EP I de MAC412

Ana Luísa Losnak (7240258)
Tiago Madeira (6920244)

15 de setembro de 2013

Sumário

1 Parte 1

A parte 1 deste EP visa a instalação da biblioteca PAPI[1] e a familiarização ao uso de suas funcionalidades, bem como a análise e discussão sobre a diferença no tempo de execução de programas similares.

Após a instalação completa, descrita em mais detalhes abaixo, alteramos o contador de tempo dos programas disponíveis no enunciado por eventos da PAPI para maior resolução no resultado. Com esses novos programas, realizamos diversas experiências, analisando o tempo de execução destes programas e discutindo hipóteses para tal diferença. Por fim, avaliamos qual hipótese era a mais correta utilizando a biblioteca PAPI para contar diversos eventos do computador ao executar cada programa. A seguir descrevemos mais detalhadamente como foi feita a parte 1.

1.1 Instalando o PAPI

Iniciamos nosso trabalho baixando a biblioteca PAPI, lendo seu arquivo `README.txt` e seguindo algumas instruções sugeridas pelo arquivo para melhor entendermos o funcionamento da PAPI. Nesse processo, instalamos a biblioteca, acessamos a documentação do projeto e realizamos alguns testes. O sistema operacional que utilizamos foi o **Ubuntu Linux**[2].

Para compilar e instalar o PAPI, executamos:

```
$ ./configure --prefix=/usr
$ make
$ su
# make install
```

Alguns dos exemplos disponíveis no diretório `utils/` do projeto foram bastante importantes para entendermos a forma como a biblioteca funciona. Em particular, o utilitário `utils/clockres.c` nos ajudou a entender como usar as funções para cronometrar tempos de execução reais e virtuais das operações tanto em ciclos como em microssegundos.

1.2 Contando o tempo com o PAPI

Foram necessárias alterações muito pequenas para fazer o tempo ser cronometrado com o PAPI no lugar de `gettimeofday()` nos arquivos `teste1.c` e `teste1_ord.c`:

[1] http://icl.cs.utk.edu/papi/
[2] http://ubuntu.com/

incluímos o cabeçalho `papi.h` no código-fonte e a função `get_time()` foi modificada de sua versão original

```
inline uint64_t get_time(void) {
    struct timeval t1;
    gettimeofday(&t1, NULL);
    return 1000000L * t1.tv_sec + t1.tv_usec;
}
```

para

```
inline uint64_t get_time(void) {
    return (uint64_t) PAPI_get_virt_cyc();
}
```

A única outra necessidade foi adicionar uma chamada à função `PAPI_library_init()` no início da função `main()` para que a biblioteca fosse inicializada.

A resolução da função `PAPI_get_virt_cyc()` é muito maior do que a da função `gettimeofday()`:

```
$ ./teste1
Time: 8171 Count 500527
$ ./teste1_ord
Time: 2671 Count 500527
$ ./teste1-papi
Time: 19567704 Count 500927
$ ./teste1_ord-papi
Time: 6495723 Count 500927
```

1.3 Hipóteses

Com uma maior resolução, iniciamos diversos testes com base nos arquivos `teste1-papi.c` e `teste1_ord-papi.c`, procurando entender o motivo de o primeiro programa ser aproximadamente 3 vezes mais lento que o segundo.

Analisando os resultados, as circunstâncias dos códigos e nossos conhecimentos até o momento, chegamos à conclusão que a diferença no tempo de execução dos programas era dado pelo acesso à variável `count`. Assim, chegamos a algumas hipóteses para o posicionamento da variável relacionadas principalmente a caches, conforme segue.

1.3.1 Número de falhas de cache

As chamadas da variável `count` eram feitas de forma consecutiva na execução do arquivo `teste1_ord-papi.c`, enquanto no arquivo `teste1-papi.c` eram feitas mais esparsamente. Isso nos levou a pensar que poderia haver diferença significativa no número de falhas de acesso à memória cache.

Usamos os eventos `PAPI_L1_DCM`, `PAPI_L2_DCM` e `PAPI_L3_DCM` para contar o número de falhas de cache no PAPI. Porém, o número foi bastante próximo para `teste1.c` e `teste1_ord.c`, de forma que tivemos que abandonar essa hipótese.

1.3.2 Número de acertos de instruções de cache

Olhando os outros eventos disponíveis na biblioteca PAPI, encontramos os eventos `PAPI_L1_ICH`, `PAPI_L2_ICH` e `PAPI_L3_ICH` que, ao invés de medirem as falhas de cache, medem o número de acertos de cache (*cache hits*).

Esse contador revelou bastante diferença entre os programas `teste1.c` e `teste1_ord.c`, como segue:

```
$ ./teste1-papi
Time: 21758132 Count 500264
Instruction cache hits (L1): 21480380
$ ./teste1_ord-papi
Time: 7742922 Count 500264
Instruction cache hits (L1): 7745845
```

Com isso, concluímos que de fato são necessários muito menos acessos ao cache quando o vetor está ordenado.

2 Parte 2

A parte 2 do EP consiste em estimar o tamanho do cache e da linha baseado nos contadores de hardware que informam o número de falhas de cache.

2.1 Utilizando hwloc

Aceitamos a sugestão do enunciado do EP de usar *hwloc*[3] para obter os valores reais do tamanho do cache e da linha. Outras alternativas seriam ler `/sys/devices/system/cpu/cpu*/*` ou ainda usar as rotinas que o PAPI no seu utilitário `utils/mem_info.c`, mas achamos interessante a promessa de um acréscimo na nota para quem usasse *hwloc*.

[3] http://www.open-mpi.org/software/hwloc/v1.7/

Instalamos o programa através do gerenciador de pacotes do Ubuntu (`apt-get install hwloc libhwloc-dev`). Também baixamos o código para usar como referência. Lendo o manual, descobrimos como inicializar o *hwloc* e como obter os tamanhos de cache e linha:

```
/* Inicializa hwloc */
if (hwloc_topology_init(&topology)) {
    fprintf(stderr, "Erro em hwloc_topology_init\n");
    exit(1);
}
hwloc_topology_load(topology);

/* Pega tamanhos reais de cache e line */
level = 0;
for (obj = hwloc_get_obj_by_type(topology, HWLOC_OBJ_PU, 0);
        obj; obj = obj->parent) {
    if (obj->type == HWLOC_OBJ_CACHE) {
        real_cache[level] = obj->attr->cache.size;
        real_line[level] = obj->attr->cache.linesize;
        level++;
    }
}
numlevels = level;
```

2.2 Utilizando contadores do PAPI

Havíamos aprendido a usar os contadores de hardware do PAPI na parte 1 do EP, quando formulamos a hipótese de que o programa `teste1_ord.c` seria mais rápido por causa do cache. Usamos aqui o mesmo código para contar o número de falhas de cache.

Para inicializar os contadores:

```
int events[3] = { PAPI_L1_DCM, PAPI_L2_DCM, PAPI_L3_DCM };
long long misses[3];
int papilevels = 3;

while (PAPI_start_counters(events, papilevels) != PAPI_OK) {
    papilevels--;
}
```

E para lê-los:

```
if (PAPI_read_counters(misses, papilevels) != PAPI_OK) {
    fprintf(stderr, "Erro em PAPI_read_counters\n");
    exit(1);
}
printf("L1 cache miss: %lld\n", misses[0]);
printf("L2 cache miss: %lld\n", misses[1]);
printf("L3 cache miss: %lld\n", misses[2]);
```

2.3 Estimando tamanho do cache e da linha

Não conseguimos fazer a parte do código para fazer a estimativa.

3 Conclusões

A enorme diferença de tempo de execução entre os programas `teste1.c` e `teste1_ord.c` comprova empiricamente que, para além da complexidade dos algoritmos, a forma como o hardware funciona faz muita diferença na otimização dos programas.

O uso de cache nos processadores para evitar acesso à memória RAM faz com que os programas rodem muito mais rápido do que rodariam se precisassem sempre acessar a memória.

A biblioteca *PAPI* pode ser uma ferramenta útil para detectarmos gargalos nas nossas implementações e entender em que hardware nossos códigos estão rodando.

O utilitário *hwloc* pode nos ajudar a ter mais ideia da topologia dos nossos computadores.