Avaliação de Desempenho de Sistemas Paralelos

Genaro Costa

Abstract

The use of parallel computing is generally associated with high performance. In such direction the technics used to performance evaluation help users in hypothesis formulation about system behavior and experimentation design to test such hypothesis. In this text we present a small introduction to parallel programming aspects, a systematic performance evaluation process and monitoring technics. We present also a set of tools available for performance monitoring and analysis.

Resumo

O uso da computação paralela está associada frequentemente com alto desempenho. Nesse sentido as técnicas de avaliação de desempenho ajudam a estruturar a formulação de hipóteses sobre o comportamento do sistema, assim como, desenhar experimentos para testa-las. Nesse minicurso apresentamos uma breve introdução sobre aspectos da programação paralela, detalhando um processo sistemático de avaliação de desempenho, assim como técnicas de monitoração. Apresentamos também uma lista ferramentas disponíveis a ser usadas para monitoração e analise de desempenho.

1.1. Introdução

Quando partimos para o uso da computação paralela, estamos buscando, geralmente, executar um programa de forma mais rápida ou um programa que não 'cabe' em um sistema tradicional. Quando buscamos executar em menos tempo, fracionamos o problema codificado em partes que possam ser distribuídos entre diverso processadores buscando diminuir o tempo de total de processamento. A questão é que, nem sempre, a relação entre uso de diversos processadores em uma máquina paralela e o ganho na redução do tempo de execução é direta. Diversas questões podem influenciar no aproveitamento do poder de computo de uma um sistema paralelo (Wilkinson & Allen, 2004).

Considere um problema a ser programado para uso de um sistema paralelo como exemplo. Como resultado Esse programa resultante consiste de uma quantidade de operações a serem realizadas para resolução do problema. As técnicas de programação paralela tomam em conta o sistema paralelo disponível e fraciona a carga de trabalho entre os processadores de forma justa visando não deixar nenhum processador ocioso e aproveitar toda a capacidade de processamento. Com isso esperamos obter um menor tempo de execução do nosso programa, ou seja, um melhor desempenho desse programa.

Questões como: tamanho do bloco de trabalho (que chamaremos de granularidade), quantidade de processos, uso de primitivas de comunicação síncrona ou assíncrona, associação entre processos e processadores, escolha de processadores e escolha do paradigma de comunicação impactam diretamente no desempenho de um programa executando em um sistema paralelo. Em um processo experimental tal questões devem ser documentadas e aquelas que suspeitamos ter impacto no desempenho devem ser exploradas através de uma serie de experiemtos.

Através do processo de experimentação podemos identificar quais aspectos influenciam no desempenho assim como escolher qual a melhor configuração para nossa aplicação paralela considerando seu desempenho (Jain, 1991). Esse minicurso trata de uma pequena introdução nas questões que influenciam no desempenho de uma aplicação paralela, seguido de uma base sistemática para construção de experimentos e apresenta técnicas e ferramentas disponíveis para uso no processo.

1.1.1. Limites Teóricos

Na criação do programa paralelo, a qual chamaremos de paralelização de um problema, esperamos uma redução no tempo de execução em relação a sua versão não paralela. A primeira questão que se postula é quanto se pode conseguir usando um conjunto de processadores trabalhando em conjunto. É intuitivo associar a quantidade de processadores á redução do tempo de execução. Esperamos, por exemplo, que com quatro processadores tenhamos uma execução quatro vezes mais rápida. Isso nem sempre acontece, devido às questões que veremos adiante. Primeiro devemos entender como comparar diferentes execuções (Lilja, 2005). O índice mais usado para comparar diferentes execuções é o *speedup*, explicado em sequencia.

1.1.2. Speedup

Quando comparamos desempenho entre dois sistemas, ou duas versões de um mesmo sistema, usamos a relação entre os tempos de execução medidos, geralmente entre antes e depois de uma modificação do mesmo visando melhoria de desempenho. Tomamos como base a comparação entre duas versões de um mesmo sistema.

Considere como *speedup* a relação entre desempenho (poder de computo) entre as duas versões de um sistema. Geralmente usamos esse termo da seguinte forma, por exemplo: "Com uso de buffers temos um *speedup* de 1.2". O *speedup* é adimensional (por ser uma relação de duas grandezas de mesma dimensão) e se expressa pela formula abaixo:

$$S = \frac{T_1}{T_2}$$

Equação 1.1. A equação do Speedup.

No caso da equação acima, temos que o *speedup* da versão 2 em relação a versão 1, representadas pela amostragem do seu tempo de execução em T_1 e T_2.

Um outro uso comum do *speedup* é representar qual o ganho que se tem em usar um certo numero de processadores. Nesse caso comparamos o tempo de execução, usando um numero de processadores, com a execução da versão não paralela (versão com execução em série). Nessa situação temos a equação abaixo onde *Sp* é dito como o *speedup S* com p processadores, expresso como a relação entre o tempo de execução

usando um sistema paralelo de p processadores e o tempo de execução da versão do programa serial.

$$S_p = \frac{T_{paralelo}}{T_{serial}}$$

Equação 1.2. A equação do Speedup.

Quando usamos 5 processadores, por exemplo, esperamos um *speedup* $S_5 = 5$. Isso nem sempre acontece devido diversos motivos. Um estudo básico consiste em analisar qual o máximo ganho que se pode obter com o uso do paralelismo em um problema. Isso pode ser representado pela lei de Amdahl que vemos adiante. Quando o *speedup* é igual ao numero de processadores usados, dizemos que o *speedup* é linear.

1.1.2. Lei de Amdahl

Consiste da analise da estrutura do programa, correlacionando entre partes que podem ser paralelizadas com as que devem ser executada em série. Esse estudo foi feito pelo arquiteto de computadores Gene Amdahl e ajuda a encontrar qual o máximo *speedup* uma aplicação paralela pode alcançar.

$$S_p = \frac{1}{(1-P) + \frac{P}{N}}$$

Equação 1.3. A equação do *Speedup*.

A equação acima mostra que, considerando o tempo de execução em série como 1, o número de processadores como N, e o percentual do programa que pode ser paralelizado como P, o *speedup* consiste da relação entre a soma da parte do programa que não é paralelizável (1 – P) e a redução da parte paralelizável, considerando o *speedup* linear para nessa parte (P/N).

1.2. Paradigmas de Programação Paralela

Para usar um sistema paralelo, dividimos a carga de trabalho entre os processadores de forma a diminuir o tempo de execução. Essa carga consiste de operações a serem realizadas sobre os dados de entrada produzindo dados de saída. Nesse sentido, durante sua execução, um programa paralelo deve transmitir dados entre os processos localizados nos diferentes processadores. Existem duas formas de transmissão, uma explícita – através de troca de mensagens e outra implícita – através de memória compartilhada. Obviamente, devemos ponderar que necessitamos de uma implementação de memória 'comum' (hardware ou software) para o uso de memoria compartilhada.

1.2.1. Troca de Mensagens

Quando um programa paralelo usa troca de mensagens, significa dizer que o envio e recepção de dados se dá através de primitivas de comunicação, como *sends* e *recvs*. O padrão atual para troca de mensagens é o MPI (*Message Passing Interface*). O MPI define o protótipo das funções disponíveis para uso pelo desenvolvedor de programas paralelos. A funcionalidade propriamente dita é codificada por alguma implementação

do MPI, a exemplo temos OpenMPI, MPICH, GridMPI, entre outros. Na tabela abaixo temos algumas das primitivas MPI mais usadas.

Tabela 1.1. Funções MPI mais usadas.

Função	Uso
MPI_Init	Inicializa o ambiente paralelo.
MPI_Finalise	Destrói o ambiente paralelo
MPI_Comm_size	Retorna a quantidade de processos participantes na execução.
MPI_Comm_rank	Retorna o índice do processo corrente no grupo de processos participantes.
MPI_Send	Envia uma mensagem contendo um ou mais elementos dados.
MPI_Recv	Recebe uma mensagem contendo um ou mais elementos de dados
MPI_Probe	Verifica se tem alguma mensagem para ser recebida.
MPI_Isend	Versão assíncrona do MPI_Send.
MPI_Irecv	Versão assíncrona do MPI_Recv.
MPI_Wait	Espera por um envio ou recepção assíncrona.
MPI_Barrier	Sincroniza um grupo de processos em um ponto do código.

O MPI fornece uma abstração de forma que programas paralelos executem transparentemente independente do hardware de rede, tanto faz se uma rede é *Fast Ethernet* ou Infiniband.

1.2.2. Memória Compartilhada

No caso de comunicação utilizando a memória compartilhada, cada linha de execução ou processo escreve e lê dados da memoria comum usando a codificação normal. Nesse caso fazemos uso de primitivas de sincronização para garantir integridade dos dados (sequencia de leitura / escrita consistente). Atualmente a implementação de memória compartilhada mais usada é a OpenMP. Nessa implementação, diretivas de compilação delimita as áreas paralelizáveis do código de forma a que o compilador insira o mecanismo de expansão de linhas de execução concorrente sobre essa área, tirando explorando o paralelismo.

Tabela 1.2. Diretivas OpenMPI mais comuns.

Diretiva	Uso
parallel	Define uma região paralela.
private	Define quais variáveis são privadas de cada linha de execução.
shared	Define quais variáveis são compartilhadas entre as linhas de execução.
firstprivate	Permite que as variáveis privadas sejam inicializadas.
reduction	Permite operar sobre uma lista de variáveis.
if	Decide sobre a criação de um time de linhas de execução.
copyin	Copia um mesmo valor a variáveis privadas de cada linha de execução.
for	Define uma região de repetição a ser paralelizada.
sections	Divide uma região em sessões.
section	Define uma sessão dentro de uma região.
barrier	Define um ponto de sincronização
critical	Definie uma região de exclusão múltipla.
master	Define uma região é executada sempre pela linha principal.
atomic	Define um endereço de memoria para atualizações atômicas.
flush	Força uma visão consistente da memória.
ordered	Força que a execução tenha a mesma ordem que a sequencia.

Dentro de um código que faz uso do OpenMP são disponibilizados também diversas funções que permitam que o código perceba e interaja com a execução paralela. Na tabela abaixo temos a lista das funções mais usadas.

Tabela 1.3. Funções OpenMPI mais usads.

Funções	Uso
omp_init_lock	Inicializa uma variável de sincronização (lock).
omp_get_num_threads	Retorna o numero de linhas de execução.
omp_get_max_threads	Retorna o numero máximo de linhas de execução permitidoo.
omp_get_thread_num	Retorna o identificador da linha de execução corrente.
omp_get_num_procs	Retorna o numero de processadores disponíveis.

1.2.3. Aplicabilidade

O uso do OpenMP é recomendado em maquinas multicore ou com mais de um processador. A analise deve ser feita sobre o problema do acesso concorrente à memoria RAM nesses sistemas, penalização e problemas decorrente da verificação de integridade dos dados da memoria cache de cada processador/core e penalização de troca de contexto. Caso de máquinas com soft cores (*hyperthreading*), geramente obtemos melhor beneficio do uso de OpenMP do que dobrar quantidade de processos MPI.

1.2.4. Paradigma de Comunicação

Quando usamos troca de mensagens na programação paralela podemos ter diferentes paradigmas de comunicação. Em cada paradigma temos um padrão de comunicação onde os processos assumem papeis com funções distintas. Os paradigmas básicos são o Master-Worker e o SPMD (Wilkinson & Allen, 2004).

Master-Worker

No paradigma de comunicação Master-Worker os processos desempenho dois papeis distintos, um *Master*, processo responsável por distribuir o trabalho e coletar o resultados e *Workers*, processos responsáveis em processar os trabalhos distribuídos pelo processo Master (Wilkinson & Allen, 2004).

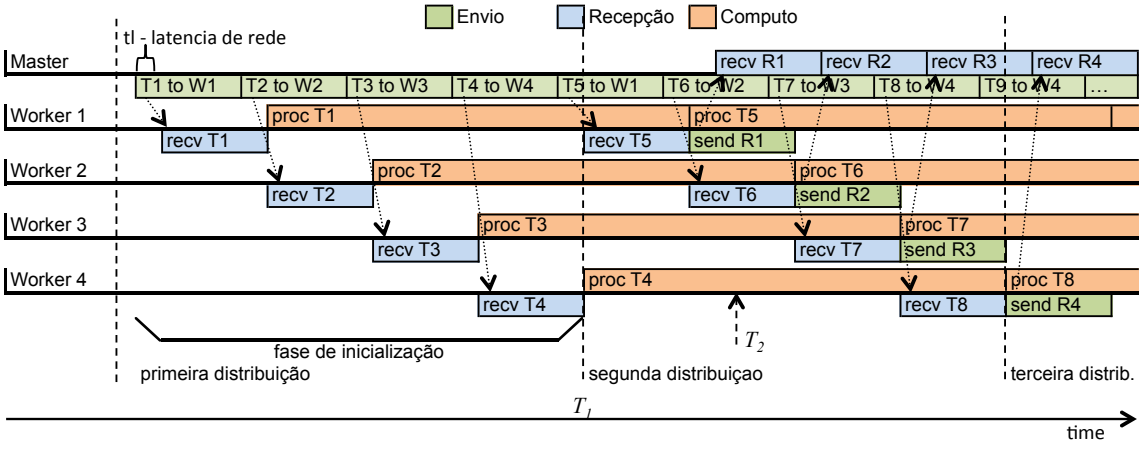

Figura 1.1. Apresenta um exemplo de execução com um processo *Master* e quatro processos *Workers*. Note que existe um tempo de inicialização até ter todos os processos trabalhando.

SPMD

No paradigma de comunicação SPMD, *Singre Program Multiple Data*, cada processo executa a mesmo trabalho comunicando com os processos vizinhos. De uma forma

simples, cada processo troca dados com os vizinhos e processa sua unidade de trabalho repetitivamente (Wilkinson & Allen, 2004). Como o processo de troca de dados consiste de comunicação e por consequente, ponto de sincronização, a diferença de processamento em um processo pode atrasar o processamento dos demais. Esse paradigma geralmente se beneficia de diferentes topologias de rede, onde um nodo de computo dispõe de mais de um canal de comunicação (interface de rede).

1.3. Experimentação

O desenho do experimento é essencial para o processo de análise de desempenho. Dentre as características de um bom experimento, a melhor é a documentação que permite a sua reprodução. Geralmente nos perdemos repetindo hipóteses por não ter documentado experimentos corretamente. Essa documentação se dá usando uma taxonomia característica que vemos a seguir.

1.3.1. Taxonomia

Quando desenhamos um experimento, temos que ter em conta que sua descrição deve ser feita através de dados que nos possibilite sua reprodução. Geralmente um experimento tem um motivo de existir além de provê apenas o resultado da computação. Por exemplo, quando variamos o numero de processos, estamos avaliando qual o impacto desse numero no desempenho. Ou podemos estar avaliando se a nossa implementação é escalável ou a que nível. Podemos, a partir desses dados então, estimar qual o percentual da nossa aplicação é serial, usando a lei de Amdahl. Antes de exercitar o processo experimental devemos definir qual a taxonomia dos dados que caracterizam um experimento (Jain, 1991).

Parâmetros: Todo valor que pode ser mudado no experimento pode ser definido como parâmetro. Note que não se trata de parâmetros do executável e sim de informações necessárias para caracterizar o experimento. Por exemplo, podemos definir como parâmetros o numero de processos, o numero de linhas de execução, quantidade de buffers usada, uso de primitivas síncronas ou assíncronas, configuração de compilação, entre outras. A definição dos parâmetros permite que comparemos diferentes experimentos.

Fatores: Quando comparamos um ou mais experimentos, analisamos a variação de um ou mais parâmetros. Esses parâmetros escolhidos para serem variados e analisados se chamam fatores. Geralmente temos comparação considerando apenas um fator. Quando consideramos mais de um fator devemos gerar um grupo de experimentos que cubra todos os casos possíveis.

Índices: Índices são valores resultantes de um experimento. Esses valores são usados para a comparação entre os mesmos. Podemos ter, por exemplo, o tempo de execução como um índice de desempenho. Na programação paralela os índices mais comuns são: tempo de execução, *speedup*, eficiência (relação entre tempo de processador ocioso e ativo) e, mais recentemente, consumo de energia.

Replicas: Raramente devemos confiar no resultado de uma execução. Para analisar a estabilidade dos índices realizamos diversas execuções com um mesmo conjunto de parâmetros. Se executamos, por exemplo, três vezes um mesmo experimento, cada execução é dita uma replica.

1.3.2. Processo Sistemático para Avaliação de Desempenho

No processo de experimentação necessitamos avaliar diversos aspectos que podem influenciar no sucesso do mesmo. A seguir temos um processo sistemático proposto por (Jain, 1991):

1. *Definir objetivos e definir o sistema*: apesar de parecer incomum isso ocorre com frequência. Quando definimos um objetivo de forma clara nos concentramos em desenhar o experimento para atender a esse objetivo. É o primeiro passo de um processo de avaliação de desempenho. Outra questão é definir os limites do sistema. Se, por exemplo, consideramos a aplicação paralela como o sistema a ser avaliado, dados como a ocupação da placa de rede estão fora do sistema.
2. *Definir as entradas e saídas*: todo sistema ou aplicação pode ser avaliado como um grupo de serviços. Avaliamos como um serviço algo que consome dados e produz algo. Em um banco de dados, por exemplo, consome consultas e devolve resultados. Analisar as entradas e saídas ajuda a definir sobre que aspectos faremos nossa analise.
3. *Selecionar as métricas*: o critério usado para comparar o desempenho entre sistemas é chamado de métrica. Em geral as métricas estão associadas a velocidade, precisão e disponibilidade de serviços. No processo de avaliação de desempenho geralmente temos índices baseados nas métricas selecionadas.
4. *Listar parâmetros*: nessa etapa se lista todos os parâmetros que podem influenciar o desempenho. Os parâmetros podem ser divididos entre parâmetros do sistema e parâmetros da carga de trabalho. Os parâmetros do sistemas configuram o sistema independente do trabalho e os parâmetros da carga de trabalho caracteriza a mesma sobre aspectos como tamanho, complexidade e/ou frequência.
5. *Selecionar fatores para estudo*: de posse da lista de parâmetros, escolhemos qual deles variamos para avaliação. Esses parâmetros são chamados de fatores.
6. *Selecionar a técnica de avaliação*: executar experimento e medir os dados da execução não é a única técnica de avaliação de desempenho. Existem a modelagem analítica e a simulação. Uma vez que conhecemos o comportamento do sistema em função dos parâmetros ou se te um modelo de eventos que se comporta como o sistema real podemos escolher entre as outras técnicas e obter resultados mais rápido que medições de execuções reais.
7. *Selecionar a carga de trabalho*: Um programa ou sistema tem parte de seu comportamento definido pela carga de trabalho. Nessa etapa escolhemos qual a carga de trabalho a ser usado no experimento. O uso de carga inadequada pode tanto saturar um sistema, fazendo com que o mesmo perda desempenho, quanto subutilizar os recursos disponíveis.
8. *Desenho do experimento*: nessa etapa avaliamos quantas execuções devem ser feitas para cobrir as variações dos fatores selecionados. Detalhamos também os passos necessários para a reprodução do mesmo.
9. *Analise e interpretação dos dados*: quando analisamos dados resultantes de um experimento devemos ter em conta que esses podem ter componentes aleatórios,

ou seja, podemos obter variação entre diferentes replicas de uma mesma execução.

10. *Apresentar os resultados*: de posse dos dados relevantes podemos construir a argumentação e apresentação dos resultados.

1.3.3. Erros Comuns

Durante o processo de analise podemos incorrer em uma serie de erros. Esses erros podem nos direcionar para conclusões equivocadas. Em seguida analisamos uma lista de possíveis equívocos (Jain, 1991):

- Não ter objetivos.
- Escolher objetivos viciados.
- Usar abordagem não sistematizada.
- Analisar sem conhecer o problema.
- Usar métricas inadequadas de desempenho.
- Usar carga de trabalho não representativa.
- Usar Técnica de avalição equivocada.
- Desprezar de parâmetros importantes.
- Ignorar fatores significantes.
- Usar desenho inadequado do experimento.
- Escolher nível de detalhe inapropriado.
- Suprimir a analise.
- Fazer uma analise errônea.
- Não fazer analise de sensibilidade.
- Ignorar erros na entrada.
- Tratar incorretamente as saturações.
- Assumir estabilidade do sistema.
- Ignorar variabilidade.
- Fazer analise muito complexa.
- Apresentar incorretamente os resultados.
- Ignorar aspectos políticos.
- Omitir hipóteses e limitações.

1.4. Instrumentação e Monitoração de Desempenho

Existem medidas que podem ser feitas externas à aplicação paralela, como por exemplo o tempo total de execução, quantidade de bytes lidos e escritos no disco por processo,

uso de memória de cada processo, entre outros. Para monitoramos o comportamento específico da aplicação devemos modificar seu código fonte de forma a possibilitar a aquisição de dados de desempenho, como, por exemplo, tempo ocioso, tempo em cada região de código ou tempo gasto nas funções de comunicação. Essa modificação é chamada de instrumentação.

A instrumentação pode ser estática ou dinâmica. É estática quando é feita no código fonte ou objeto antes da compilação e dinâmica quando é feita depois da compilação. Pode ser também automática ou manual. No caso de instrumentação manual inserimos explicitamente no código a chamada às primitivas de para coleta de dados e a na automática esse processo é feito através de ferramentas sem necessidade de alterar o código fonte original.

Nos concentraremos em duas técnicas de coleta de dados: *profilling* e *tracing*. Cada uma produz tipos diferente de dados, assim como possibilita uma visão distinta do comportamento da aplicação.

1.4.2. *Profiling*

O processo de *Profiling* consistem e periodicamente parar a execução do programa e verificar qual a função está executando. Isso pode ser feito analisando o contador de programa e percorrendo a pilha de chamadas dos procedimentos, associado com os símbolos gerados na compilação. Com isso se espera que estatisticamente a qualidade apresentada na amostragem corresponda ao tempo que o programa passou em cada região do código do programa.

1.4.3. *Tracing*

A técnica de *tracing* consiste em guardar registro da entrada e saída do processador em regiões do código fonte da aplicação. A definição das regiões podem ser explícitas ou implícitas. No caso de regiões explícitas usamos chamadas a primitivas demarcando o inicio e fim de cada região. Já no caso de regiões implícitas, se usa as definição de funções da linguagem de programação, gerando automaticamente um registro antes de executar a função e outro depois da execução do mesmo.

Outra técnica comum é o uso de bibliotecas fazendo proxy das funções que queremos fazer *tracing*. Um exemplo comum disso é a biblioteca MPE. Essa biblioteca se coloca entre o programa paralelo e a implementação MPI de forma a poder registrar todas as chamas às funções de comunicação.

Precisão do Relógio

Quando trabalhamos com medição de tempo usamos primitivas para ler o valor do relógio nos instantes que queremos medir o tempo de execução. No caso da técnica de *tracing*, por exemplo, lemos o valor do relógio quando entramos e quando saímos de uma função. O problema é que, nos sistemas computacionais, o relógio incrementa por unidades não tão finas pelo sistema operacional. Por exemplo, a função *gettimeofday* do Linux tem uma resolução de microssegundos.

1.4.3. Distorções e Overheads

Quando usamos qualquer uma das duas técnicas de monitoração temos uma inclusão de computo extra na execução do programa. Esse computo resulta em mais tempo de execução, assim como pode modificar o comportamento do programa monitorado. Essa distorção no tempo de execução é chamada de *overhead* da monitoração.

A técnica de *sampling* geralmente tem overhead constante e relacionado com o intervalo de amostragem. Quando menor o intervalo, maior a precisão de quais regiões estão sendo executadas e maior o *overhead* causado pela monitoração. Se aumentamos o tempo entre amostragem, teremos uma menor precisão já que chamadas a funções de curta duração poderão não ser detectadas, mas teremos um menor *overhead* no tempo de execução.

Já a técnica de *tracing* depende da frequência de geração de registros históricos. Se habilitamos o *tracing* de uma função de curta duração usada frequentemente, teremos um volume grande de dados históricos e uma grande penalização no tempo de execução. Algumas implementações de *tracing*, como a do TAU, permite desabilitar a geração de registros caso se detecte que a função é chamada frequentemente.

1.5. Análise de Desempenho

Quando analisamos o desempenho de uma aplicação geralmente temos como objetivo entender porque a aplicação comporta com relação a um referencial definido. Por exemplo, podemos tomar como referencial a lei de Amdahl, ter medido as regiões seriais e paralelas da aplicação e comparar a relação de *speedup* com o modelo teórico.

O processo geralmente é incremental e passa pela formulação de hipóteses, modificação do código, execuções de experimentos, comparação entre fatores considerando os índices e métricas selecionados. Dentre os problemas de desempenho de programas paralelos temos:

- *Envio tardio de mensagem*: isso acontece quando um processo demora para enviar uma mensagem necessária para outro processo em execução. Nesse caso o processo que espera pela mensagem ficar ocioso até que a mesma seja enviada.

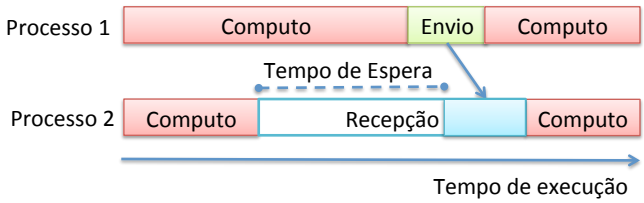

Figura 1.2. Exemplo de envio tardio entre dois processos usando troca de mensagens. O processo 2 fica esperando pela mensagem a ser enviada pelo processo 1.

- *Recepção tardia de mensagem*: acontece quando um processo envia dados maiores que seu buffer de envio sem que o processo destino dessa mensagem inicie a recepção da mesma. Nesse caso o envio é bloqueado, ficando o processo que envia ocioso até que seja iniciada a recepção no processo destino.

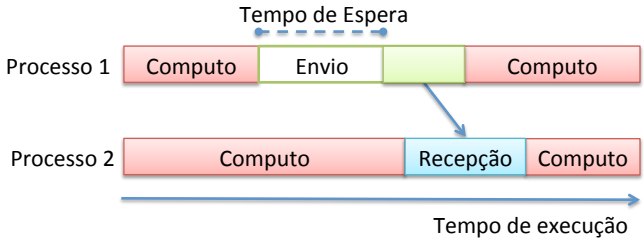

Figura 1.3. Exemplo de recepção tardia entre dois processos usando troca de mensagens. O processo 1 fica esperando o processo 2 iniciar a recepção da mensagem.

- *Desbalanceamento entre computo e comunicação*: a carga de comunicação limita a escalabilidade enquanto o computo determina o desempenho. Podemos dizer que os aspectos da comunicação, por limitar a escalabilidade, limita o máximo de computo que pode ser usado, por consequentemente tem grande impacto no desempenho de um programa em um sistema paralelo.
- *Granularidade incorreta*: a escolha do tamanho da divisão do trabalho tem impacto direto no desempenho. Granularidade mais fina geralmente impacta em maior influencia do tempo gasto nas comunicações com relação ao tempo de computo. Já granularidades mais grossas aumentam o tempo de alcançar o regime de execução assim como finalização e coleção dos resultados. Outro ponto importante é impacto no uso da memória *cache*. Se trabalharmos com granularidades muito finas, corremos o risco de não ter uma boa taxa de acertos, tendo uma grande penalização no desempenho refletido no tempo de execução de cada divisão do trabalho.
- *Boa correlação entre números de processos e linhas de execução (threads):* quando trabalhamos com maquinas multicore temos a opção de mapear um processo por core ou um processo por máquina explorando a multiplicidade de cores com memória compartilhada. Essa correlação depende tanto da aplicação quanto da arquitetura de concorrência pelos recursos comuns entre os cores, como: rede e memória principal.

1.6. Ferramentas

Nessa sessão trataremos de ferramentas disponíveis para monitoração e analise de dados de desempenho de aplicações paralelas e sequenciais.

1.6.1. Código Sequencial

Existe disponíveis nos sistema Linux diversos utilitários que podem ser usados no processo de analise de desempenho. Alguns já estão disponíveis no sistema operacional, como o *gprof* e *vmstat*, e outros em pacotes, como o *oprofile*.

gprof

O utilitário *gprof* está disponível na maioria dos sistemas Unix/Linux atuais. Amostra os tempos gastos em cada função e provê saídas com gráfico de chamadas de funções e métricas como:

- Tempo gasto em cada função.

- Identificador de cada função.
- O percentual de tempo de CPU gasto em cada função e todas funções chamadas dentro de cada função.
- O número de vezes que a função foi chamada.
- O número de chamadas feitas em cada função.

O *gprof* tem como entrada o arquivo de coleta de métricas gerado por executáveis compilados com a configuração para gerar esses dados. Isso no Linux/GCC é feito através a opção de compilação '*–pg'*. Os binários compilados com essa opção gerarão a cada execução um arquivo contendo os dados referentes a aquela execução.

vmstat

O utilitário *vmstat* (Virtual Memory Statistics) permite a coleta de dados do sistema de forma periódica. Serve para entender o efeito da execução em cada maquina, assim como identificar distorções como uso de swap, que penaliza bastante a execução. Com o vmstat temos informações sobre os processos em execução, consumo da memória, uso do swap, uso do disco, quantidade de interrupções e trocas de contexto e métricas especificas da CPU como: tempo em modo protegido, tempo em modo usuário e tempo ocioso.

OProfile

O OProfile (Levon, 2000) consiste de ferramentas que possibilita o uso de *profilling* a nível de sistema operacional. Com isso podemos fazer *profilling* tanto de processos específicos como de vários processos. O processo é bastante simples, disponibiliza os comandos da tabela abaixo:

Tabela 1.2. Comandos disponibilizados pelo *OProfile*.

Comandos	Uso
opcontrol	Permite iniciar e parar a sessão de *profilling*.
opreport	Apresenta um relatório com os dados coletados.
opgprof	Apresenta os dados no formato semelhante ao gprof.
opannotate	Apresenta os dados de desempenho associado ao código fonte.
oparchive	Empacota as informações sobre os executáveis de forma a possibilitar analise em outra máquina diferente da maquina onde as informações foram coletadas

1.6.2. OPARI

O *OpenMP Pragma And Region Instrumentor* (OPARI) é uma ferramenta de instrumentação automática que gera um novo código fonte contendo instrumentação a partir de um código que faz uso das primitivas OpenMP. O OPARI foi desenvolvido como parte dos projetos KOJAK e TAU e possibilita coleta de métricas na programação com OpenMP. Com o OPARI podemos analisar o tempo gasto para entrar na região paralela, o tempo gasto no código paralelizado, o tempo gasto em sincronizações e o tempo gasto para sair de cada região paralela (Fürlinger, 2010) (Mohr, Malony, Shende, & Wolf, 2001).

1.6.3. PAPI

PAPI, *Performance Application Programing Interface* (Terpstra, Jagode, You, & Dongarra, 2010) e (Browne, Dongarra, Garner, N., & Mucci, 2000), é um modulo de *kernel* no Linux que provê uma forma de acesso aos contadores de desempenho disponíveis nos processadores. Com esses contadores é possível verificar o nível de acerto em memoria cache, precisão do preditor de saltos assim como outros detalhes de arquitetura. Os dados disponibilizados pelo PAPI são usados por diversas ferramentas de analise de desempenho.

Para instalarmos PAPI em um sistema devemos compilar um modulo para a versão especifica do *kernel* que usamos. Na tabela abaixa temos, a lista de eventos disponibilizado pelo PAPI.

Tabela 1.2. Alguns eventos que podem ser lidos usando PAPI.

Nome Evento	Sgnificado
PAPI_INT_INS	Operações com Inteiros
PAPI_FP_INS	Operações com Ponto Flutuante
PAPI_LD_INS	Instruções *Load*
PAPI_SR_INS	Instruções *Store*
PAPI_BR_INS	Total de saltos
PAPI_VEC_INS	Instruções Vector/SIMD
PAPI_FLOPS	Operações com Ponto Flutuante por segundo
PAPI_RES_STL	Ciclos parado a espera de um recurso
PAPI_FP_STAL	Unidades FP paradas
PAPI_TOT_CYC	Total de ciclos de relógio
PAPI_IPS	Instruções executadas por segundo
PAPI_LST_INS	Total de instruções *load*/*store* executadas
PAPI_SYC_INS	Instruções de Sincronização Executadas
PAPI_L1_DCM	L1 falha da cache de dados
PAPI_L1_ICM	L1 falha da cache de Instruções
PAPI_L2_DCM	L2 falha da cache de dados
PAPI_L2_ICM	L2 falha da cache de Instruções
PAPI_L1_TCM	L1 total cache misses
PAPI_L2_TCM	L2 total cache misses
PAPI_CA_SHR	Requisição de acesso para uma linha de acesso compartilhado de cache (SMP)
PAPI_CA_CLN	Requisição de acesso para uma linha limpa de cache (SMP)
PAPI_CA_INV	Invalidação de linha de cache (SMP)
PAPI_TLB_DM	Falha na TLB de dados
PAPI_TLB_IM	Falha na TLB de instruções
PAPI_MEM_SCY	Ciclos esperando para acesso a memoria
PAPI_MEM_RCY	Ciclos esperando para leitura de memoria
PAPI_MEM_WCY	Ciclos esperando para escrita de memoria
PAPI_HW_INT	Interrupções de Hardware
PAPI_TOT_INS	Total de instruções executadas

1.6.4. MPE

O *MPI Parallel Environment* (MPE), disponível em (Performance Visualization for Parallel Programs), consiste de um conjunto de ferramentas simples e uteis para instrumentar e analisar o desempenho de aplicações paralelas baseadas em MPI. Dentre as funcionalidades disponíveis temos:

- Um conjuntos de bibliotecas para *profilling* e *tracing* que são incorporadas na aplicação paralela no processo de compilação. Essas biblioteca geram arquivos contendo os dados da instrumentação para análise feita após cada execução (*postmotem*).
- Disponibiliza scripts que facilitam o processo de compilação usando MPE, diversos utilitários para manipulação de arquivos no formato CLOG2 e SLOG2.
- Disponibiliza um visualizador de dados de *tracing* no formato SLOG2 chamado Jumpshot.

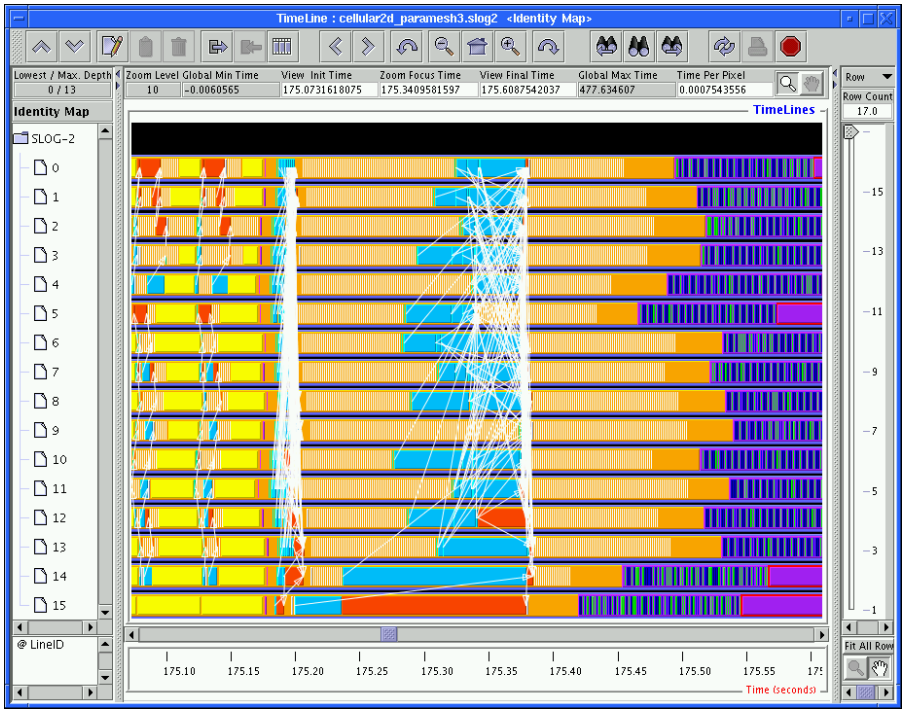

Figura 1.4. Exemplo de uma visualização usando o Jumpshot, retirada de (Performance Visualization for Parallel Programs). Cada barra vertical significa um processo. As linhas brancas são as comunicações entre processos. Cada cor está associada a uma primitiva (ou evento configurado).

1.6.5. TAU

O TAU (Tuning and Analyzes Utilities), detalhado em (Shende & Malony, 2006), é um conjunto de ferramentas implementadas para ajudar desenvolvedores a monitorar e analisar o desempenho de suas aplicações. Essas ferramentas permitem diversos tipos de instrumentação, aquisição de dados para analise de desempenho, formatos de dados históricos de *tracing*, conversão entre formatos e programa de visualização Os formatos de instrumentação suportados são:

- Código Fonte – suporta um lista extensiva de linguagens assim como pré-processamento de código fonte para inserção de instrumentação.
- Código Objeto – disponibilizado através de uma modificação do processo de compilação o qual permite a inserção de instrumentação no código binário depois da fase de otimização.

- Bibliotecas proxy – suporta proxy da biblioteca MPI para monitoração do seu uso.
- Código Binário – usa a biblioteca de reescrita dinâmica DyninstAPI para modificar o código binário em execução, inserindo o código necessário para a instrumentação.
- Componentes de Software – permite geração de registro de chamadas à interface do componente através da geração de componentes proxy contendo código de instrumentação.
- Máquina Virtual – usa a Java Virtual Machine Profiler Interface (JVMPI) para registrar TAU como agente responsável pelo *profilling*. Com isso passa a receber informações sobre a VM e chamadas de funções.

As ferramentas presentes no TAU pode coletar informações sobre o comportamento da aplicação em diferentes formas. Podemos fazer *profilling* de uma região, gravar o tempo gasto em cada região. Permite gerar uma amostra para cada execução de uma região ou o tempo total acumulado em cada região. Possibilita o uso de *tracing* explicito em implícito. A configuração é bastante flexível e permite ser preciso, o que diminui o overhead gerado pela instrumentação. Os dados coletados são armazenados em um formato próprio, mas, possui uma grande variedade de conversores entre diversos formatos conhecidos de histórico de *tracing*.

Um dos aspectos mais interessantes do TAU é que permite criar um banco de dados com os dados coletados em cada experimento. Com isso, usando as ferramentas de visualização, podemos ter hipóteses sobre causa e efeito da variação dos fatores nos experimentos. A figura abaixo mostra uma das muitas formas de visualização dos dados coletados.

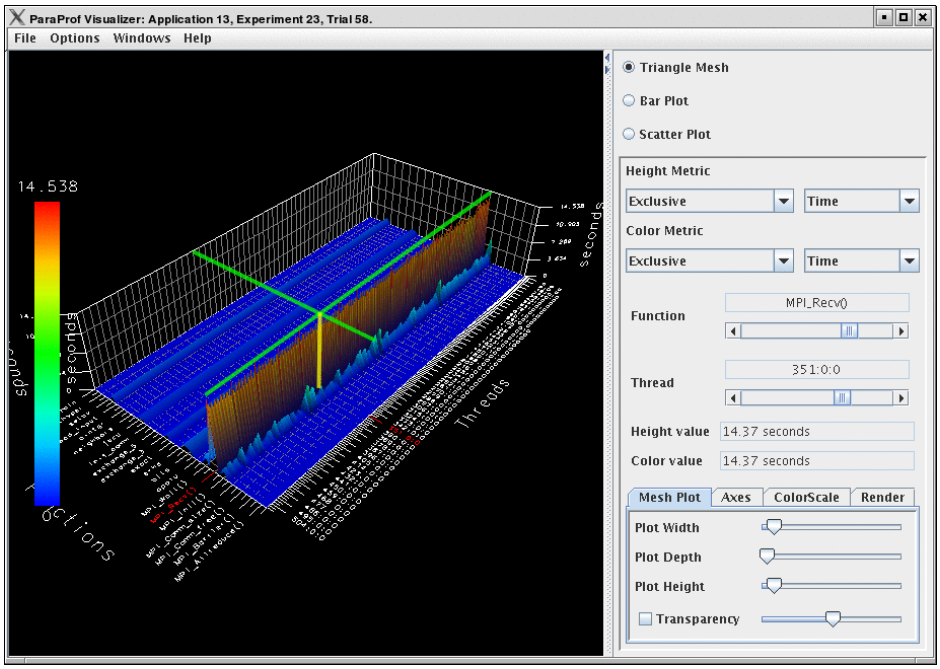

Figura 1.5. O ParaProf permite apresentar o tempo gasto em cada função em cada thread em uma visualização 3D.

1.6.7. KOJAK

O projeto KOJAK (*Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks*) (Mohr & Wolf, 2004) consiste de um grupo de ferramentas que possibilita ir alem do processo de monitoração e prover analise automática de desempenho para aplicações paralelas. Faz uso de uma base de dados de problemas comuns que podem gerar gargalo dentro de aplicações paralela representados através de um padrão. A ferramenta busca esses motivos de ineficiências dentro dos dados gerados pela instrumentação e apresenta de uma forma intuitiva.

Faz uso de bibliotecas proxy para geração de dados de *tracing* e coleta contadores disponibilizados por PAPI. O processo de instrumentação usando o KOJAK seguem os seguintes passos:

- Passamos o programa do usuário pelas ferramentas de instrumentação do OPARI e TAU gerando um programa instrumentado.
- Compilamos o programa instrumentado, gerando um executável.
- Executamos o programa no sistema paralelo e esse, devido a instrumentação, gera arquivos históricos no formato EPILOG.
- Esse arquivo EPILOG é então processado pelo analisador, chamado EXPERT, que faz uso de uma linguagem de reconhecimento e analise de eventos, EARL, para encontrar os padrões de ineficiências e gerar o resultado da analise.
- Esse resultado da analise pode ser visualizado pelo apresentador do EXPERT de forma intuitiva.

Na figura abaixo temos o desenho de um processo de analise de desempenho usando o KOJAK (Mohr & Wolf, 2004).

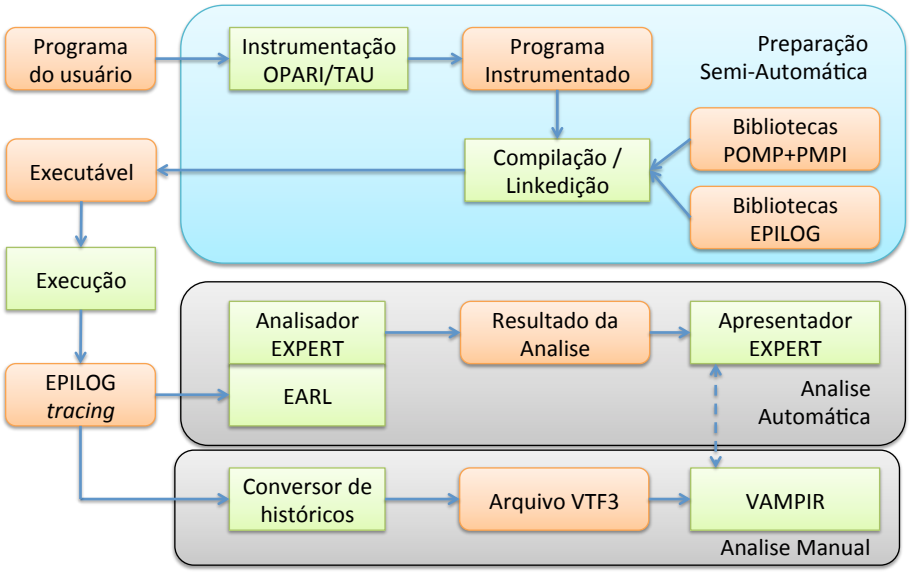

Figura 1.6. Sequencia de execução usando o conjunto de ferramentas do KOJAK, traduzido de (Mohr & Wolf, 2004). Note que o VAMPIR, TAU e OPARI são projetos externos implementados e mantidos por terceiros.

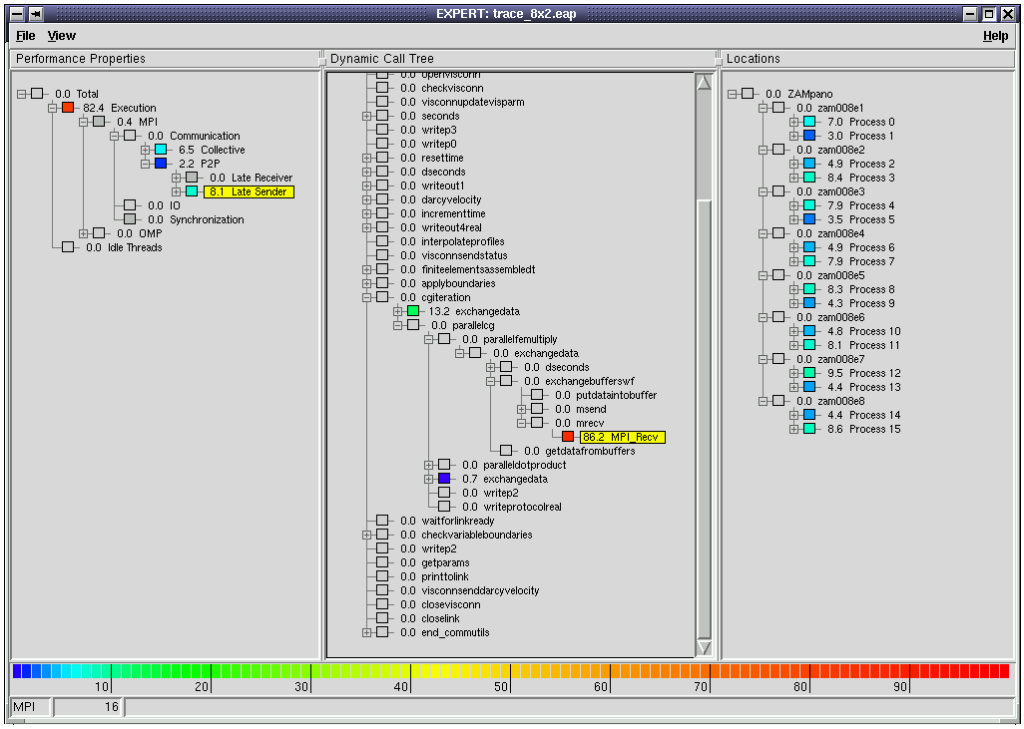

Figura 1.7. Tela de visualização dos problemas de desempenho apresentada pelo EXPERT (Mohr & Wolf, 2004). Apresenta uma escala de cores significando a penalização.

1.6.8. VAMPIR

VAMPIR e VAMPIR NG são ferramentas disponíveis comercialmente que permite a visualização das informações de *tracing* geradas por aplicações instrumentalizadas. É bastante utilizada nos grandes centros de supercomputação e seu processo de instrumentação está integrada na distribuição MPI OpenMPI. A apresentação é semelhante ao Jumpshot do MPE (apresentado anteriormente). Na figura abaixo temos um exemplo de tela do VAMPIR NG (Brunst, Kranzlmüller, & Nagel, 2005).

1.7. Considerações Finais

Nesse trabalho apresentamos uma introdução à analise de desempenho em sistemas paralelos, trabalhando principalmente na sistematização do processo experimental, assim como apresentação de ferramentas e técnicas de monitoração de dados relevantes para essa analise. Também apresentamos problemas comuns que impactam no desempenho de aplicações paralelas e ferramentas disponíveis para analise e detecção desses problemas.

Usando o processos sistemático de analise de desempenho apresentado temos melhor rigor cientifico e conseguimos resultados de forma rápida e estruturada, sem experimentos desnecessários. Quase todas as ferramentas apresentadas, salvo o VAMPIR, são de acesso livre, o que nos deixa boas opções para trabalhar na analise de desempenho de aplicações paralelas.

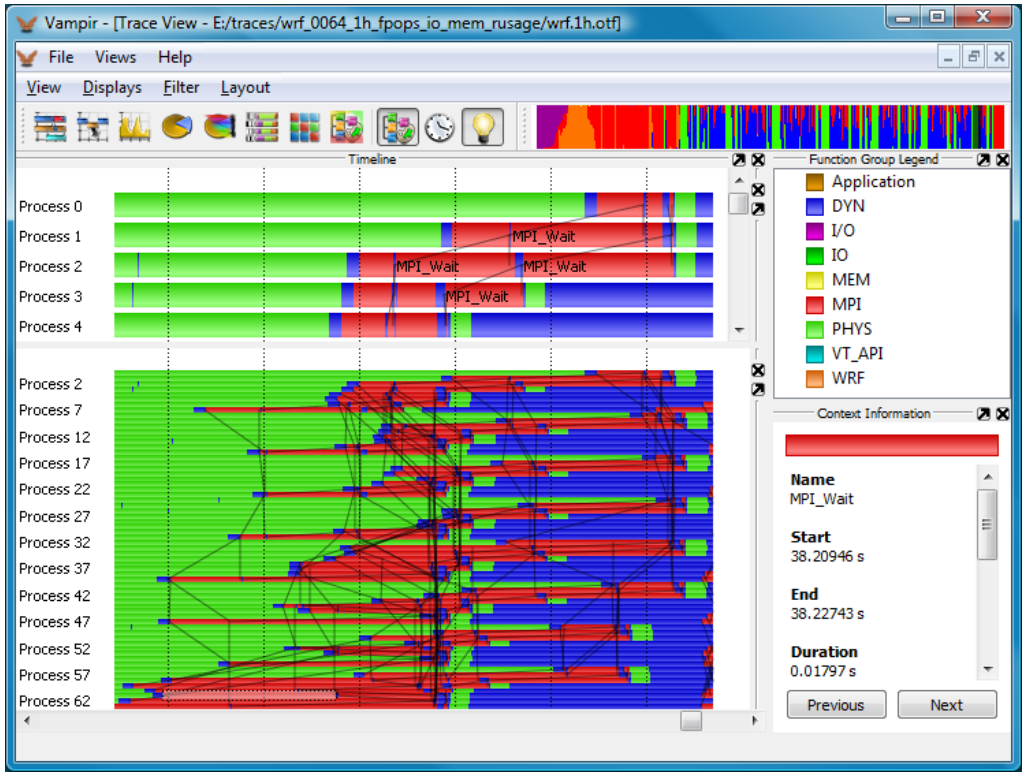

Figura 1.8. Visualização da linha de tempo do VAMPIR NG. Nela podemos ver claramente a troca de mensagens e quanto tempo se gasta em na aplicação, IO, MPI e o tempo de overhead da instrumentação.

Um problema existente em aplicações paralelas grandes é a geração de dados históricos da técnica *tracing*. Quanto maior o numero de processadores, maior a geração de dados e por consequente, maior a complexidade dos dados gerados e apresentação em ferramentas de visualização. Existem pesquisas tanto na forma de representação de dados gerados por sistemas de larga escala quanto em formas alternativas de visualização considerando milhares de processadores e milhões de mensagens trocadas entre eles.

1.8. Referências

Wilkinson, B., & Allen, M. (2004). *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc.

Brunst, H., Kranzlmüller, D., & Nagel, W. (2005). Tools for scalable parallel program analysis-Vampir NG and DeWiz. *Distributed and Parallel Systems* , 93--102.

Browne, S., Dongarra, J., Garner, N., H. G., & Mucci, P. (2000). A portable programming interface for performance evaluation on modern processors. *International Journal of High-Performance Computing Applications , 14* (3), 189--204.

Fürlinger, K. (2010). OpenMP Application Profiling - State of the Art Directions for the Future. *Procedia Computer Science , 1* (1), 2107--2114.

Jain, R. (1991). *The Art of Computer System Performance Analysis*. Wiley.

Levon, J. (2000). *OProfile manual*. Acesso em 8 de 2011, disponível em SourceForge OProfile: http://oprofile.sourceforge.net/doc/

Lilja, D. (2005). *Measuring computer performance: a practitioner's guide*. Cambridge Univ Pr.

Mohr, B., & Wolf, F. (2004). KOJAK - A tool set for automatic performance analysis of parallel programs. *Euro-Par 2003 Parallel Processing* , 1301--1304.

Mohr, B., Malony, A., Shende, S., & Wolf, F. (2001). Design and prototype of a performance tool interface for OpenMP. *The Journal of Supercomputing , 23* (1), 105--128.

Performance Visualization for Parallel Programs. (s.d.). Acesso em 08 de 2011, disponível em Laboratory for Advanced and Numerical Software: http://www.mcs.anl.gov/research/projects/perfvis/

Shende, S. S., & Malony, A. D. (2006). The TAU parallel performance system. *International Journal of High Performance Computing Applications , 20* (2), 287.

Terpstra, D., Jagode, H., You, H., & Dongarra, J. (2010). Collecting performance data with papi-c. *Tools for High Performance Computing 2009* , 157--173.