Programmazione Avanzata e parallela

Lezione 31

Luca Manzoni

Profiling

E come utilizzarlo

- Per valutare dove un programma sta utilizzando la maggior parte del suo tempo (e cosa ne limita le performance) è necessario compiere delle misurazioni

- Per ora abbiamo utilizzato delle funzioni di libreria per ottenere dei tempi

- **Domanda 1**: quanto tempo ci mette clock() a eseguire?

- Sapendo che ogni chiamata ci mette un certo tempo e non sempre è lo stesso, ha senso provare a misurare tempi sotto una certa soglia?

Profiling

Performance counters

- I processori moderni sono dotati di performance counters in hardware
- Si tratta di registri che contano alcuni eventi (e.g., numero di istruzioni eseguite, numero di cache miss, etc.)
- Hanno costo prossimo a zero (si tratta di un contatore binario collegato a qualche componente già esistente)
- Verificando il contenuto dei contatori ad intervalli regolari possiamo ottenere delle informazioni su cosa sta facendo un programma

Profiling

Profiling statistico

- Un profiler statistico compie dei sample di un programma ad intervalli regolari
- Questo ci permette di ottenere informazioni su, per esempio, quanto tempo il programma ha speso nell'eseguire certe funzioni o istruzioni
- Negli esempi useremo perf (https://perf.wiki.kernel.org/index.php/Main_Page), che è fornito con Linux, simili funzionalità sono fornite da VTune di Intel su altri sistemi

Profiling

Perf stat

- `perf stat ./nome_programma`
- Fornisce statistiche riguardanti:
  - il numero di istruzioni eseguite (e istruzioni per ciclo di clock)
  - Il numero e la percentuale di “branch-misses”
- **Primo test**: cosa succede se confrontiamo una somma con e senza branch?

Profiling

Perf stat per le cache

- `perf stat -e cache-references,cache-misses ./nome_programma`
- Riporta il numero di accessi alla cache e quante volte abbiamo avuto dei miss
- Però questo non ci riporta informazioni su dove questi sono avvenuti!
- Per questo serve creare un report dell’esecuzione da analizzare dopo

Profiling

Perf report

- `perf record ./nome_programma`
- Crea un report che può essere analizzato con il comando successivo:
- `perf report ./nome_programma`
- Un programma con interfaccia testuale che ci permette di verificare quali funzioni e quali istruzioni hanno impiegato più tempo

Profiling

Simulazione

- Un approccio differente rispetto a quello di prendere dei sample del programma mentre esegue
- Si simula l'esecuzione di un programma vedendo tutte le istruzioni che eseguirebbero branch o accessi alla memoria…
- …e simulando l'effetto sulle cache
- Sicuramente più lento (deve "simulare" l'effetto delle istruzioni)
- Possiamo anche simulare l'effetto di cache (o, in generale di hardware) diverso da quello presente sulla macchine

Profiling

Valgrind / cachegrind

- Parte di Valgrind è Cachegrind, che permette di compiere queste simulazioni
- `valgrind --tool=cachegrind --branch-sim=yes ./nome_programma`
- Simula il programma dando un report di utilizzo della cache e del branch predictor
- Simula solo due livelli di cache:
  - L1, divisa in dati e istruzioni
  - LL (“last level”) che unisce L2, L3, etc

Profiling

Valgrind / cachegrind

- Cachegrind genera anche un file **cachegrind.out.[numero]**
- Questo file contiene le informazioni per le singole funzioni
- È leggibile usando **cg_annotate cachegrind.out.[numero]**
- Se abbiamo compilato con l'opzione -g diventa possibile annotare il codice sorgente riga per riga
- **cg_annotate --auto=yes cachegrind.out.[numero]** mostra le statistiche riga per riga
- Con opzioni come **--show=Dr,D1mr,DLmr,Bc,Bcm** possiamo limitare le statistiche da mostrare

Profiling

Per Python

- Python ha un profiler integrato: **cProfile**
- È un profiler deterministico
- Ha diverse limitazioni: difficile fare il profiling di codice nativo o di codice parallelo
- Ci sono profiler più moderni, tra questi Scalene: https://github.com/plasma-umass/scalene
    - Profiling di CPU, memoria e GPU, incluso il codice nativo