EP I de MAC412

Ana Luísa Losnak (7240258)
Tiago Madeira (6920244)

15 de setembro de 2013

Sumário

1 Parte 1

Pequena introdução (o que é a parte 1)

1.1 Instalando o PAPI

Iniciamos nosso trabalho baixando a biblioteca PAPI[1], 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 e 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`. Incluímos o cabeçalho `papi.h` no código-fonte e a função `get_time()` foi modificada de sua versão original:

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

```
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

Pequena introdução

1.3.1 Registradores

Um pouco sobre hipótese 1

1.3.2 Contando número de falhas de cache

Bastante sobre essa hipótese, com códigos que usamos para contar cache misses

1.3.3 Outra hipótese

Outra hipótese vai aqui

2 Parte 2

A parte 2 do EP consiste em estimar o tamanho do cache bla bla bla

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*.

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);
```

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

```
/* 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.

2.3 Estimando tamanho do cache e da fila

Aqui vai a parte difícil do EP

3 Conclusão

Aqui vai a conclusão