Escalonamento Através de Perfilamento em Sistemas Multi-core

Emilio Francesquini

Departamento de Ciência da Computação - Universidade de São Paulo (USP)
Rua do Matão, 1010 - São Paulo - SP - Brasil – 05508-090

emilio@ime.usp.br

Para obter informações suficientes para efetuar um bom escalonamento de aplicações em ambientes multi-core, precisamos, primeiramente, obter informações sobre o hardware e sobre o software que este sistema se propõe a executar. As propriedades do hardware, apesar de serem relativamente estáticas, podem mudar ao longo do tempo: pode-se adicionar mais memória, trocar um disco rígido por outro com uma velocidade de acesso superior, em clusters e sistemas NUMA pode ocorrer a adição de novos nós, etc.

Já o perfilamento de softwares é bem mais complicado. Diferentemente do hardware os softwares não são, em sua maior parte, estáticos como o hardware. Um mesmo software, sem alteração alguma no seu código, dependendo apenas das suas entradas pode mudar o seu comportamento completamente. Em certos casos fica muito difícil determinar de antemão quais serão as características de execução de uma determinada aplicação antes de, de fato, executá-la.

O que propomos é, então, fazer o perfilamento das aplicações e do hardware de forma que automaticamente o sistema escalonador possa verificar as características de hardware e daquela execução específica da aplicação para tentar escaloná-la da melhor forma. Para fazer tal escalonamento precisamos de ferramentas que nos auxiliem nesta tarefa. Dividimo-nas em duas categorias, as de perfilamento de hardware e de software.

1. Perfilamento de hardware

Os recursos de hardware são limitados e isso faz com que seja necessário o escalonamento de forma a permitir com que as aplicações executem de forma satisfatória. No entanto, apesar dos recursos serem sempre limitados, por vezes eles são maiores do que a aplicação precisa. Para saber exatamente quais recursos de hardware podem ser um gargalo e quais não preocupam tanto precisamos de uma maneira de quantificar estes recursos, que é o que as ferramentas brevemente descritas abaixo se propõem.

i. Stream (http://www.streambench.org/)

É uma ferramenta para a medição da largura de banda para acesso à memória existente na máquina. Ela é capaz de dizer com qual velocidade os dados podem ser lidos da memória principal. Os valores obtidos através dessa ferramenta são bem úteis no sentido de que se sabemos que as aplicações que desejamos executar são realmente famintas por acesso à memória, então provavelmente queremos que os seus threads estejam em sockets separados para evitar contenções. A saída (resumida) do Stream é reproduzida abaixo:

```
=== Caught the last thread.
Average cpu bandwidth:  Copy: 2092MB/sec/cpu Scale: 2249MB/sec/cpu Add: 2464MB/sec/cpu
Triad: 2480MB/sec/cpu
Total system bandwidth: Copy: 4184MB/sec  Scale: 4499MB/sec  Add: 4929MB/sec  Triad:
```

```
4960MB/sec
```

ii. Portable Hardware Locality (antigo libtopology) (http://runtime.bordeaux.inria.fr/hwloc/)

Tão importante quanto saber a velocidade de acesso à memória, é importante sabermos qual é a sua capacidade. O tamanho dos caches, do seu arranjo hierárquico e compartilhamento pelos diversos processadores é também muito importante. O Portable Hardware Locality (ou mais resumidamente hwloc) é capaz de nos fornecer todo esse tipo de informação através de uma chamada a um programa, ou através de uma API que é de relativa facilidade para sua utilização.
Abaixo reproduzimos uma saída da execução deste programa em um MacBook 2.4GHz com 4GiB de RAM:

```
$> lstopo -v
System(0KB HP=0*0kB  )
  NUMANode#0(4096MB)
    Socket#0
      L2Cache#0(3072KB)
        L1Cache#0(32KB)
          Core#0
            P#0
        L1Cache#1(32KB)
          Core#1
            P#1
depth 0:  type #0 (System)
absent:        type #1 (Machine)
 depth 1: type #2 (NUMANode)
  depth 2:     type #3 (Socket)
multiple: type #4 (Cache)
     depth 5:  type #5 (Core)
      depth 6: type #6 (Proc)
$>
```

Nessa saída pode-se ver claramente que a máquina tem 4GiB de RAM além de ter um cache L2 de 3072KiB e um cache L1 de 32KiB. Sendo que o cache L1 é específico de cada processador enquanto o cache L2 e a memória principal é compartilhada por esses dois processadores.

iii. Bonnie (http://www.textuality.com/bonnie/)

Assim como algumas aplicações podem ser famintas por acesso a memória, outras podem fazer muitos acessos ao disco e tornar esse recurso de hardware o seu gargalo para a execução. Então, seria de se esperar, que um escalonador em um ambiente de grade/cluster fornecesse para essas aplicações um disco ou uma máquina exclusiva para evitar concorrências. Essas características de cada aplicação deverão ser levantadas pelos perfiladores de aplicação que comentaremos mais adiante, no entanto a capacidade do hardware pode ser medida através de um software chamado Bonnie. O Bonnie quando executado dá uma saída como a que reproduzimos abaixo.

```
File './Bonnie.3255', size: 104857600
Writing with putc()...done
Rewriting...done
Writing intelligently...done
Reading with getc()...done
Reading intelligently...done
Seeker 2...Seeker 3...Seeker 1...start 'em...done...done...done...
              -------Sequential Output-------- ---Sequential Input-- --Random--
              -Per Char- --Block--- -Rewrite-- -Per Char- --Block--- --Seeks---
Machine    MB K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU  /sec %CPU
          100 41438 58.0 56280  9.6 55506  9.0 80925 98.1 1425528 96.8 4665.7  9.4
```

Nela podemos ver as taxas de utilização da CPU e da velocidade de leitura e escrita de dados no disco, e com posse dessas informações o escalonador pode tomar alguma decisão mais informada sobre o formato que o escalonamento de uma aplicação deve possuir.

2. Perfilamento de software

Ao contrário do perfilamento de hardware, o perfilamento de software tende ser uma tarefa bem mais complicada. Isso se dá principalmente pela imensa variedade de comportamentos e variações presentes dentro de apenas uma aplicação que pode mudar de perfil dependendo das entradas que ela recebe. Como é impossível avaliar o perfil da aplicação dados o seu fonte e suas entradas, a melhor maneira de obter informações sobre ela é executando-a e coletando a maior quantidade de dados possível. Com posse desses dados o escalonador juntamente com as informações que ele possui do hardware (obtidas com as ferramentas da seção 1) pode tomar decisões mais informadas sobre o escalonamento das aplicações. Abaixo apresentamos algumas ferramentas que servem a este propósito.

i. GProf (http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html)

O GNU Profiler, ou GProf, é capaz de informar para cada chamada do seu código o tempo que ela leva para executar. Assim, é possível determinar em que chamada, ou em qual função da aplicação os gargalos de tempo de execução estão. Para isto o programa deve ter sido compilado com a flag -pg. Uma saída de exemplo do GProf, está abaixo.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
```

```
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

Essa saída por si só não ajuda muito o escalonador, já que são funções internas ao aplicativo. No entanto, a idéia de se utilizar o GProf é a de, com a ajuda da informação obtida de algumas outras ferramentas que descrevemos abaixo, sermos capazes de avaliar se (e medir quanto) um escalonamento é melhor do que o outro, já que teremos os tempos relativos e absolutos de execução de cada parte da aplicação.

ii. OProfile (http://oprofile.sourceforge.net/)

Esse perfilador pode ser utilizado para medir os mais diversos tipos de informações no sistema. É possível através dele medir o número de cache misses, ciclos de processador, translation-look-aside-buffer (TLB) misses, referências à memória, branch miss-predictions, interrupções, entre outros. A quantidade de informações oferecidas pelo OProfile é imensa, e por ser um perfilador que pode ser executado em modo online um escalonador baseado nas informações obtidas por esta ferramenta pode tomar decisões em tempo de execução para alterar o escalonamento atual, claro considerando todas as informações que foram obtidas pelo passo de perfilamento do hardware. Abaixo colocamos em comparação a saída do OProfile quando avaliando o número de cache misses em um programa feito para ocasionar esse tipo de erro e em outro feito para evitar este tipo de erro.

Saída do perfilador para um programa feito para causar cache-miss:

```
# opreport -l ./cache-miss
CPU: ppc64 POWER5, speed 1656.38 MHz (estimated)
Counted CYCLES events (Processor cycles) with a unit mask of 0x00
(No unit mask) count 1000
samples  %        symbol name
121166   53.3853  inc_second
105799   46.6147  main
```

Saída do perfilador para o mesmo programa acima aperfeiçoado para evitar o cache-miss:

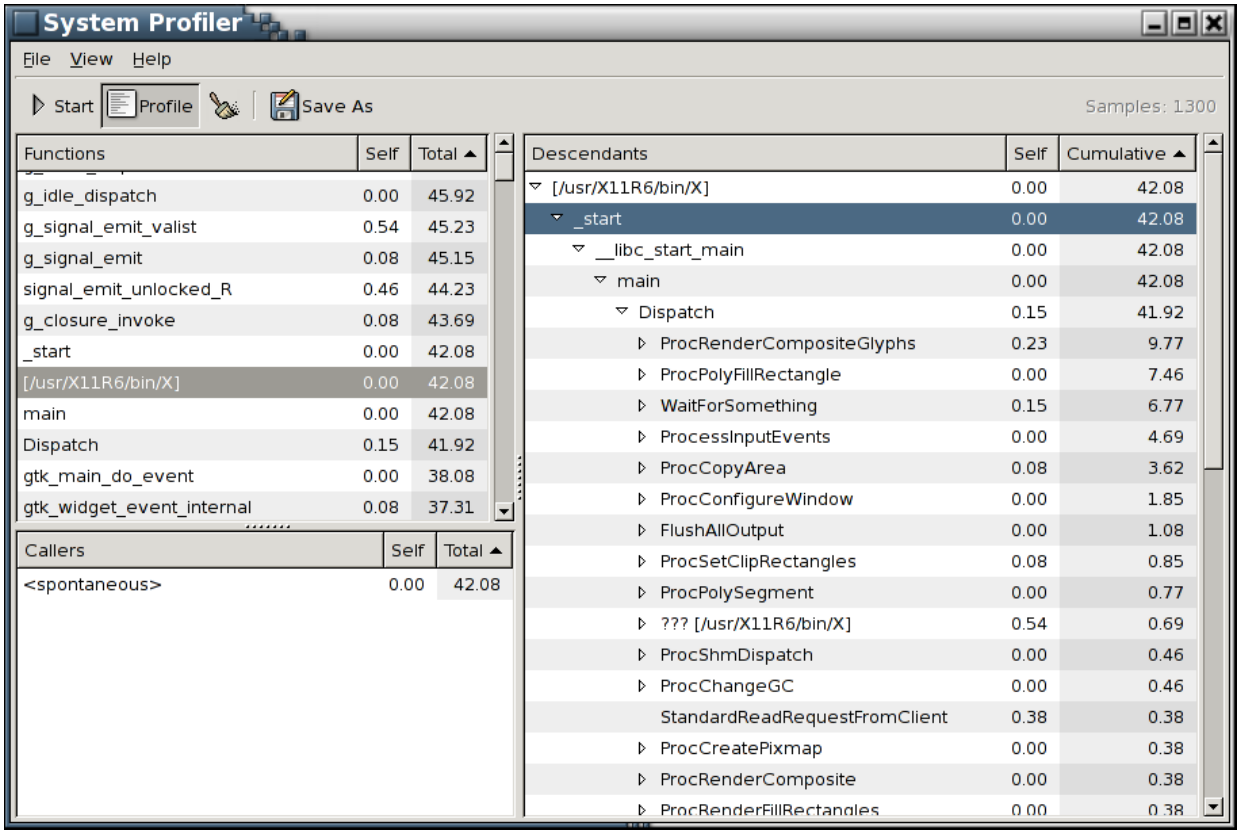
```
# opreport -l ./cache-miss
CPU: ppc64 POWER5, speed 1656.38 MHz (estimated)
Counted CYCLES events (Processor cycles) with a unit mask of 0x00
(No unit mask) count 1000
samples  %        symbol name
104916   58.3872  inc_second
74774    41.6128  main
```

Note que a melhora no mesmo programa, apenas evitando os cache-misses, neste caso é da ordem de 21% o que de certa forma nos fornece um limite superior sobre o quanto podemos melhorar um programa apenas evitando os cache-misses.

iii. Sysprof (http://www.daimi.au.dk/~sandmann/sysprof/)

O Sysprof, assim como o GProf, é um perfilador das aplicações em execução na máquina.

Para cada chamada da pilha de uma aplicação ele é capaz de fornecer informações sobre o tempo de execução. Ao contrário do GProf, o Sysprof apresenta informações, apesar de não tão detalhadas como as do GProf, de todos os processos do sistema de uma maneira gráfica. Abaixo segue uma impressão da tela do Sysprof com algumas informações dos processos que estão executando.



iv. Stat Monitor (http://sourceforge.net/projects/statmonitor)

Escrito completamente em C, faz o perfilamento das aplicações em execução salvando os seus resultados em arquivos XML de fácil leitura para humanos e máquinas. Oferece informações sobre a utilizacão da CPU, da utilização da memória, utilização da rede e do disco. As informações apresentadas são específicas por aplicação.

Apesar de ser bem simples, essa ferramenta é útil para se ter uma idéia geral do comportamento da aplicação. Uma saída de exemplo é mostrada abaixo:

```
<
      Stats pname="(firefox)"
      pid="5385"
      time="3085.9"
      usertime="2.3"
      systemtime="4.1"
      procuserload="2.4"
      maxprocuserload="2.4"
      procsysload="4.2"
      masprocsysload="4.2"
      userload="4.8"
```

```
      maxuserload="4.8"
      systemload="8.0"
      maxsystemload="8.0"
      vmem="0"
      maxvmem="3217043720"
      rmem="0"
      bytesin="128029"
      maxbytesin="128029"
      bytesout="51387"
      maxbytesout="51387"
      disk_throughput_read="0"
      disk_throughput_write="0"
      disk_bytes_read="0"
      disk_bytes_write="0"
/>
```

v. strace e ltrace

Já disponíveis diretamente a partir da maioria das distribuições linux, essas duas aplicações são úteis para entender quais tipos de operações uma aplicação em específico está utilizando.

A primeira destas ferramentas, o strace (provavelmente o nome vem de system trace) lista todas as chamadas feitas ao sistema (syscalls) para uma determinada aplicação. É interessante inclusive utilizá-la em uma chamada tão ingênua como 'cat /proc/version' para entender a sua serventia. A saída de tal chamada é listada abaixo. Algumas partes foram omitidas em prol da clareza:

```
execve("/bin/cat", ["cat", "/proc/version"], [/* 37 vars */]) = 0
brk(0)                                  = 0x849a000
open("/proc/version", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFREG|0444, st_size=0, ...}) = 0
read(3, "Linux version 2.6.27-7-generic (..., 1024) = 130
write(1, "Linux version 2.6.27-7-generic (..., 130) = 130
read(3, "", 1024)                       = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
Process 5723 detached
```

A saída do ltrace é um pouco diferente, mas em suma traz o mesmo tipo de informações sobre a aplicação em questão. A diferença é, no entanto, que o ltrace traz as listagens das chamadas às bibliotecas dinâmicas além das chamadas ao sistema. Uma saida do ltrace para uma chamada idêntica à acima pode ser vista abaixo (novamente algumas linhas foram retiradas em prol da clareza e do espaço):

```
__fxstat64(3, 1, 0xbfeefbbc)                    = 0
open64("/proc/version", 0, 027773575674)        = 3
__fxstat64(3, 3, 0xbfeefbbc)                    = 0
```

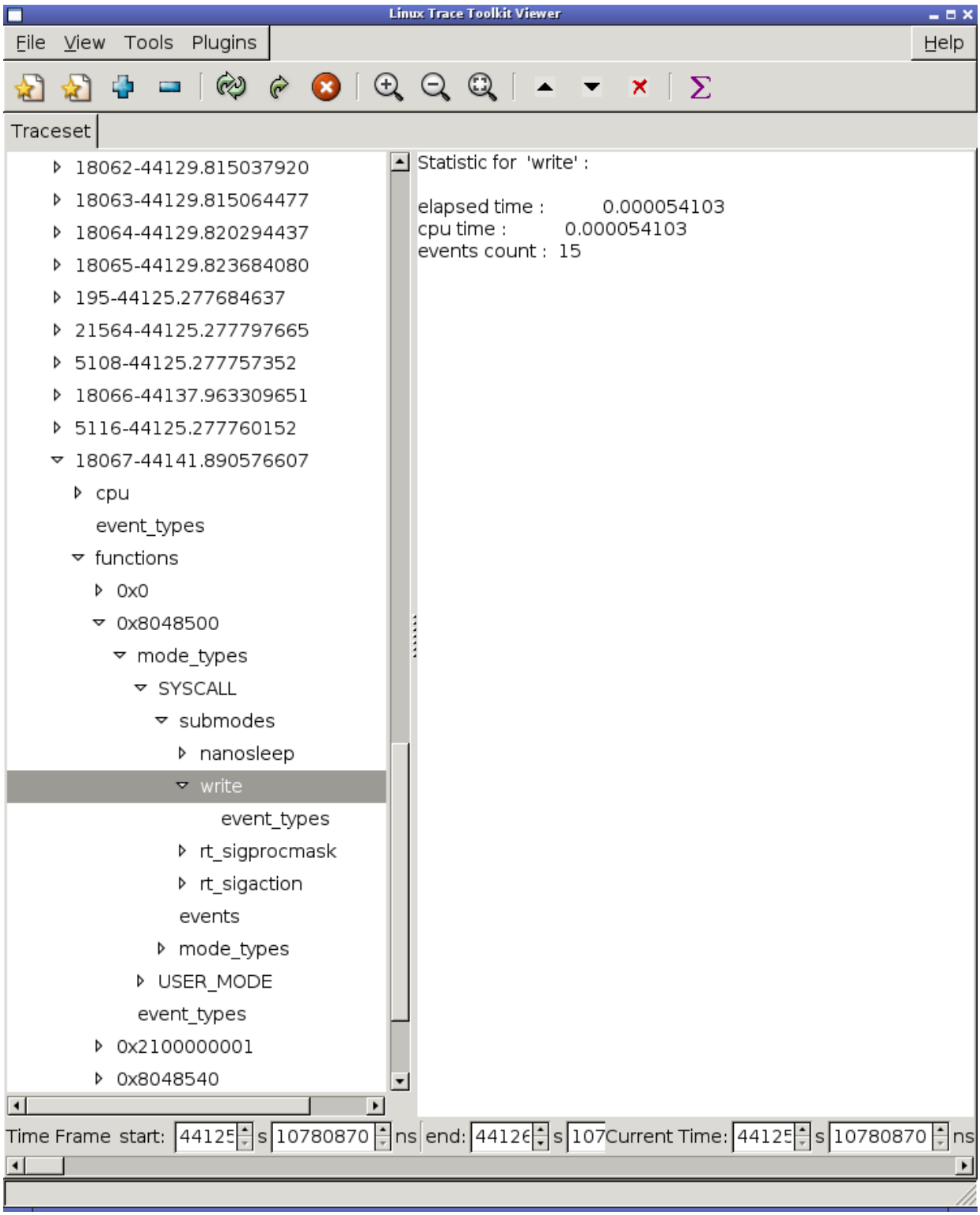
```
malloc(5119)                                        = 0x925d0d0
read(3, "Linux version 2.6.27-7-generic (..., 1024) = 130
write(1, "Linux version 2.6.27-7-generic (..., 130) = 130
read(3, "", 1024)                                   = 0
free(0x925d0d0)                                     = <void>
close(3)                                            = 0
exit(0 <unfinished ...>
__fpending(0xb80c34c0, 0xb80f1ff4, 0x8048590, 0xb80c2ff4, 0) = 0
fclose(0xb80c34c0)                                  = 0
__fpending(0xb80c3560, 0xb80f1ff4, 0x8048590, 0xb80c2ff4, 0) = 0
fclose(0xb80c3560)                                  = 0
+++ exited (status 0) +++
```

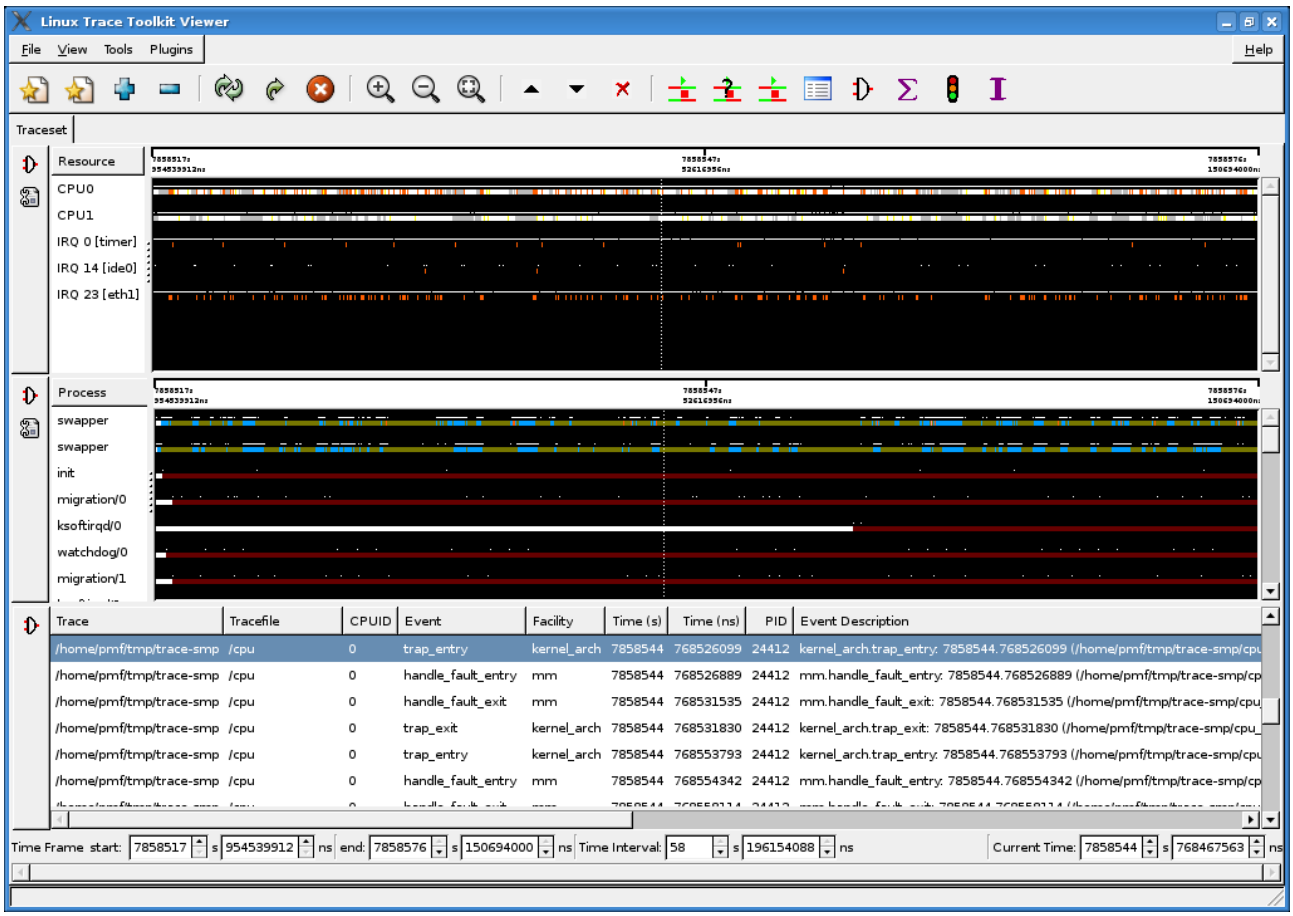
vi. LTTng (http://ltt.polymtl.ca/)

Ao contrário das ferramentas anteriormente apresentadas, o LTTng (Linux Trace Toolkit next generation) permite que cada aplicação crie e dispare eventos que serão avaliados pela ferramenta de perfilamento. Isso permite uma avaliação da execução da aplicação muito mais detalhada já que o desenvolvedor é capaz de definir pontos específicos de interesse. Como o seu próprio nome sugere, o LTTng é uma ferramenta de rastreamento que gera um arquivo com informações analíticas sobre todos os eventos registrados. Este arquivo é normalmente enorme, e a sua avaliação, apesar de poder ser feita manualmente ou através de alguma aplicação feita sob medida com este intuito, pode ser feita através de uma interface GUI oferecida pelo LTTng. Essa interface gráfica é muito rica e tem como principal vantagem ser capaz de separar as informações por aplicação – as outras ferramentas não são capazes de fazer este tipo de separação de uma maneira fácil ou direta. Algumas vantagens, então, ficam evidentes como por exemplo o perfilamento de aplicações que rodam em máquinas que tem muitos processos executando ao mesmo tempo.

Pela facilidade que o LTTng oferece ele foi escolhido para efetuar o perfilamento de algumas aplicações, inclusive algumas escritas em Java com eventos criados especificamente para tal aplicação. Foi utilizado por [1] e [2], onde este fez inclusive melhoramentos para poder obter informações dos contadores de hardware (de forma a conseguir informações sobre cache-misses e TLB-misses).

Mostramos abaixo algumas fotografias dessa interface gráfica.


Linux Trace Toolkit Viewer
File View Tools Plugins
Help
Traceset
18062-44129.815037920
18063-44129.815064477
18064-44129.820294437
18065-44129.823684080
195-44125.277684637
21564-44125.277797665
5108-44125.277757352
18066-44137.963309651
5116-44125.277760152
18067-44141.890576607
cpu
event_types
functions
0x0
0x8048500
mode_types
SYSCALL
submodes
nanosleep
write
event_types
rt_sigprocmask
rt_sigaction
events
mode_types
USER_MODE
event_types
0x2100000001
0x8048540
Statistic for 'write' :
elapsed time : 0.000054103
cpu time : 0.000054103
events count : 15
Time Frame start: 44125 s 10780870 ns end: 44126 s 107 Current Time: 44125 s 10780870 ns



3. Princípios de um escalonador

O escalonador que propomos deve levar em consideração as informações obtidas pelas ferramentas de perfilamento e escolher qual é a melhor distribuição dos processos e páginas de memória para cada um dos processos a ser executado.

Uma das idéias principais para um escalonador deste tipo é a de pares padrão de comportamento/escalonamento mais apropriado. O raciocínio por trás disto é o de que existem aplicações que são famintas por acesso a disco, outras por CPU e outras ainda por banda de memória. De fato existem aplicações cujo perfil não se encaixa nestes listados ou ainda cujo perfil é uma mistura deles. Ainda assim, o que pretendemos é ter as características gerais de como a aplicação se comporta para poder tomar uma decisão mais acertada sobre o escalonamento. Tendo uma idéia geral do comportamento da aplicação podemos gerar algumas regras simples, que poderiam eventualmente ser calculadas por uma máquina de regras, para gerar um escalonamento. Para exemplificar, uma tal regra das mais diretas e simples é a de que é interessante evitar, se possível, escalonar dois processos famintos por acesso acesso a memória no mesmo processador ou em em processadores que compartilhem algum nível de cache para evitar cache misses. Alguns pesquisadores adotam procedimentos um pouco mais diretos como [3] e [4]. No entanto, apesar da intenção ser facilitar, as aplicações reais podem não ser tão simples assim. Os processos podem ser famintos por banda de acesso a memória, mas é possível que eles trabalhem em conjunto e que compartilhem alguma estrutura de dados para escrita. Neste caso, se o escalonador decidir colocá-los em processadores que não compartilham nenhum nível de cache, o desempenho pode inclusive ser pior do que o se tivessem sido colocados no mesmo processador.

Se temos este tipo de problema, como decidir qual é melhor solução? Primeiramente é bom deixar claro que não temos a pretensão de resolver o problema para gerar a solução ótima, e isso tem alguns motivos, esses sim, bem simples. Via de regra os problemas de escalonamento da vida

real não tem todos os dados sobre as necessidades da aplicação disponíveis de antemão, ou seja, os escalonadores para este tipo de aplicação tem que tomar as melhores decisões informadas com posse das informações que têm até o momento, em outras palavras devem ser online. Escalonadores online devem executar rapidamente de modo a não causar atrasos, ou pelo menos causando atrasos negligenciáveis às aplicações.

As ferramentas de perfilamento de hardware nos informam sobre os limites do hardware e sobre o que ele é capaz de fazer, já as ferramentas de perfilamento de software nos informam sobre o comportamento do software além dos recursos que ele necessita para efetuar o seu trabalho. Parece razoável que, sabendo os limites e necessidades de hardware e software, sejamos capazes de escolher uma distribuição dos processos pelos processadores de forma a obter uma execução cujo desempenho seja melhor do que o de simplesmente distribuir os processos a esmo pelos processadores.

Essa é justamente a intenção nas próximas fases deste trabalho: auxiliado pelas ferramentas de perfilamento mostradas acima e por aplicações de testes especificamente criadas para se comportar de uma determinada maneira, pretendemos ser capazes de identificar automaticamente o perfil de uma aplicação específica e sugerir um escalonamento que possa trazer ganhos de desempenho para sua execução.

4. Aplicações de teste

A princípio, conforme está descrito no cronograma, o objetivo deste trabalho era lidar com diversas aplicações, efetuar o seu perfilamento e tentar classificá-las conforme o seu comportamento. No entanto, o prazo estimado para fazer estas tarefas acabou se mostrando demasiadamente estrito. Por conta de dificuldades como problemas de instalação de ferramentas, incompatibilidades de versão de kernel e afins, as tarefas que foram apresentadas e estimadas como sendo corriqueiras acabaram tomando muito mais tempo do que era esperado. Houve uma subestimação da complexidade do trabalho.

Por conta dessas dificuldades decidiu-se por escolher apenas uma aplicação de teste, mas que no entanto possuísse muitas das características desejadas como alto uso de CPU, alto uso de memória, possibilidade de execução em múltiplos threads e variações de comportamento (ou de perfil) entre uma execução e outra. Desta forma, através de uma aplicação de teste bem escolhida, poderíamos testar as diversas ferramentas e assim utilizar o tempo que seria gasto para a escolha de outras aplicações executando perfilamentos e analisando as ferramentas escolhidas.

Uma aplicação com todas essas características não é tão comum. Na busca, após descartarmos alguns candidatos como o BOINC [6] por ele não dispor de muita liberdade de configuração para as aplicações que nele executam, encontramos o GIMPS (Great Internet Mersenne Prime Search) [5]. O GIMPS é amplamente utilizado em benchmarks de CPU e memória, além de sua função original de encontrar por primos de Mersenne. Para evitar que pessoas utilizando este software apenas para fazer benchmarks comprometessem a busca, os seus criadores dotaram-no de diversas configurações muito interessantes. Tais configurações de execução incluem: testes de estresse de CPU, testes de estresse de memória, testes de estresse de CPU e memória em conjunto (causando propositadamente muitos cache-misses de L2) e configurações do número de threads a serem executados simultaneamente. Essas configurações de execução, no âmbito do GIMPS, são curiosamente chamadas de testes de tortura.

O GIMPS faz a fixação dos processos em cada um dos processadores através de uma política de round-robin. Desta forma, apenas caso ele tenha sido configurado para executar mais threads do que o número de processadores disponíveis é que um processador será associado a mais de um processo. O resultado obtido através do comportamento desta política é bem parecido com aquele do comportamento padrão do Linux que vai tentar, automaticamente, distribuir os processos entre os processadores com menos carga, fazendo inclusive a sua migração caso necessário.

Apesar do GIMPS possuir muitas das características desejáveis para uma aplicação de testes, ele está longe de ser completo. Ainda que ele teste à utilização de CPU e de memória, ele acaba não testando de forma direta a ocorrência de cache-misses e TLB-misses. Não testa também o acesso ao disco, ou seja, não exercita a capacidade de perfilamento no caso de aplicações que são famintas por IO. Este papel será desempenhado pelos próprios perfiladores do hardware: Stream e Bonnie. Como estes softwares foram escritos para justamente exigir o máximo destes recursos, são ótimos candidatos para perfilamento.

5. Trabalhos relacionados

Existem diversos trabalhos [1, 2, 4, 7, 8, 9, 10] relacionados, no contexto multi-core, ao perfilamento e à subsequente refixação de processos e páginas de memória. Alguns destes trabalhos utilizam mecanismos bem complexos e quiçá impraticáveis. Neste trabalho vamos, no entanto, nos limitar aos esquemas já utilizados no cotidiano dos desenvolvedores. O nosso intuito é o de podermos avaliar como os problemas estão sendo resolvidos hoje para que a ferramenta que propomos desenvolver seja capaz de atacar os problemas que estão surgindo na prática.

De uma forma geral podemos dividir as arquiteturas de computadores em dois grupos: SMP e não SMP tais como NUMA, clusters e grids. O objetivo principal deste trabalho é a arquitetura SMP, e é para ela que desejamos dar maior atenção. No entanto, explicaremos de forma sucinta como as demais arquiteturas estão tratando estes problemas para que se tenha uma visão do todo.

i. SMP

Os trabalhos feitos sobre esta arquitetura levam principalmente em consideração a colocação dos processos pelos diversos cores da máquina. Esta distribuição de processos a cores é feita, via de regra, através de uma API de fixação ("*pinning"*) de processos [19]. A idéia toda consiste, conforme discutimos na seção 3, na tentativa de se colocar os processos a serem executados nos cores disponíveis de forma que a interferência entre eles seja minimizada. Além desta estratégia, existem casos onde a interferência entre os processos pode ser tão significativa a ponto da execução seqüencial dos processos ser mais rápida do que a qualquer tentativa de executá-los concorrentemente [15].

Apesar de um sistema SMP ser bem mais estável, ou seja, sujeito a menores variações de hardware, do que um sistema NUMA, as variações entre cada sistema desses é enorme. Assim, muitas das otimizações que podem funcionar perfeitamente em uma máquina podem muito bem servir apenas para atrapalhar o desempenho da execução do mesmo programa em outra máquina semelhante.

As características acima são melhor discutidas na próxima seção (5.ii), onde, no contexto de máquinas NUMA, todas essas peculiaridades se tornam mais evidentes, além das soluções serem praticamente as mesmas.

ii. NUMA/Clusters

Existem diversas propostas disponíveis no mercado hoje. As soluções variam em muitos aspectos, algumas se preocupam diretamente com a alocação dos processos enquanto outras se preocupam mais com a localização das páginas de memória (principalmente no caso de sistemas com DSM). Algumas propostas deixam toda a decisão sobre a alocação das tarefas aos processadores à cargo da própria aplicação enquanto em outras é possível definir exatamente onde cada processo executará e onde cada página de memória será alocada. Não é a nossa intenção

discorrer aqui sobre todas as soluções existentes, mas apenas mostrar algumas das opções disponíveis. Para isso escolhemos duas maneiras bem distintas de se lidar com esse problema, o SLURM um escalonador para clusters Linux e a arquitetura NUMA, que apesar de não ser um escalonador, emprega muitas idéias interessantes para a definição da maneira como a alocação dos recursos será feita.

a) SLURM

O SLURM – *Simple Linux Utility for Resource Management* – [11, 12, 13] é um gerenciador de recursos de código fonte aberto. Ele foi desenvolvido para o gerenciamento de clusters Linux de tamanho arbitrário. O SLURM apresenta três funções principais, primeiramente ele é capaz de alocar recursos (de forma exclusiva ou não) aos usuários por determinados períodos de tempo de forma a serem capazes de executar o seu trabalho. Ele também provê um arcabouço para que os usuários do cluster possam iniciar, executar e monitorar a execução de suas tarefas. E finalmente, mas não menos importante, ele gerencia uma fila de requisições pendentes no caso de requisições conflitantes.

As versões do SLURM anteriores à 1.2 enxergavam cada um dos cores que compunham o cluster como processadores distintos.

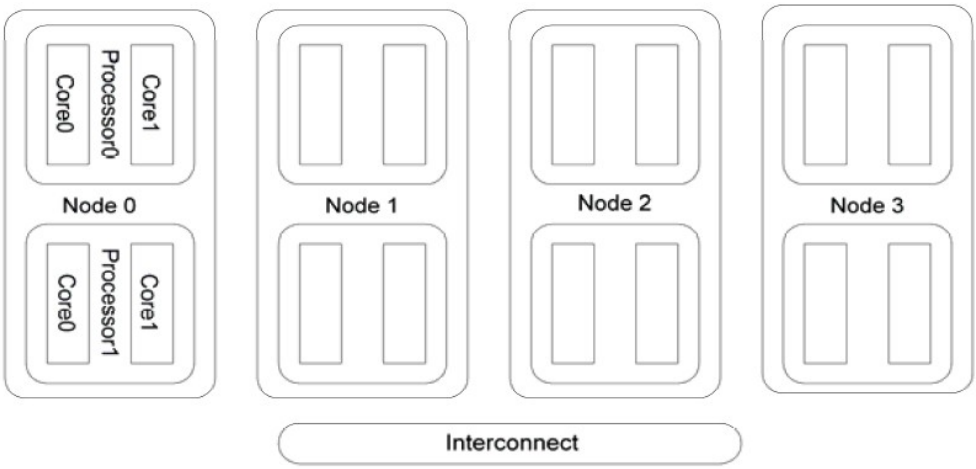


Figura 4: Cluster de quatro nós, com dois processadores com dois cores cada (extraído de[13])

Isso significava, que em um cluster, como o da figura 4, uma aplicação com 8 tarefas utilizando-se o algoritmo de escalonamento padrão resultaria na alocação das tarefas da seguinte maneira: tarefa 0, 1, 2 e 3 no nó 0 e tarefas 4, 5, 6 e 7 no nó 1. Isso em aplicações que exigem muito acesso à memória é o que se menos deseja, pois cada um dos cores do mesmo processador acabam competindo por recursos de acesso a memória o que pode causar problemas de contenção. Cientes deste problema, a partir da versão 1.2, os desenvolvedores do SLURM colocaram a disposição algumas funcionalidades para lidar com os processadores multi-core.

A partir da versão 2.6 do kernel do Linux tornou-se possível fixar (*to pin*), através de chamadas da API do sistema operacional, a localização da execução dos processos a processadores específicos. As interfaces do sistema operacional são de simples utilização: para cara processo que se deseja fixar deve-se fornecer uma mascara de bits, onde o bit *i* indica se o processo pode ou não rodar no processador *i*. Essa abordagem apesar de direta não é de fácil utilização. O esquema de numeração dos cores e processadores de uma máquina varia de acordo com o fabricante da placa

mãe, do processador, do fabricante do BIOS e até mesmo de uma versão do BIOS para outra do mesmo fabricante.

Para evitar esses problemas e fazer o uso desses recursos de fixação de execução mais acessíveis aos usuários não tão experientes, os projetistas do SLURM decidiram utilizar uma abordagem de um nível mais alto, no entanto ainda permitindo que os usuários mais experientes pudessem controlar a utilização de recursos com comandos de nível mais baixo caso desejassem. Para utilizar essa nova abordagem, o usuário, no momento do despacho do processo para a execução, passa a especificar alguns parâmetros que definem de uma maneira simples como ele quer que o seu processo seja executado. Esses parâmetros mais simples, que detalhamos abaixo, são na verdade traduzidos automaticamente para uma máscara de bits que é utilizada para a fixação de cada uma das tarefas do processo em cores específicos.

Baixo nível – fixação explícita	
--cpu_bind=map_cpu<list>	Especifica uma fixação pelo ID da CPU para cada tarefa
--cpu_bind=mask_cpu<list>	Expecifica uma fixação através de uma máscara pelo ID da CPU para cada tarefa
--cpu_bind=rank	Faz a fixação pelo rank da tarefa
Alto nível – geração automática da máscara	
--sockets-per-node=*S*	Número de soquetes por nó para dedicar a uma tarefa (pode ser especificado como uma faixa de valores)
--cores-per-socket=*C*	Número de cores por nó para dedicar a uma tarefa (pode ser especificado como uma faixa de valores)
--threads-per-core=*T*	Número de threads por nó para dedicar a uma tarefa (pode ser especificado como uma faixa de valores)
-B *S[:C[:T]]*	Opção reduzida para especificar ao mesmo tempo --sockets-per-node, --cores-per-socket, --threads-per-core
Novas restrições de alocação	
--minsockets=*MinS*	Os nós precisam ter no mínimo esse número de soquetes
--mincores=*MinC*	Os nós precisam ter no mínimo esse número de cores por soquete
--minthreads=*MinT*	Os nós precisam ter no mínimo esse número de threads por core
Controle de criação de tarefas	
--ntasks-per-node=*ntasks*	Número de tarefas a criar em cada nó
--ntasks-per-socket=*ntasks*	Número de tarefas a criar em cada soquete
--ntasks-per-core=*ntasks*	Número de tarefas a criar em cada core
Dicas sobre o perfil da aplicação	

Tabela 1: Parâmetros disponíveis a partir da versão 1.2 do SLURM para o controle da alocação dos processos aos processadores.

Como exemplo, as seguintes linhas de comando são equivalentes em um cluster com 4 nós, 4 soquetes por nó e 2 cores por soquete para a execução de um processo MPI com 32 tarefas.

Parâmetros de alto nível
#mpirun -srun -n 32 -N 4 – B 1:1 a.out
Parâmetros de baixo nível
Com a numeração dos cores por blocos
#mpirun -srun -n 32 –N 4 -cpu_bind=mask_cpu:1,4,10,40,2,8,20,80 a.out
ou
#mpirun -srun -n 32 –N 4 -cpu_bind=map_cpu:0,2,4,6,1,3,5,7 a.out

Tabela 2: Exemplos de parâmetros para a execução de processos com o SLURM

Como pode-se perceber, a utilização dos parâmetros de alto nível simplifica muito a tarefa do desenvolvedor da aplicação que passa a não mais ter que explicitar exatamente onde cara tarefa do seu processo deverá ser executada.

Não discutiremos aqui os ganhos de desempenho que a fixação das tarefas a serem executadas aos processadores podem trazer, isso pode ser visto em [13, 14, 15, 16]. Por hora basta notar que a evitação da migração dos processos pode economizar tempo, tanto pelo tempo necessário pela migração em si, como pela perda dos caches dos processadores que ocorre devido a migração.

b) NUMA

NUMA é um acrônimo para *Non-Uniform Memory Access*. Esse termo descreve computadores cuja arquitetura faz com que o acesso a diferentes páginas de memória tenham tempos distintos para cada um dos processadores, daí o nome de "não uniforme". Um cluster pode se comportar como uma máquina NUMA, por exemplo, se estiver utilizando o Linux com suporte NUMA instalado. Nesses casos, para o usuário, o cluster aparenta ser uma grande máquina SMP quando na verdade o que ocorre é o uso intenso de DSM controlado pelo sistema operacional.

De fato, as máquinas NUMA por si só não se enquadram na categorias de escalonadores para clusters multi-core, tópico principal deste trabalho. No entanto, as maneiras pelas quais os programadores têm utilizado essas arquiteturas são. Esses usuários desenvolveram técnicas para forçar o escalonamento das suas tarefas nós cores que desejam, e são essas as técnicas que nos interessam.

Assim como no caso do SLURM, o principal motivo de se querer controlar a alocação dos recursos às tarefas é o ganho de desempenho. Ao contrário do SLURM, no caso das aplicações que rodam sobre arquiteturas NUMA, a comunicação entre os processos, via de regra, não é feita através de passagens de mensagens, mas sim através de memória compartilhada. Os tempos de comunicação na arquitetura NUMA são definidos pelo tempo de acesso a memória, que podem variar muito dependendo se a página desejada se encontra no mesmo nó do processo que a requisita ou em outro nó da rede. À razão entre o tempo de acesso a memória remota e a memória local dá-se o nome fator-NUMA. Esse valor pode variar muito conforme a rede de interconexão dos nós. Existem máquinas onde esse fator é tão baixo como 1,2 [17], mas claramente esse valor pode ser muito maior.

Ao contrário das aplicações mais tradicionais para clusters, que são desenvolvidas com o MPI ou APIs de troca de mensagem semelhantes, as aplicações para arquiteturas NUMA são geralmente criadas com pthreads ou OpenMP. Isso porque, na verdade, a ilusão que uma arquitetura NUMA quer fornecer aos seus utilizadores é a de que há uma só máquina e a sua programação, à princípio, poderia ser idêntica a uma máquina SMP multi-core. Isso, no entanto, não é bem verdade

e quanto maior for o fator NUMA da máquina em questão mais esse problema fica evidenciado. Isso é agravado pelo fato que que pthreads e OpenMP foram originalmente desenvolvidos para máquinas UMA e não dispõem de mecanismos para o suporte de máquinas NUMA [17]. Doravante nós nos referiremos, sem perda de generalidade, às aplicações multi-thread como aplicações OpenMP, sejam elas realmente aplicações OpenMP ou pthreads.

Em uma máquina com o fator NUMA muito próximo de 1, os seus utilizadores não teriam problemas, já que ela seria praticamente idêntica a uma máquina UMA. Infelizmente esse não é o caso no mundo real. Justamente por esses motivos sistemas operacionais com suporte a NUMA como o Linux e o Solaris oferecem ferramentas para especificar as políticas de escalonamento de memória e dos threads para uma dada aplicação. Essas ferramentas estão disponíveis no nível programático, através de APIs, ou ainda através de ferramentas de linha de comando. A vantagem desta é a de que ela não exige alteração alguma no código da aplicação, no entanto aquela permite a alteração em tempo de execução das estratégias sendo utilizadas para o escalonamento das páginas de memória e dos threads. Neste texto nos ateremos somente ao caso do sistema operacional Linux.

Desde a versão 2.6 do kernel do linux há o suporte para definição de afinidade de memória em arquiteturas NUMA. A política de alocação padrão utilizada no Linux é a de first-touch, ou seja, a página é alocada no nó onde o primeiro thread que fez acesso a ela estiver executando. A idéia dessa estratégia é deixar as páginas de memória próximas aos threads que a acessam.

Pode não ficar claro dentro de uma aplicação OpenMP a diferenciação entre o thread que inicializou uma estrutura de dados e o thread que utiliza essa estrutura de dados. Isso é natural dentro de aplicações OpenMP. Nesse caso, pode ocorrer de todas as estruturas ficarem no thread mestre que inicializou as estruturas de dados e criou os threads, por exemplo, e todos os demais threads acabarem tendo que fazer acesso remoto à memória do nó onde o thread mestre executa. Para evitar esse tipo de problema é comum que os desenvolvedores das aplicações utilizem uma técnica denominada parallel-init, onde cada thread inicializa a própria estrutura de dados de forma a deixar as páginas de memória que ele acessa próximas a ele.

A técnica do parallel-init funciona muito bem quando o padrão de acesso à memória é regular e a carga de trabalho de cada thread é definida estaticamente. Além disso não há garantia alguma que essa técnica vá funcionar. Para conseguir essas garantias é necessário utilizar as chamadas de API providas pelo sistema operacional. São duas as funções mais utilizadas para fazer esse trabalho, `mbind` e `sched_setaffinity` [18, 19]. Através da utilização controlada dessas funções é possível minimizar o impacto da arquitetura NUMA nas aplicações.

A função `mbind` permite definir como as páginas de memória deverão ser alocadas. Ela permite definir a política de alocação da memória, os nós onde a página (ou seqüência de páginas) será alocada e um flag que indica se a migração de páginas será permitida. As políticas de alocação permitidas são *bind*, onde a memória será alocada no conjunto específico de nós, *interleave*, onde a memória será alocada ciclicamente entre um conjunto de nós, sendo o primeiro aquele que provocou a falha de página, e *preferred* no qual as páginas serão alocadas preferencialmente no conjunto de nós especificados.

A função `sched_setaffinity` permite associar um thread a um processador/núcleo específico através do fornecimento de uma máscara de bits.

As vantagens de ganho de desempenho nesses casos é semelhante a aquelas obtidas através da especificação dos parâmetros de máscara de fixação para o SLURM e como naquele caso não vamos nos aprofundar neste assunto. Além dos trabalhos já citados na seção anterior, para maiores detalhes sobre os ganhos de desempenho que tais estratégias podem alcançar, pode-se consultar [17].

c) **Análise das propostas atuais**

As propostas apresentadas mostram de uma maneira geral o estado atual das soluções

existentes. Podemos dividí-las em duas categorias, a categoria de baixo nível e a categoria de alto nível.

Aquelas de baixo nível, presentes tanto no SLURM quanto na arquitetura NUMA, são de fato muito poderosas. Elas permitem um completo controle do utilizador quanto a localização de cada uma das tarefas a serem executadas. No caso NUMA ainda se tem a possibilidade de definir onde cada página da memória será alocada (o que não faz muito sentido no caso do SLURM já que no caso geral a comunicação é feita por troca de mensagens). Todo esse poder, no entanto, não é de fácil utilização. A definição de onde serão executados cada um dos processos é morosa e exige dos seus usuários não só o conhecimento da aplicação que desejam executar como também exige o conhecimento sobre as características de cada uma das máquinas que compõem o cluster e sobre topologia da sua rede de interconexão.

Já a versão de alto-nível que está disponível no SLURM é bem mais amigável. Nela não é necessário conhecer exatamente as características de cada um dos nós, mas ainda é necessário conhecer o comportamento da aplicação e esta precisa ser bem comportada, ou seja, aplicações cujo o padrão de processamento ou comunicação muda durante a execução ou entre as execuções não são bem atendidas por essa solução. Além disso em clusters heterogêneos, ou seja, naqueles onde os nós componentes não são todos idênticos e que portanto tem diferentes capacidades de processamento e desempenho, a utilização dos parâmetros de alto nível desconsidera as idiossincrasias de cada nó tratando-os todos como iguais.

Em ambos os casos pode ser necessário rescrever trechos, ou eventualmente praticamente toda a aplicação, quando se troca o cluster sobre o qual a aplicação está sendo executada para que a aplicação tenha o desempenho desejado. Ainda que a aplicação não precise ser rescrita, a correta configuração dos parâmetros para a execução, seja através de ferramentas de alto ou baixo nível, dificilmente é criada otimamente nas primeiras execuções. Isso ocorre porque aplicações simples o suficiente ou com padrões de comportamento bem regulares são raras. Freqüentemente a definição desses parâmetros exige várias rodadas de experimentação. Clusters heterogêneos adicionam um nível a mais de complexidade a essa tarefa.

6. Trabalhos futuros

A prioridade para se trabalhar, sem dúvida alguma, consiste na criação de um perfilador específico para nossas necessidades. Este perfilador provavelmente se aproveitará de muitas das partes dos vários perfiladores apresentados, mas também precisará muitas partes especificamente escritas para si. Este perfilador além de compreender as características das aplicações perfiladoras já citadas, deve também ser capaz de determinar para uma determinada aplicação em execução qual é o seu perfil. Se é um perfil que exige muitos acessos à memória, ou se é uma aplicação faminta por E/S ou ainda se é uma aplicação do tipo de uso intensivo de CPU. Esse perfilador, ao contrário dos perfiladores já existentes, deverá ter alguma inteligência para que seja mais do que um simples apresentador de resultados. Ele deverá ser capaz de sugerir mudanças que quando feitas devam melhorar o desempenho da execução do software. Por exemplo, uma saída desejável de tal perfilador seria fixar processos x e y nos cores a e b para que não concorram diretamente.

Feita essa primeira fase, o segundo passo seria utilizar este perfilador para que fosse parte integrante do escalonador. Primeiramente um escalonador no espaço do usuário interessado apenas na aplicação específica, e mais tarde, talvez, um escalonador no espaço do kernel de forma a ser aplicado em todos os processo em execução de uma determinada máquina. Este último passo deve ser feito de tal forma que o impacto do perfilador e do escalonador sejam negligenciáveis ao sistema como um todo, sob pena de não utilização devido à carga adicional que seria colocada sobre o sistema.

7. Conclusão

As máquinas atuais estão ficando cada vez mais poderosas, não apenas através do aumento do tamanho do cache ou do aumento da freqüência de funcionamento mas principalmente através do aumento do número de cores. Essa mudança de enfoque traz diversas vantagens para os projetistas de hardware como a diminuição do gasto de energia e a simplificação do design dos circuitos. É um novo alívio aos projetistas que agora podem gozar de mais alguns anos de aumento de desempenho relativamente tranqüilo no desenvolvimento de processadores cada vez mais rápidos. Enquanto esta mudança de enfoque traga uma certa tranqüilidade aos projetistas de hardware, ela coloca muito mais peso nas costas dos desenvolvedores de software. Antes um programador não necessitava saber muito além da estrutura básica de funcionamento do hardware para fazer programas eficientes. Além disto suas otimizações funcionavam, quase que em sua totalidade, para todos as máquinas onde sua aplicação poderia possivelmente ser executada. Isso já não é mais assim. As hierarquias de cache são apenas um dos muitos pontos diferenciados que precisam ser levados em consideração pelo desenvolvedor. Essa dificuldade fez com que recentemente houvesse um renascimento das linguagens de programação funcionais, pois com elas um desenvolvedor não precisa se preocupar com a programação concorrente em si, pois por construção tais programas não tem estado compartilhado entre as diversas linhas de execução. Ainda assim, mesmo com a utilização deste tipo de linguagem, o problema de utilizar eficientemente o hardware persiste, já que elas criarão diversos processos/threads para sua execução e acaba deixando à cargo do SO a alocação destes processos aos cores. Em suma, facilita a vida do desenvolvedor mas não resolve o problema da eficiência de uso do hardware.

Já existem ferramentas para se fazer bom uso dos processadores e dos recursos de hardware de uma máquina. Todas elas no entanto, excluindo-se aquelas presentes do próprio SO, são manuais e não portáveis. Não existe uma maneira simples de escrever um software e executá-lo nos mais diversos ambientes e fazer com que ele ofereça um desempenho ótimo.

Aqui propusemos um esquema para que, ainda que não consigamos um desempenho ótimo, sejamos capazes de utilizar o hardware para os programas concorrentes de uma maneira otimizada, melhor que a simples alocação dos processos aos processadores à esmo ou utilizando heurísticas como o menos carregado primeiro. Para sermos capazes deste objetivo propusemos a utilização de perfilamento de aplicações de forma que as atitudes de otimização que hoje são tomadas manualmente passassem a ser tomadas automaticamente, livrando o programador e o usuário de ter que fazer uma otimização para cara um dos sistemas onde o seu software irá rodar.

8. Referências

[1] Resource Overbooking and Application Profiling in Shared Hosting Platforms, Bhuvan Urgaonkar, Prashant Shenoy and Timothy Roscoe, ACM Transations on Internet Technologies (TOIT), vol 9, number 1, Pages 1-45, February 2009

[2] Robert W. Wisniewski, Reza Azimi, Mathieu Desnoyers, Maged M. Michael, Jose Moreira, Doron Shiloach, and Livio Soares**,** Experiences Understanding Performance in a Commercial Scale-Out Environment**,** *In European Conference on Parallel Computing (*Euro-Par 2007)

[3] Managing Shared L2 Caches on Multicore Systems in Software, David Tam, Reza Azimi, Livio Soares, and Michael Stumm *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA 2007)*

[4] S. Kim, D. Chandra, and Y. Solihin, “Fair cache sharing and partitioning in a chip

multiprocessor architecture" in *PACT*, 2004.

[5] GIMPS, Great Internet Mersenne Prime Search - http://www.mersenne.org/

[6] BOINC, Berkeley Open Infrastructure for Network Computing - http://boinc.berkeley.edu/

[7] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandervoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? In Proceedings of the 16th ACM Symposium of Operating Systems Principles (SOSP), Saint-Malo, France, Oct. 1997.

[8] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In ICS In- ternational Conference on Supercomputing, Cambridge, Massachusetts, June 2005

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *HPCA*, 2005.

[10] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *EuroSys*, 2007.

[11] SLURM Home page - https://computing.llnl.gov/linux/slurm/

[12] SLURM: Simple Linux Utility for Resource Management, A. Yoo, M. Jette, and M. Grondona, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44-60, Springer-Verlag, 2003.

[13] Susanne M. Balle Daniel J. Palermo. Enhancing an Open Source Resource Manager with Multi-core/Multi-threaded Support, Job Scheduling Strategies for Parallel Processing, JSSPP (2007), 37-50

[14] M. Snir and J. Yu. On the theory of spatial and temporal locality. Technical Report DCS-R-2005-2564, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, July 2005

[15] Dave Field, Deron Johnson, Don Mize and Robert Stober, Scheduling to Overcome the Multi-Core Memory Bandwidth Bottleneck, Hewlett-Packard Development Company, technical report, November 2007

[16] CASTRO, M., Fernandes, L. G., Pousa, C. R., Méhaut, J-F., Dupros, F., Aguiar, M. S., "NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines". In: Workshop on Parallel and Distributed Scientific and Engineering Computing, IPDPS, 2009.

[17] POUSA, C. R., Martin-Marangozova, V., Méhaut, J-F., Dupros, F., Carissimi, A., "Explorando Afinidade de Memória em Arquiteturas NUMA". In: Workshop em Sistemas Computacionais de Alto Desempenho., 2008.

[18] A NUMA API for LINUX. Technical report, Novell, Abril 2005

[19] R. Love. Kernel Korner: CPU affinity. Linux Journal, 2003(111):8, 2003.