Programação paralela com o OpenMP

Em paralelo

O OpenMP traz o poder do multiprocessamento aos programas em C, C++ e Fortran.

por Wolfgang Dautermann

Marcus Hannerstig– www.sxc.hu

A nova safra de processadores com múltiplos núcleos está levando o mercado de desktops a um patamar de desempenho antes alcançável apenas por workstations e servidores. Da mesma forma, aqueles servidores quebra-galho que ficam na sala de servidores mas são alojados em gabinetes de PCs comuns também se beneficiam da multiplicação dos núcleos. No entanto, simplesmente ter um sistema multiprocessado não significa que todos os núcleos estejam sendo usados em todo o seu poder.

Na realidade, freqüentemente apenas um dos processadores está ocupado. A **figura 1** mostra a saída do *top* durante a execução do software *Xaos* de cálculos fractais. O programa parece usar 100% da CPU, mas a taxa real de uso é de apenas 60%.

Pressionar a tecla **[1]** lista as CPUs separadamente. Nesse modo (**figura 2**), é fácil conferir a carga de cada núcleo: uma CPU está 90% ocupada, enquanto a outra está descansando (carga de 0,3%).

O Linux recebeu suporte a sistemas multiprocessados há muitas luas, e há tempos as distribuições já instalam o kernel SMP por padrão. Portanto, o Linux já tem o que é preciso para usar o poder de múltiplos núcleos. Mas e o software?

Um programa em execução no sistema precisa estar ciente da arquitetura multiprocessada para desfrutar dos benefícios de desempenho. O *OpenMP* é uma especificação de API para "...paralelismo multi-thread com memória compartilhada" [1]. A especificação do OpenMP define um conjunto de diretivas do compilador, rotinas de bibliotecas e variáveis de ambiente para suportar ambientes multiprocessados.

Programadores C/C++ e *Fortran* podem usar o OpenMP com o intuito de criar novos programas apropriados para sistemas multiprocessados, e também de converter programas já existentes para rodarem com maior eficiência em ambientes multiprocessados.

```
top - 16:05:28 up 8 days,  1:58,  5 users,  load average: 1.01, 0.63, 0.27
Tasks: 122 total,   3 running, 119 sleeping,   0 stopped,   0 zombie
Cpu(s): 43.9%us, 12.6%sy,  0.0%ni, 41.4%id,  0.0%wa,  0.0%hi,  2.0%si,  0.0%st
Mem:   8174384k total,  4981140k used,  3193244k free,      744k buffers
Swap: 19543032k total,        0k used, 19543032k free,  3652792k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26082 daute     25   0 94384  53m 4644 R  100  0.7   2:17.14 xaos.bin
31941 root      15   0  117m  43m 6916 S   13  0.5   0:44.09 Xorg
    1 root      18   0   804  304  244 S    0  0.0   0:02.61 init
    2 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/0
    4 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    5 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/1
    6 root      10  -5     0    0    0 S    0  0.0   0:00.18 events/0
    7 root      10  -5     0    0    0 S    0  0.0   0:00.17 events/1
    8 root      12  -5     0    0    0 S    0  0.0   0:00.00 khelper
    9 root      10  -5     0    0    0 S    0  0.0   0:00.00 kthread
   14 root      10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/0
   15 root      10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/1
   16 root      14  -5     0    0    0 S    0  0.0   0:00.00 kacpid
   17 root      14  -5     0    0    0 S    0  0.0   0:00.00 kacpi_notify
  116 root      11  -5     0    0    0 S    0  0.0   0:00.00 cqueue/0
  117 root      11  -5     0    0    0 S    0  0.0   0:00.00 cqueue/1
```

Figura 1 Por padrão, o *top* exibe a carga total das CPUs...

Multi-pista

A execução de programas de forma serial utiliza apenas um núcleo. A paralelização permite um uso mais eficiente do sistema multiprocessado.

A interface de programação do OpenMP, em constante desenvolvimento por vários fabricantes de hardware e compiladores desde 1997, oferece uma opção bem simples e portável para paralelizar programas escritos em C/C++ e Fortran.

O OpenMP é capaz de aumentar significativamente o desempenho dos programas, mas somente se a CPU realmente for exigida – e se a tarefa em questão for paralelizável, é claro. Esse costuma ser o caso em programas que dependem muito do processador.

Um, dois, vários

A API OpenMP oferece aos programadores uma opção simples para efetivamente paralelizar seus programas seriais pré-existentes por meio da especificação de algumas diretivas adicionais de compilação, que ficam semelhantes a esse código:

```
#pragma omp nome_da_diretiva
➥ [cláusulas]
```

Compiladores sem suporte ao OpenMP, como versões do GCC anteriores à 4.2, vão simplesmente ignorar as diretivas do compilador, o que significa que o código-fonte ainda pode ser compilado de forma serial:

```
$ gcc -Wall test.c
test.c: In function 'main':
test.c:12: warning: ignoring
➥ #pragma omp parallel
```

Códigos específicos para OpenMP também podem ser compilados de forma condicional, com a diretiva `#ifdef`: a OpenMP define a macro `_OPENMP` para isso.

Um programa com OpenMP é iniciado normalmente como se fosse serial, com uma única thread. A primeira declaração OpenMP apresentada cria múltiplas threads:

```
... uma thread
#pragma omp parallel
{ ... várias threads }
... uma thread
```

A **figura 3** mostra como o programa é distribuído por múltiplas threads e depois reunido numa única.

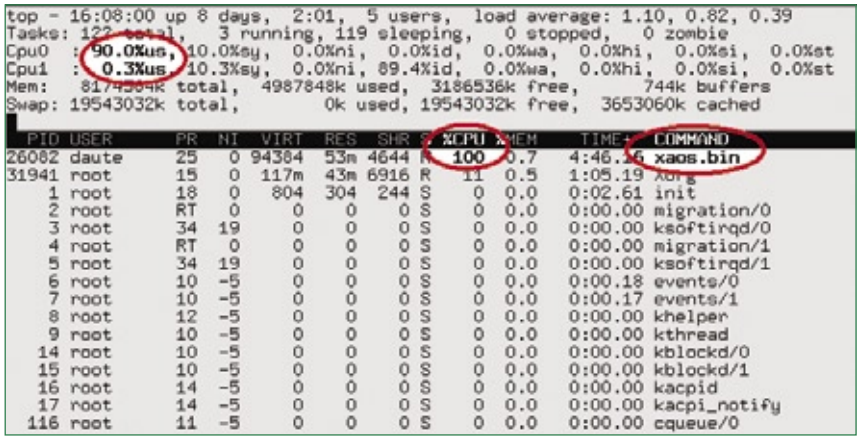

Figura 2 ...mas pode informar o valor de cada uma individualmente.

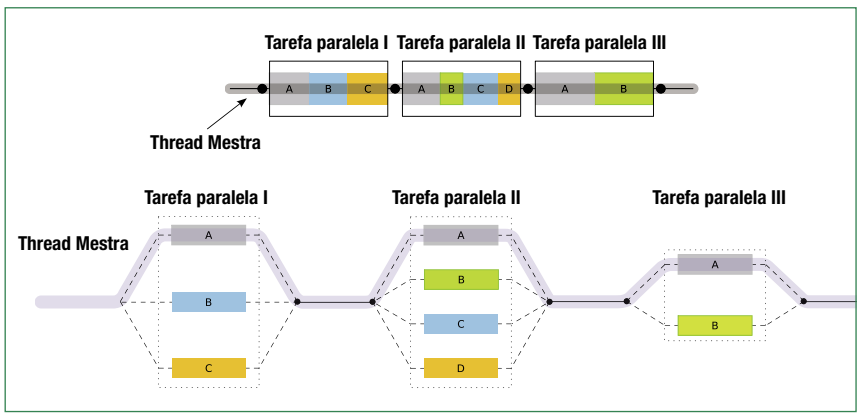

Figura 3 Método *Fork-Join* do OpenMP.

Dividir e conquistar

Agora você criou múltiplas threads, mas elas ainda fazem a mesma tarefa. A idéia é que as threads processem dados diferentes umas das outras. Em C, há duas formas para resolver isso. Em Fortran há ainda uma terceira: "compartilhamento de trabalho em paralelo".

Exemplo 1: Seções e loops paralelos

```
Variação 1: Seções paralelas
.../* uma thread */
#pragma omp parallel /* várias threads */
{
#pragma omp sections
#pragma omp section
... /* Seção A do programa rodando em paralelo com B e C */
#pragma omp section
... /* Seção B do programa rodando em paralelo com A e C */
#pragma omp section
... /* Seção C do programa rodando em paralelo com A e B */
}
... /* uma thread */


Variação 2: Loops paralelos
... /* uma thread */
#pragma omp parallel [cláusulas ...]
#pragma omp for [cláusulas ...]
for (i=0;i<N;i++) {
    a[i]= i*i; /* paralelizado */
    }
... /* uma thread */
```

A primeira variação, as seções paralelas, roda seções do programa (blocos de código do programa não interdependentes) que suportam a execução simultânea em paralelo.

Para isso ocorrer, `#pragma omp parallel` define múltiplas threads. Isso significa que é possível rodar múltiplos blocos de programa independentes em threads individuais sem restrições quanto ao número de *seções* paralelas (**exemplo 1, variação 1: Seções Paralelas**). Além disso, pode-se combinar as duas diretivas de compilação, `parallel` e `sections`, para formar uma única diretiva, como em `#pragma omp parallel sections`.

A segunda variação, loops `for()` paralelos, paraleliza loops, o que é especialmente útil no caso de programas matemáticos computacionalmente intensivos (**exemplo 1, variação 2: Loops Paralelos**).

Tabela 1: Cláusulas

`shared(lista_de_variáveis)`	Existe apenas uma versão das variáveis, e todas as seções paralelas do programa a acessam. Todas as threads têm acesso de leitura e escrita. Se uma thread alterar uma variável, isso também afetará as outras threads. Padrão: todas as variáveis são `shared()`, exceto as de loops em `#pragma omp for`.
`private(lista_de_variáveis)`	Cada thread possui uma cópia privada não inicializada da variável. Padrão: somente variáveis de loops são privadas.
`default(shared|private|none)`	Define o comportamento padrão das variáveis: `none` significa que é preciso declarar explicitamente cada variáveis como `shared()` ou `private()`.
`firstprivate(lista_de_variáveis)`	Semelhante a `private()`; porém, neste caso, todas as cópias são inicializadas com o valor da variável antes da região ou loop paralelos.
`lastprivate(lista_de_variáveis)`	A variável recebe o valor da última thread que a alterou em processamento seqüencial após o loop ou região paralelos terem sido terminados.

A **figura 4** mostra como isso funciona. Novamente é possível combinar `#pragma omp parallel` e `#pragma omp for` para formar `#pragma omp parallel for`.

Escopo

Na programação com memória compartilhada, múltiplas CPUs podem acessar as mesmas variáveis. Isso torna o programa mais eficiente e economiza cópias. Em alguns casos, cada thread precisa de sua própria cópia das variáveis – como as variáveis do loop no caso do `for()` paralelo.

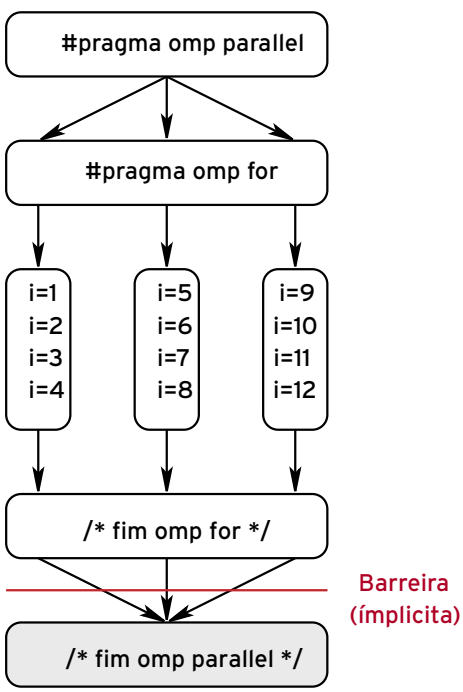

Figura 4 Loop `for()` paralelo.

As cláusulas especificadas nas diretivas OpenMP (veja as descrições na **tabela 1**) definem as propriedades dessas variáveis. Pode-se acrescentar cláusulas ao `#pragma`, por exemplo:

```
#pragma omp parallel
➥for shared(x, y) private(z)
```

Erros em declarações `shared()` e `private()` de variáveis são uma das causas mais comuns de erros na programação paralela.

Redução

Agora já mostramos como criar threads e distribuir o trabalho por múltiplas threads. Porém, como fazer todas as threads trabalharem num resultado único – por exemplo, para somar os valores de um vetor? A função `reduction()` (**exemplo 2**) cuida disso.

O compilador cria uma cópia local de cada variável de `reduction()` e inicializa-as de forma independente do operador (por exemplo, `0` para `+` e `1` para `*`). Caso múltiplas threads estejam cuidando, cada uma, de uma parte do loop, a thread mestre soma os três subtotais ao final.

Quem é mais rápido?

Depurar programas paralelos é uma forma de arte. É especialmente difícil encontrar erros que não ocorram em programas seriais e não ocorram regularmente no processamento paralelo. Essa categoria inclui o que se conhece como condições de corrida: resultados diferentes em execuções repetidas do programa com múltiplos blocos executados em paralelo, dependendo de qual thread é a mais rápida em cada execução. O código do **exemplo 3** começa preenchendo um vetor em paralelo e depois prossegue calculando a soma desses valores em paralelo.

Sem a declaração do OpenMP `#pragma omp critical(soma_total)` na **linha 13**, pode ocorrer a seguinte condição de corrida:

- a thread 1 carrega o valor atual de `soma` num registrador da CPU;
- a thread 2 carrega o valor atual de `soma` num registrador da CPU;
- a thread 2 adiciona `a[i+1]` ao valor no registrador;
- a thread 2 grava o valor do registrador de volta na variável `soma`;
- a thread 1 adiciona `a[i]` ao valor no registrador;
- a thread 1 grava o valor do registrador na variável `soma`.

Exemplo 2: `reduction()`

```
01 a=0; b=0;
02 #pragma omp parallel for
➥private(i) shared(x, y, n)
➥reduction(+:a, b)
03 for (i=0; i<n; i++) {
04      a = a + x[i] ;
05      b = b + y[i] ;
06      }
```

Exemplo 3: Evitar condições de corrida

```
01 #ifdef _OPENMP
02 #include <omp.h>
03 #endif
04 #include <stdio.h>
05 int main() {
06     double a[1000000];
07     int i;
08     #pragma omp parallel for
09     for (i=0; i<1000000; i++) a[i]=i;
10     double soma = 0;
11     #pragma omp parallel for shared (soma) private (i)
12     for (i=0; i<1000000; i++) {
13         #pragma omp critical (soma_total)
14         soma = soma + a[i];
15     }
16     printf("soma=%lf\n",soma);
17 }
```

Como a thread 2 ultrapassa a thread 1 nessa execução, ganhando assim a "corrida", `a[i+1]` não será calculado corretamente. Apesar da thread 2 calcular a soma e guardá-la na variável `soma`, a thread 1 a sobrescreve com um valor incorreto.

A declaração `#pragma omp critical` assegura que isso não ocorra. Todas as threads executam o código crítico, mas apenas um de cada vez. Com isso, o **exemplo 3** realiza a adição corretamente sem as threads paralelas atrapalharem o resultado. Para operações elementares (por exemplo, `i++`), `#pragma omp atomic` executará um comando de forma atômica. O acesso de escrita a variáveis `shared()` também é protegido quando se usa uma declaração `#pragma omp critical`.

Todos presentes?

Em alguns casos é necessário sincronizar todas as threads. A declaração `#pragma omp barrier` cria uma barreira virtual: todas as threads esperam até que a última delas alcance a barreira antes do processamento prosseguir. Mas é preciso pensar com cuidado antes de criar uma barreira artificial – fazer threads suspenderem o processamento diminuirá o ganho de desempenho obtido pelo uso do paralelismo. As threads que esperam não fazem nenhum trabalho. O **exemplo 4** ilustra um caso em que a barreira é inevitável.

A linha `Calcula()` desse exemplo calcula o segundo argumento com referência ao primeiro. Os argumentos nesse caso podem ser vetores, e a função de cálculo pode ser uma complexa operação matemática com matrizes. Aqui, é essencial usar `#pragma omp barrier` – qualquer falha na sincronia significaria que algumas threads começariam a segunda rodada de cálculos antes de os valores do cálculo em `B` ficarem disponíveis.

Alguns esquemas OpenMP (como `parallel`, `for`, `single`) incluem uma barreira implícita que pode ser explicitamente desativada com uma cláusula `nowait`, como em `#pragma omp for nowait`. Outros mecanismos de sincronização incluem:

- `#pragma omp master {código}`: código que será executado uma única vez e somente pela thread mestra;
- `#pragma omp single {código}`: código que será executado uma única vez, mas não necessariamente pela thread mestra;
- `#pragma omp flush (variáveis)`: variáveis em cache gravadas novamente na memória garantem uma visão consistente da memória.

Esses mecanismos de sincronização ajudarão a manter o código rodando tranqüilamente em ambientes multiprocessados.

Funções de biblioteca

A **tabela 2** lista algumas outras funções do OpenMP. Para usá-las, é preciso incluir o cabeçalho `omp.h` no código C/C++. Para garantir que o programa seja compilável também sem o OpenMP, é interessante acrescentar a linha `#ifdef _OPENMP` para a compilação condicional:

```
#ifdef _OPENMP
#include <omp.h>
threads = omp_get_num_threads();
#else
threads = 1
#endif
```

Funções de bloqueio (*locking*) permitem que uma thread bloqueie um recurso reservando acesso exclusivo (`omp_set_lock()`) a ele. Outras threads então podem usar `omp_test_lock()` para conferirem se o recurso está bloqueado. Essa configuração é útil caso se deseje que várias threads

Exemplo 4: Barreira inevitável

```
01 #pragma omp parallel shared (A, B, C)
02 {
03     Calcula(A,B);
04     printf("B foi calculado a partir de A\n");
05 #pragma omp barrier
06     Calcula(B,C);
07     printf("C foi calculado a partir de B\n")
```

Tabela 2: Funções do OpenMP

Função	Explicação
`int omp_get_num_threads()`	Retorna o número de threads.
`int omp_get_thread_num()`	Retorna o número da thread atual.
`void omp_set_num_threads(int)`	Define o número de threads a serem usadas em regiões paralelas futuras.
Funções de bloqueio	
`void omp_init_lock(omp_lock_t*)`	Inicializa uma trava (*lock*).
`void omp_set_lock(omp_lock_t*)`	Espera e depois define uma trava; bloqueia caso a trava não esteja disponível.
`int omp_test_lock(omp_lock_t*)`	Espera e depois define uma trava; não bloqueia se a trava não estiver disponível.
`void omp_unset_lock(omp_lock_t*)`	Remove uma trava.
`void omp_destroy_lock(omp_lock_t*)`	Destrói uma trava.

gravem dados num arquivo, mas seja preciso restringir acesso a uma thread por vez. Quando são usadas funções de bloqueio, é importante atentar para *deadlocks*.

Esse fenômeno ocorre quando threads precisam de recursos mas bloqueiam uma à outra. Por exemplo, quando a thread 1 bloqueia o recurso A e precisa usar o recurso B, enquanto a thread 2 faz o contrário, ambas as threads esperam para sempre.

Exemplo 5: Hello, world

```
01 /* helloworld.c (Versão OpenMP) */
02
03 #ifdef _OPENMP
04 #include <omp.h>
05 #endif
06 #include <stdio.h>
07 int main(void)
08 {
09     int i;
10 #pragma omp parallel for
11     for (i = 0; i < 4; ++i)
12     {
13         int id = omp_get_thread_num();
14         printf("Ola, mundo da thread %d\n", id);
15         if (id==0)
16             printf("Existem %d threads\n", omp_get_num_
➥threads());
17     }
18     return 0;
19 }
```

Variáveis de ambiente

Algumas variáveis de ambiente controlam o comportamento da execução dos programas OpenMP; a mais importante é `OMP_NUM_THREADS`. Ela especifica quantas threads conseguem operar em regiões paralelas, pois threads em excesso na realidade tornam o processamento mais lento. O comando `export OMP_NUM_THREADS=1` faz com que o programa rode com somente uma thread – exatamente como um programa serial normal.

Exemplo 6: Compilação do Hello world

```
$ gcc -Wall -fopenmp
➥helloworld.c
$ export OMP_NUM_THREADS=4
$ ./a.out
Ola mundo da thread 3
Ola mundo da thread 0
Ola mundo da thread 1
Ola mundo da thread 2
Existem 4 threads
```

Exemplo 7: Notificação

```
$ icc -openmp helloworld.c
helloworld.c(8): (col. 1)
remark:
OpenMP DEFINED LOOP WAS
PARALLELIZED.
```

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1} + \cdots$$

Figura 5 Fórmula de cálculo de pi de Gregor Leibniz.

Mãos à obra

Para usar o OpenMP, é preciso o conjunto de compiladores GCC na versão 4.2 ou posterior. Outros compiladores compatíveis são o da Sun [2], gratuito, e o da Intel [3], gratuito para uso não comercial.

O **exemplo 5** mostra uma versão em OpenMP do clássico *Hello World*. Para usar o OpenMP no GCC basta usar a opção `-fopenmp` na compilação. O **exemplo 6** mostra os comandos para compilar o programa juntamente com sua saída.

Se for usado o compilador da Sun, a opção é `-xopenmp`. Com o compi-

Exemplo 8: Cálculo de pi

```
01 /* pi-openmp.c (Versão OpenMP) */
02
03 #include <stdio.h>
04 #define STEPCOUNTER 1000000000
05 int main(int argc, char *argv[])
06   {
07   long i;
08   double pi = 0;
09   #pragma omp parallel for reduction(+: pi)
10   for (i = 0; i < STEPCOUNTER; i++) {
11     /* pi/4 = 1/1 - 1/3 + 1/5 - 1/7 + ...
12     Para evitar precisar mudar
13     continuamente o sinal (s=1; a cada
14     etapa s=s*-1), adicionamos dois
15     elementos ao mesmo tempo. */
16     pi += 1.0/(i*4.0 + 1.0);
17     pi -= 1.0/(i*4.0 + 3.0);
18   }
19   pi = pi * 4.0;
20   printf("Pi = %lf\n", pi);
21   return 0;
22 }
```

Exemplo 9: Pi paralelo

```
$ gcc -Wall -fopenmp -o
➥ pi-openmp
pi-openmp.c
$ export OMP_NUM_THREADS=1
$ time ./pi-openmp
Pi = 3.141593
real    0m31.435s
user    0m31.430s
sys     0m0.004s
$ export OMP_NUM_THREADS=2
$ time
./pi-openmp
Pi = 3.141593
real    0m15.792s
user    0m31.414s
sys     0m0.012s
```

lador da Intel, a opção é `-openmp`. O compilador da Intel até notifica o programador quando algo é paralelizado (**exemplo 7**).

Benefícios?

Para um exemplo de aumento de desempenho com o OpenMP, vejamos um teste que calcula pi [4] com uso da fórmula de Gregory Leibniz (**exemplo 8** e **figura 5**). Esse método não é, de forma alguma, o mais eficiente para calcular o valor de pi; porém, o objetivo não é eficiência, mas fazer as CPUs trabalharem.

Paralelizar o loop `for()` com o OpenMP de fato otimiza o desempenho (**exemplo 6**). O programa roda com o dobro da velocidade quando usa duas CPUs em vez de uma, pois praticamente todo o cálculo pode ser paralelizado.

Se o programa for monitorado com a ferramenta *top*, é possível verificar que as múltiplas CPUs realmente estão trabalhando e que o programa `pi-openmp` usa mesmo 200% da CPU.

Esse efeito tão positivo sobre a velocidade do programa não será sempre tão pronunciado para qualquer programa. Nesse caso, é possível retornar à execução serial de trechos do programa. Os resultados sempre seguirão a Lei de Amdahl [5] (veja o **quadro 1** para uma explicação). ■

Quadro 1: Lei da Amdahl

"Speedup" descreve o fator pelo qual um programa pode ser acelerado por paralelismo. Num caso ideal, a execução de um programa com *N* processadores levaria somente 1/*N* do tempo exigido por um programa serial. Esse caso ideal é conhecido como "speedup linear". No mundo real, o speedup linear costuma ser impossível de alcançar porque certas partes do programa não são particularmente apropriadas para a paralelização.

Dada a parte de um programa que suporte paralelismo, *P* (portanto, 1 – *P* é a parte não paralelizável) e o número de processadores disponíveis, *N*, o speedup máximo é calculado pela fórmula:

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

Se a parte serial do programa (1–P) for 1/4, o speedup não pode ser maior que 4. Não importa quantos processadores sejam usados.

Mais informações

[1] OpenMP: http://www.openmp.org

[2] Compilador da Sun: http://developers.sun.com/sunstudio

[3] Compilador da Intel: http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/

[4] Cálculo do Pi (Wikipédia): http://pt.wikipedia.org/wiki/Pi#M.C3.A9todos_de_c.C3.A1lculo

[5] Lei de Amdahl (Wikipédia): http://pt.wikipedia.org/wiki/Lei_de_Amdahl