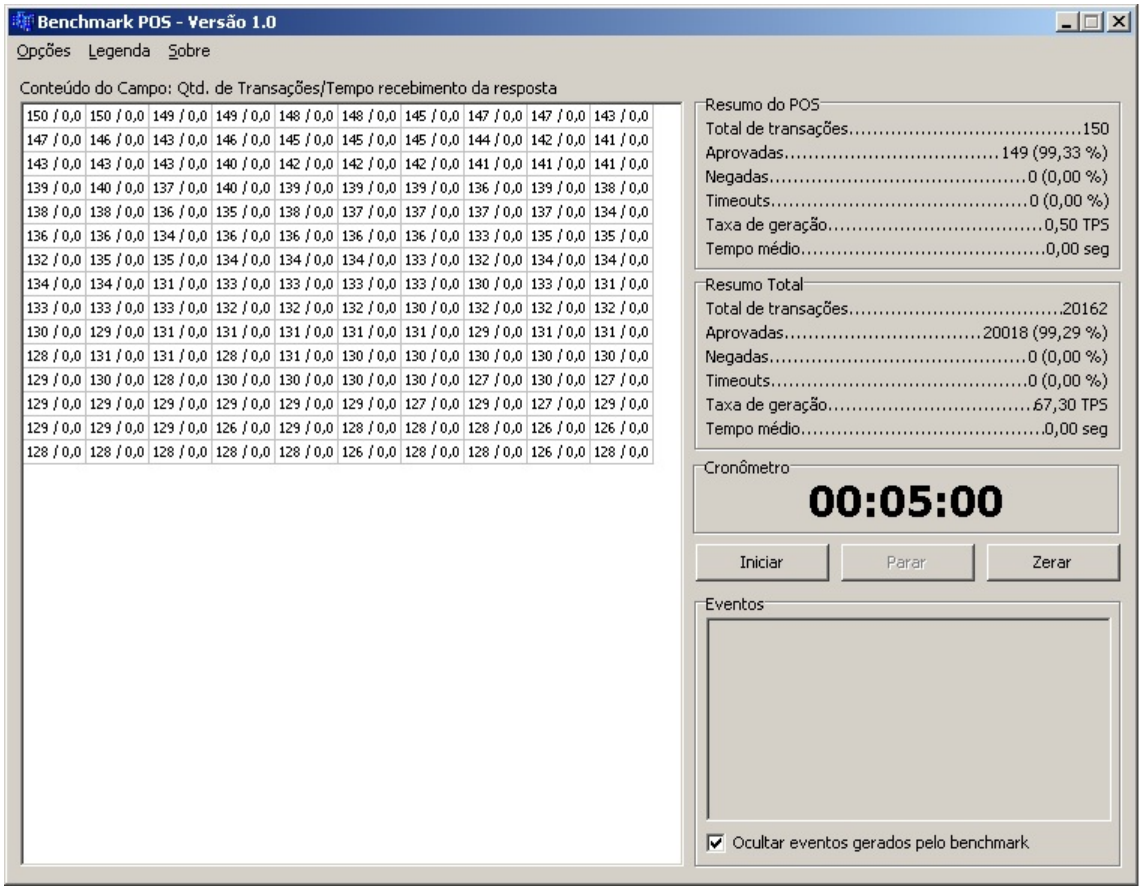

Figura 6.9: *Benchmark* Monothread *150 POS*

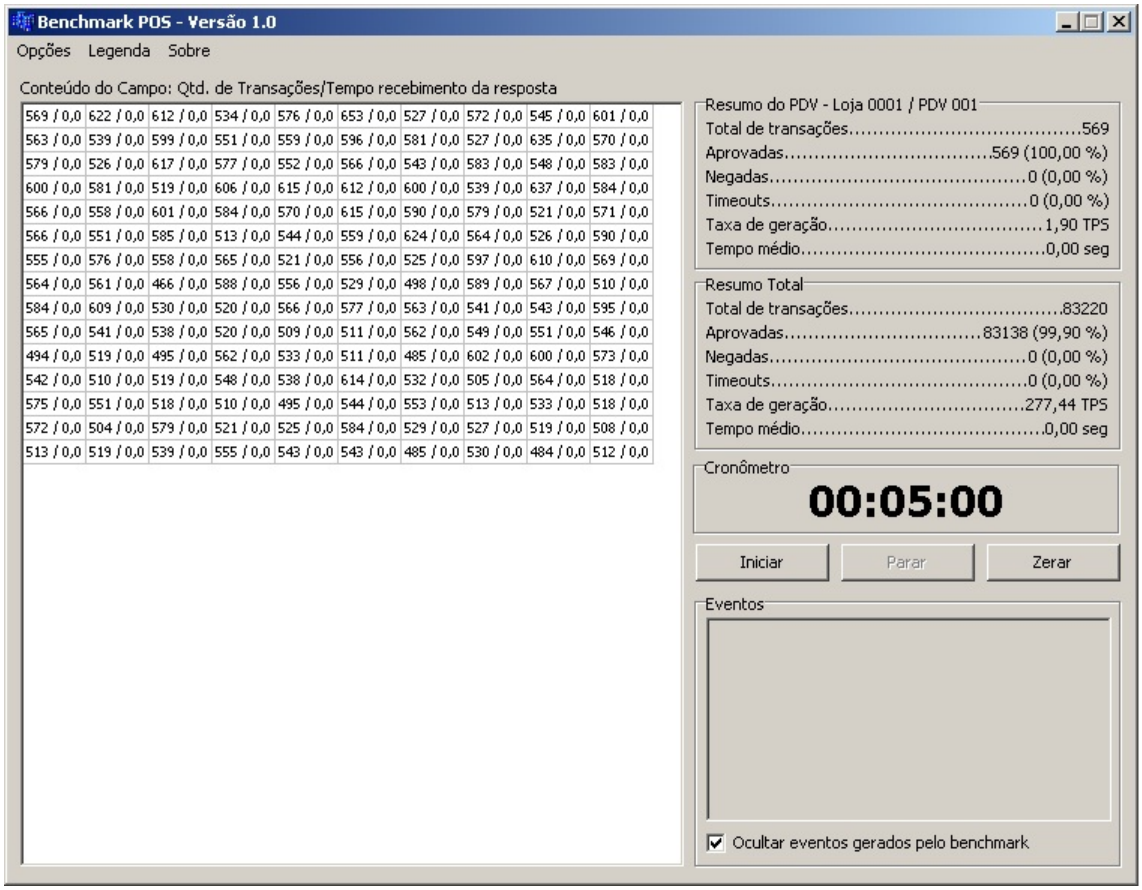

Figura 6.10: *Benchmark* Multithread *150 POS*

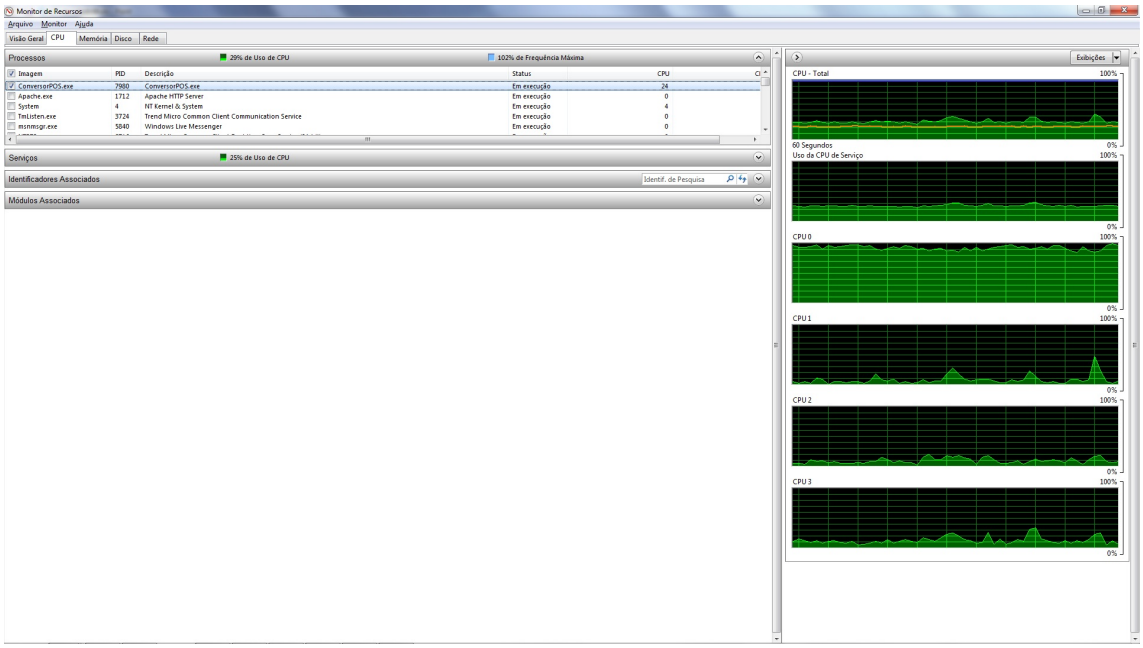

Figura 6.11: *CPU* - Aplicativo *Monothread* recebendo transações de 150 *POS*

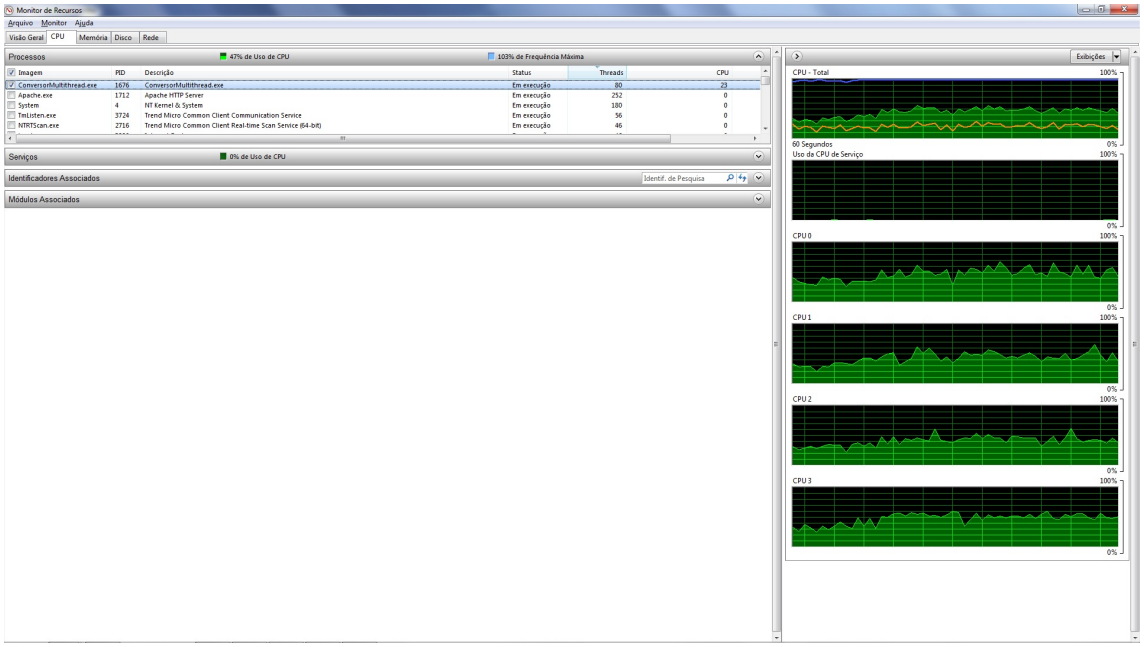

Figura 6.12: *CPU* - Aplicativo *Multithread* recebendo transações de 150 *POS*

Nos testes realizados com carga de 150 *POS*, o aplicativo *multithread* obteve uma melhora no seu desempenho, aumentando o número de transações por segundo para 277,44. Já no aplicativo *monothread* o desempenho ficou estagnado. Verifica-se essa análise nas figuras 6.9 e 6.10. Com relação ao uso das *CPUs*, pode-se verificar nas figuras 6.11 e 6.12 que o aplicativo *monothread* continua com uso de 100% da *CPU* e o aplicativo *multithread* distribui o trabalho para as *CPUs* disponíveis, utilizando apenas 25%. Como houve uma estagnação para o aplicativo *monothread*, não é necessário apresentar outros testes para ele, pois o desempenho e o uso da *CPU* continuarão os mesmos. Para o aplicativo *multithread*, a carga de transações simultâneas foi aumentada a fim de visualizar a capacidade máxima da programação paralela, já que ainda havia um bom percentual livre do uso das *CPUs*.

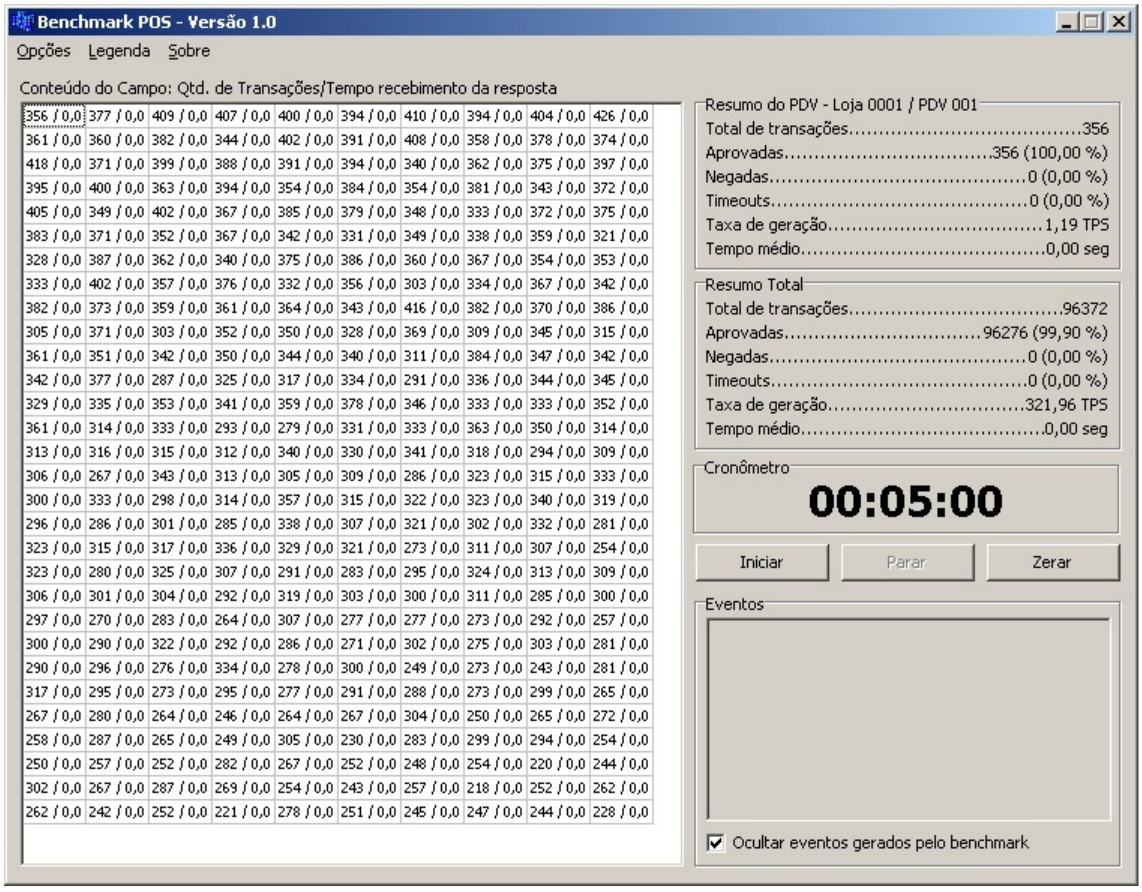

Figura 6.13: *Benchmark* Multithread *300 POS*

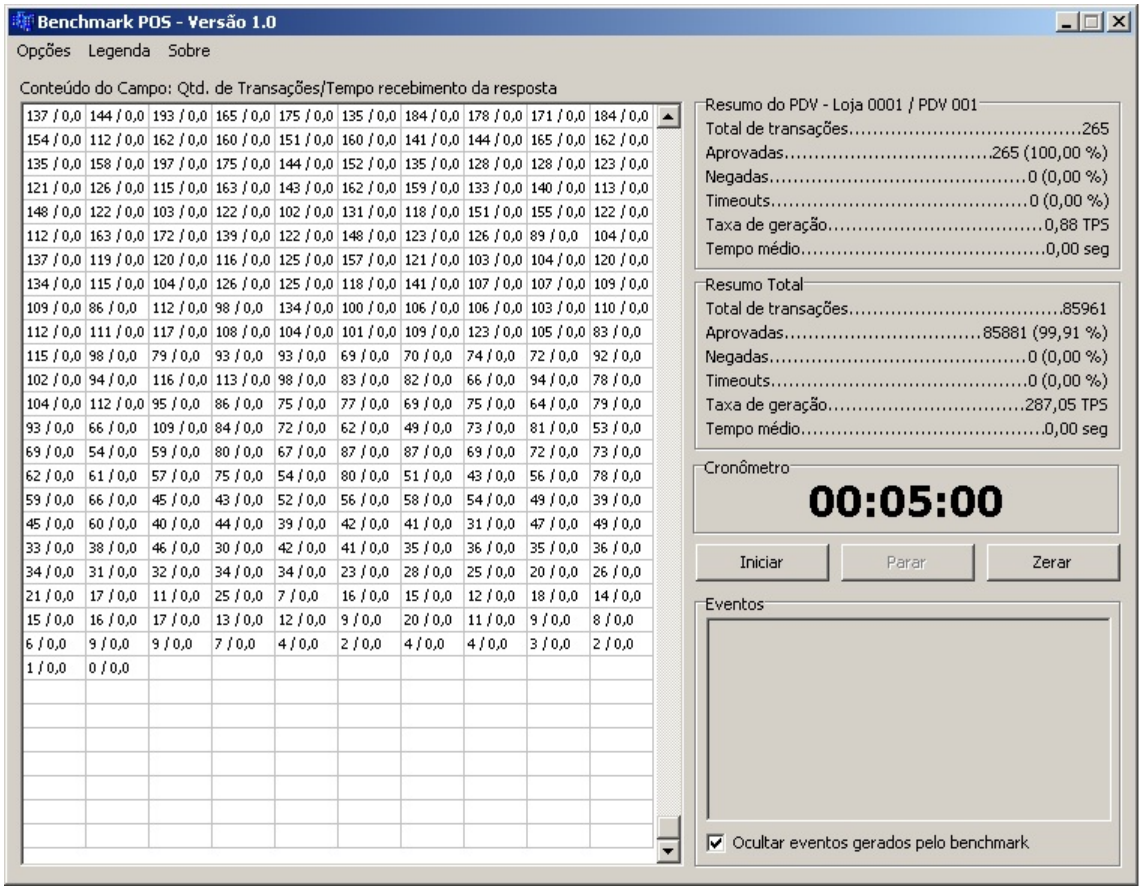

Figura 6.14: *Benchmark* Multithread *600 POS*

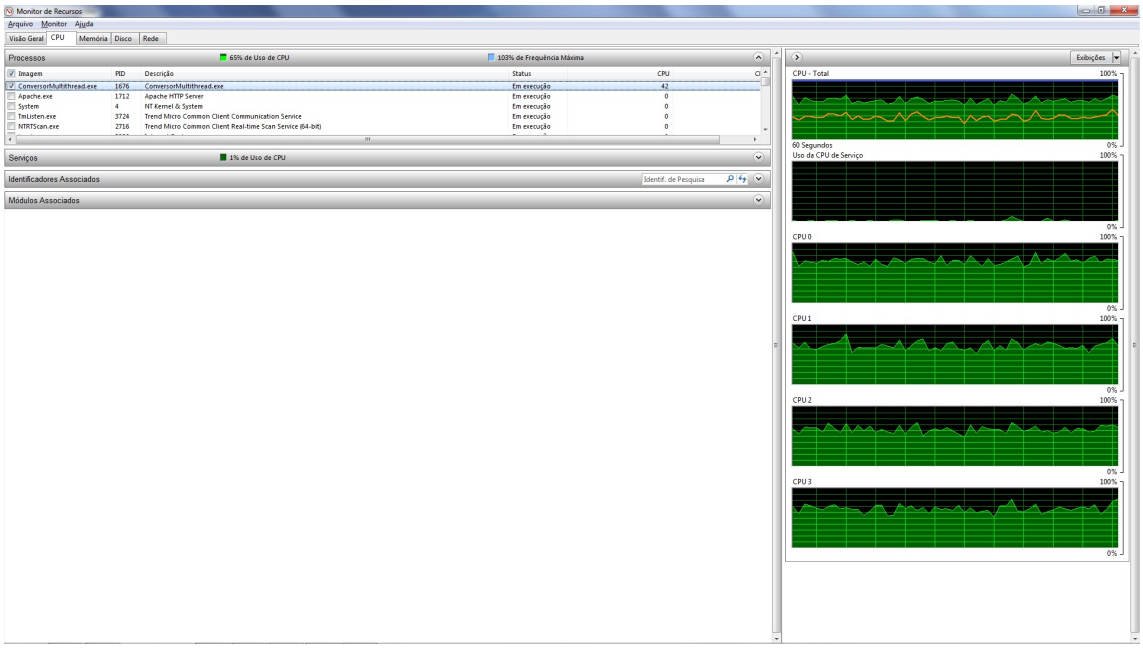

Figura 6.15: *CPU* - Aplicativo *Multithread* recebendo transações de 300 *POS*

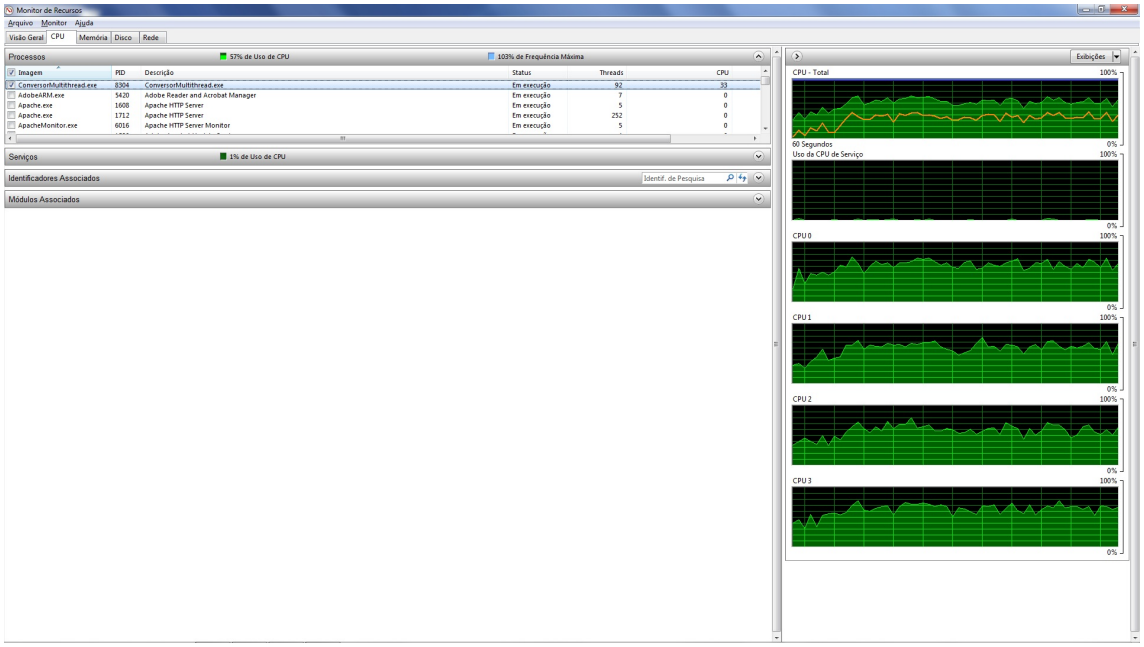

Figura 6.16: *CPU* - Aplicativo *Multithread* recebendo transações de 600 *POS*

Os testes com carga de 300 e 600 *POS* com o aplicativo *multithread* geraram alguns resultados inesperados. O desempenho do *benchmark* com 300 *POS* foi de 321,96 TPS, e com 600 *POS* foi de 287,05 TPS. Analisando o gráfico de uso das *CPUs* verifica-se que o teste com 300 *POS* teve um maior uso da *CPU* do que o teste com 600 *POS*. Sendo assim, foi necessário descobrir o motivo para que isso tenha ocorrido. Analisando os gráficos de memória, *HD* e rede verificou-se um desempenho normal nos dois testes, ou seja, sem chegar a 100% do uso desses recursos. O único gráfico que estava anormal era o de conexões *TCP*, em que aparecia de maneira intermitente 100% e 0% do uso, como pode ser visto na figura 6.17. Após essa análise, verificamos se o problema estaria na aplicação *benchmark* que gera as transações, considerando que essa aplicação fosse duplicada em outra máquina. Após essas modificações, verificou-se uma melhora no desempenho da aplicação *multithread* que recebia transações de 600 *POS*, enviados por duas máquinas. O desempenho dos *benchmarks* foi de 202 TPS e 218 TPS, totalizando um desempenho de 420 TPS, conforme as figuras 6.18 e 6.19. Um maior uso das *CPUs*, conforme esperado, também ocorreu, o que pode ser verificado na figura 6.20.

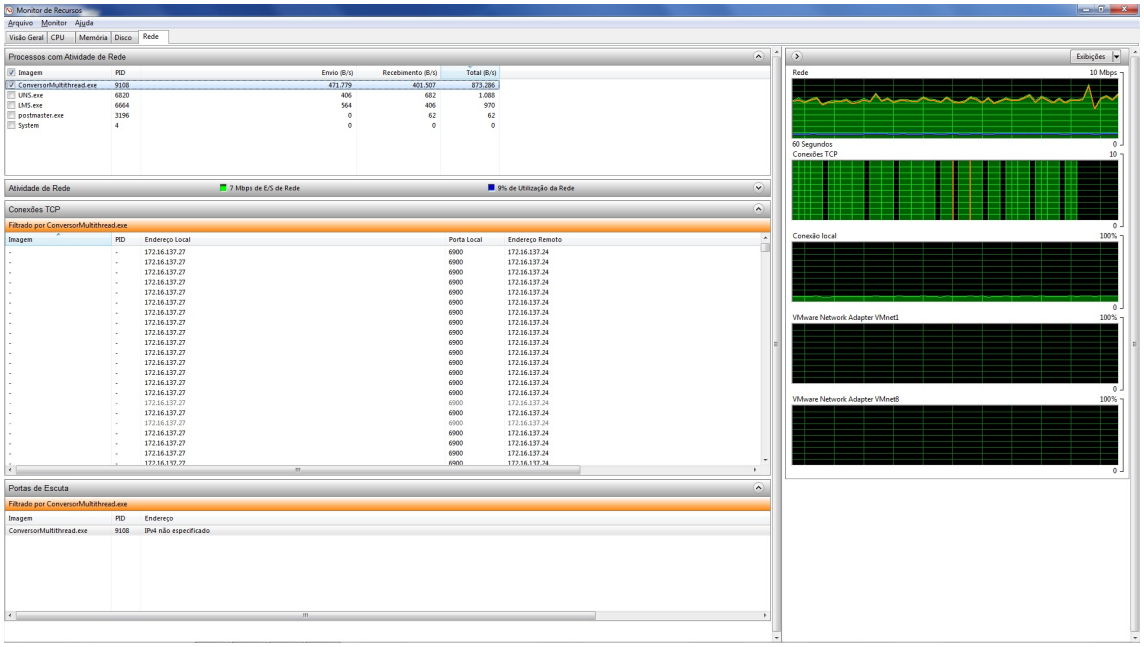

Figura 6.17: Conexões *TCP* - Aplicativo *Multithread* recebendo transações de 600 *POS*

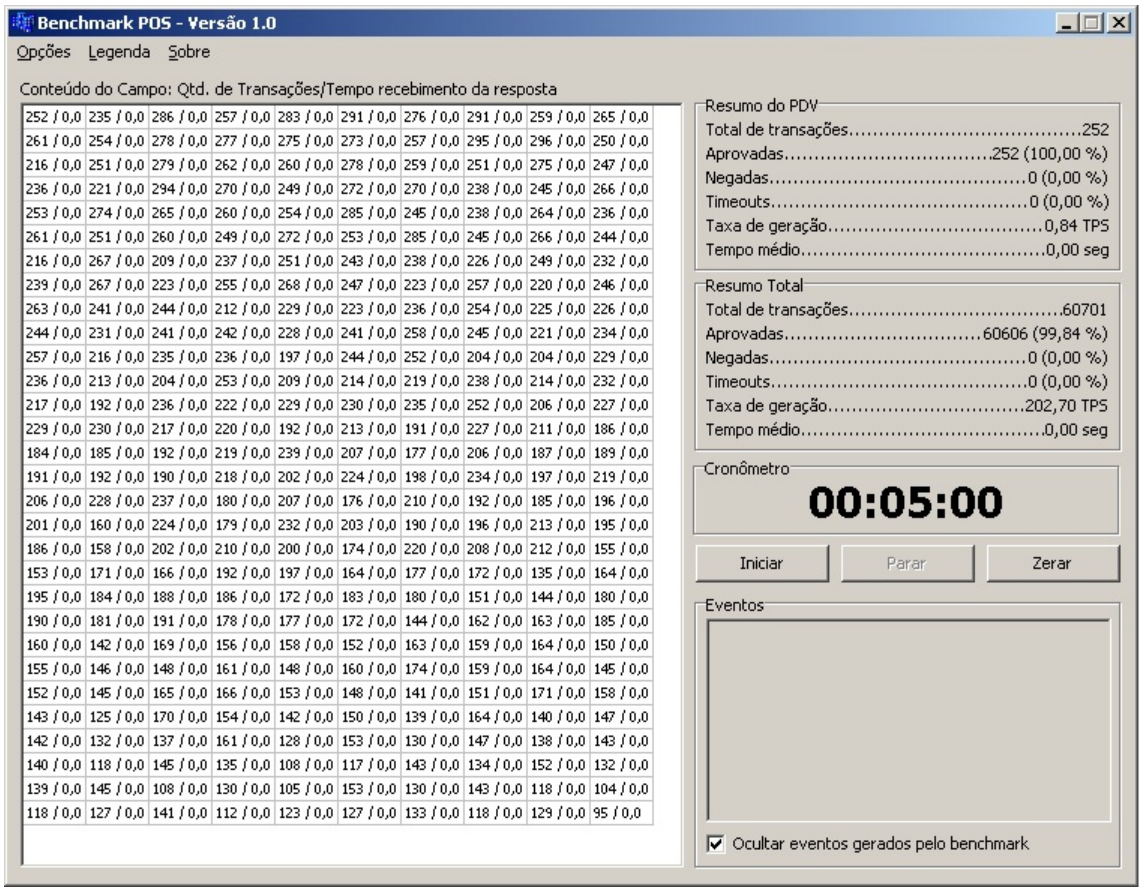

Figura 6.18: *Benchmark* Multithread *300 POS* máquina A

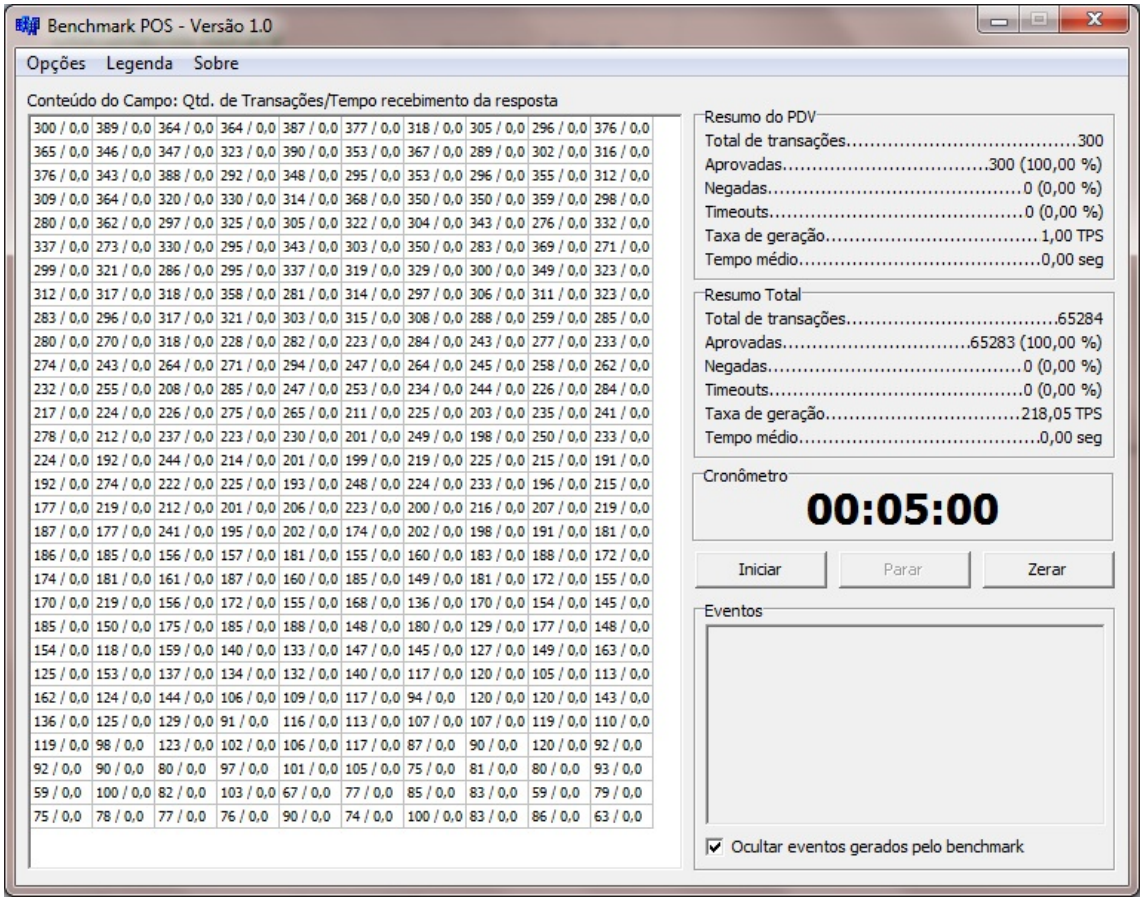

Figura 6.19: *Benchmark* Multithread *300 POS* máquina B

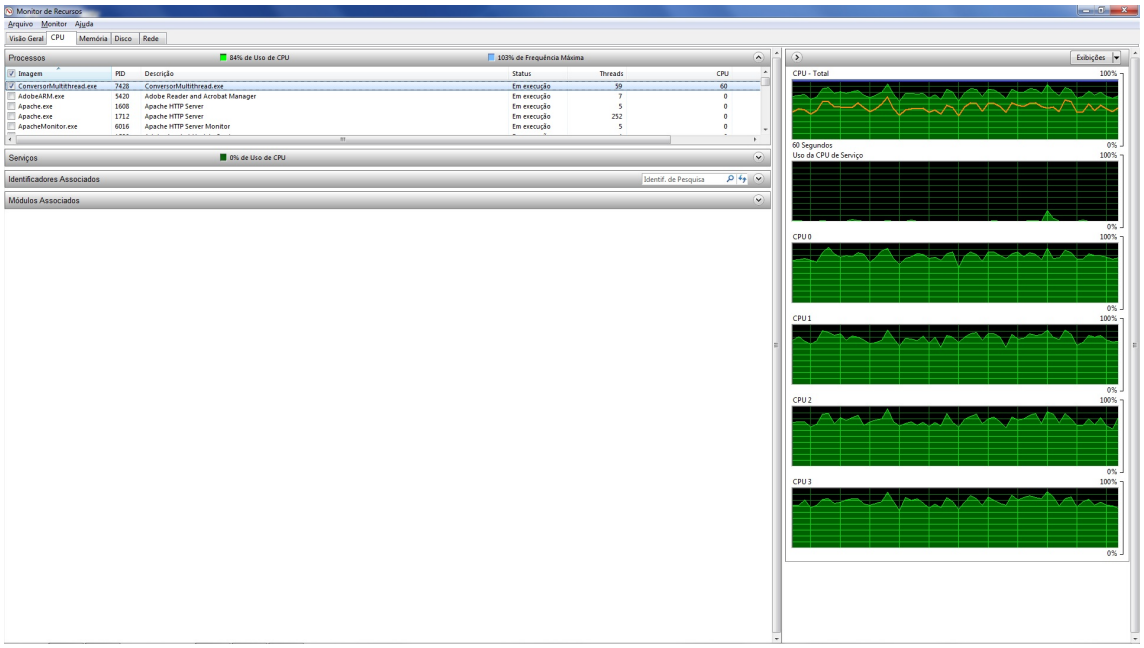

Figura 6.20: *CPU* - Aplicativo *Multithread* recebendo transações de 600 *POS*

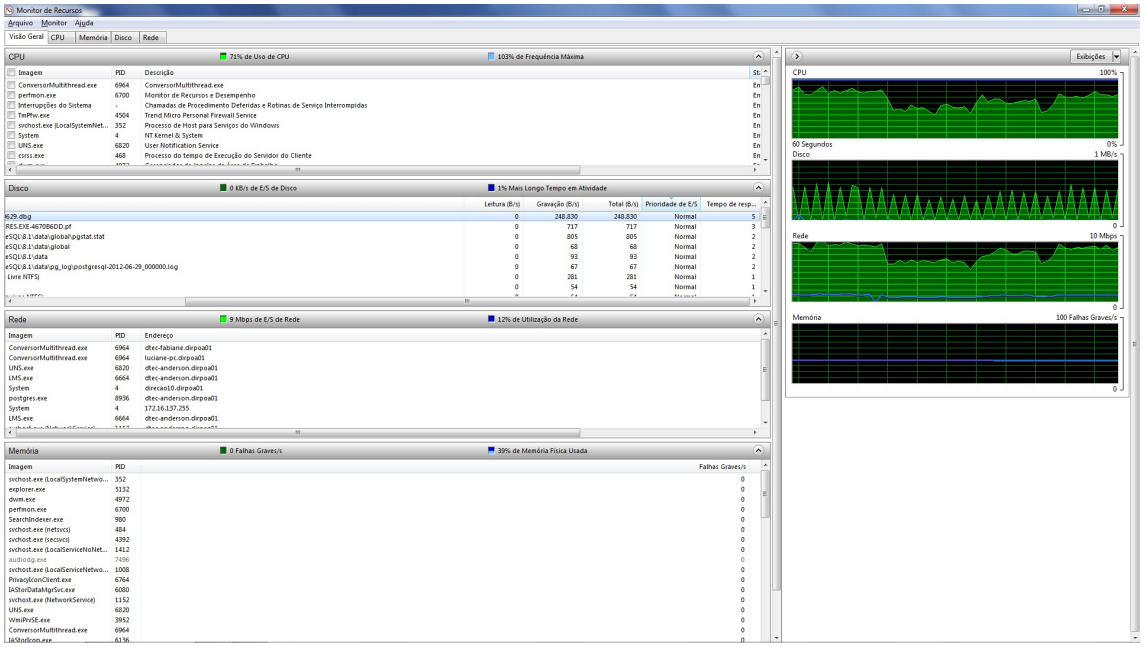

Figura 6.21: Análise geral - *CPU*, *HD*, Memória e Rede

6.3 Conclusão sobre o capítulo

Os *benchmarks* apresentados avaliaram todo o ambiente de testes das aplicações programadas de maneira sequencial e paralela. O gráfico da figura 6.23 e a tabela 6.22 comparam os ciclos de testes realizados variando de 1 a 800 o número de POS, que enviam as transações de forma simultânea.

	1 POS	10 POS	30 POS	60 POS	150 POS	300 POS	600 POS	800 POS
Mono	2,86	28,4	67,35	66,97	67,3	67,33	71,5	68,35
Multi	2,97	29,3	84,44	159,49	277,44	321,96	420,75	430,23

Figura 6.22: Tabela Comparativo de Desmpenho

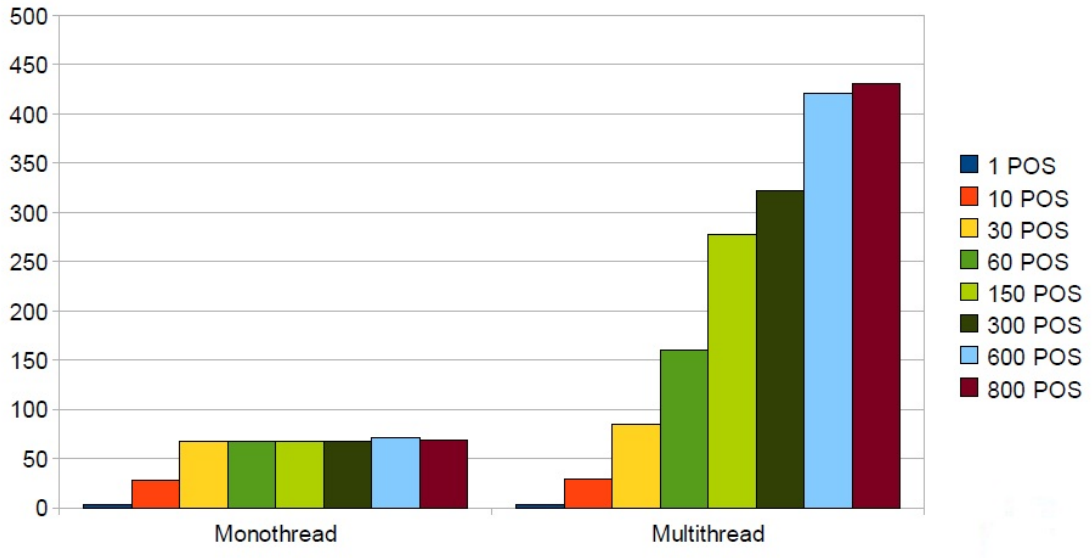

Figura 6.23: Comparativo de Desmpenho

Nos ciclos de testes com até 30 POS não percebe-se um melhor desempenho da aplicação *multithreaded*. Entretanto, nos ciclos com mais de 30 POS, ou seja, que geram uma carga de dados superior e uma exigência maior das CPUs, percebe-se uma melhora significativa do aplicativo paralelizado quando comparado ao sequencial.

7 CONSIDERAÇÕES FINAIS

Concluída a análise dos dados obtidos e apresentados os resultados alcançados a partir dos testes dos programas sequencial e paralelo, pode-se afirmar que, mesmo com todas as dificuldades encontradas em paralelizar um programa a partir do seu código sequencial, encontramos os resultados esperados, pois a programação paralela aumentou a eficiência do aplicativo estudado executado em máquinas *multicore*, já que um maior número de tarefas são processadas por unidade de tempo e, assim, os tempos de espera e de processamento de cada tarefa são reduzidos. Na aplicação desse trabalho foi obtido um ganho de até 5 vezes na capacidade de processamento de transações em uma única máquina, pelo uso da arquitetura multicore já disponível, com programação paralela.

Entretanto, salienta-se que para os testes do *benchmark* que possuíam um número menor de *POS* o desempenho não foi muito diferente, portanto a programação paralela diferenciou-se da sequencial nessa aplicação apenas para carga de trabalho que necessita de um uso maior da *CPU*.

Finalizando, deve-se ressaltar que transformar um programa sequencial em um paralelo não é uma tarefa muito fácil, devido a dificuldades como as descritas nesse trabalho. Por isso, os novos profissionais da área devem programar, desde o princípio, de forma a utilizar todos os recursos de *hardware* atuais, isto é, utilizando recursos da programação paralela, para que num futuro não seja necessário passar pelo trabalho complicado de paralelizar um programa inicialmente sequencial.

REFERÊNCIAS

BRINKHUS, R. *Algoritmo Genético Paralelo: avaliação de diferentes abordagens na solução de um problema inverso em vibrações*. 2009. Salão de Iniciação Científica - Instituto de Informática, Universidade Federal do Rio Grande do Sul, porto Alegre. 2009.

CARISSIMI, A. S. ; TOSCANI, S. S. ; OLIVEIRA, R. S. *Sistemas Operacionais*. 2a. ed. Porto Alegre : Sagra Luzzato, 2004.

Cygwin. *Cygwin Project*. Disponível em: <http://cygwin.com>. Acessado em junho de 2012.

GHEZZI, Carlo; Jazayeri, Mebdi. *Conceitos de linguagens de programação*. - Rio de Janeiro: Campus, 1991.

LIMA, J. *Controle de Granularidade com threads em Programas MPI Dinâmicos*. 2009. 65 f. Dissertação(Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre. 2009.

MORALES, D. *Compilação de Código C/MPI para C/PThreads*. 2009. 61 f. Monografia(Graduação em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre. 2009.

PILLA, L. *Análise de perfis paralelos em processadores gráficos*. 2009. Salão de Iniciação Científica - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre. 2009.

Pthreads Win32. *Open Source POSIX Threads for Win32*. Disponível em: <http://sourceware.org/pthreads-win32/>. Acessado em junho de 2012.

SEBESTA, Robert W. *Conceitos de Linguagens de Programação*. - 4. ed. - Porto Alegre: Bookman, 2000.

TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 2a. ed. Pearson, 2003.