Hot Spot Analysis with Pixie and Prof

Beth Richardson

NCSA Consultant and Trainer

bethr@ncsa.uiuc.edu

Hot Spot Analysis with

What is Profiling

Profiling (hot spot analysis)

- determines where a program spends its time

For both tuning and parallelization

- we want to focus our efforts on those loops doing the bulk of the computation load

90-10 rule

- 90% of the computation is done in 10% of the code

Profiling Strategy

Tune the serial code

Profile the tuned serial code

- gives the number of cycles per subroutine
- gives the number of cycles per source code line

Parallelize the computationally intensive loops

- run the parallelized code and measure speedup

If speedup is not sufficient, profile the parallelized code

- gives the number of cycles used per thread
- compare cycle counts to check load balancing

Profiling Tools

Power Challenge and Origin

- pixie and prof used together
  - does hot spot analysis
- perfex
  - monitors the hardware counters
  - useful for identifying cache problems

Exemplar

- cxpa
  - performance analyzer
  - available in both an X windows version and a command line version

Serial Profiling with pixie and prof

The following are the steps for profiling with pixie and prof.

f77 -O3 prog.f

- Produce an optimized a.out

setenv LD_LIBRARY_PATH .

- tells the loader where to look for its libraries

pixie a.out

- instruments a.out and produces a.out.pixie
  - creates a file named the original executable's name appended with ".pixie"

Serial Profiling Steps (Cont.)

a.out.pixie

- now run the pixified executable
- this creates the profile data during execution
- a file is generated named a.out.Counts
- the output file is named the original executable's name appended with ".Counts"

prof -h -pixie a.out a.out.Counts > proflist

- now run prof on the Counts file
- the output of the prof command contains the profile listings

Serial Profiling Steps (Cont.)

You specify on the prof command:

- **-h means give me "all" output in the prof output file**
- **-pixie means this is a Counts file created by pixie**
- **a.out is the name of the original executable**
- **a.out.Counts is the name of the Counts file**
- **proflist is the name of the prof output file**

Serial Profiling Listing

Gives the number of cycles executed in each procedure (aka subroutine).

The procedures are listed in descending order of cycle count, and the output is truncated here.

Cycles	%	Cum%	Secs	Calls	Proc
42630984	58.47	58.47	0.57	3626	VSUB
6498294	8.91	67.38	0.09	579	PFSOR
6141611	8.42	75.81	0.08	559	PBSOR
3654120	5.01	80.82	0.05	444	PFSOR1
2615860	3.59	84.41	0.03	142	VADD
1580424	2.17	86.57	0.02	261	ITSRCG
1144036	1.57	88.14	0.02	298	ITSRSI
886044	1.22	89.36	0.01	424	ITJSI
861136	1.18	90.54	0.01	344	ITJCG

Serial Profiling Listing (Cont.)

Also gives the number of cycles per source code line. The lines are listed in descending order of cycle count, and the output is truncated here.

Cycles	%	Cum%	Line	Proc
36556944	50.14	50.14	8106	VSUB
5313198	7.29	57.43	6974	PFSOR
4968804	6.81	64.24	6671	PBSOR
2989882	4.10	68.34	8107	VSUB
2564544	3.52	71.86	7097	PFSOR1
1988420	2.73	74.59	8103	VSUB
1629776	2.24	76.82	8045	VADD
994210	1.36	78.19	8108	VSUB
969056	1.33	79.52	8049	VADD
483018	0.66	80.18	6972	PFSOR

Serial Profile Analysis

The program being analyzed is a large code, which has approximately 10000 source code lines, and consists of many subroutines.

The first profile listing page shows that over 50% of the computation is done inside the VSUB subroutine.

The second profile listing page shows that line 8106 in subroutine VSUB accounted for 50% of the total computation.

Going back to the source code, line 8106 is a line inside a do loop.

If you put a c$doacross in front of that do loop you can get 50% of the program to run in parallel with almost no work on your part

Parallel Profiling with pixie and prof

Profile the parallelized code to check for load imbalance.

f77 -mp -O3 prog.f
- compile for multiprocessing

setenv LD_LIBRARY_PATH .
- tells the loader where to look for its libraries

setenv MP_SET_NUMTHREADS 3
- states how many threads you want to use

pixie a.out
- instruments a.out and produces a.out.pixie
- creates a file named the original executable's name appended with ".pixie"

Parallel Profiling Steps (Cont.)

a.out.pixie

- now run the pixified executable
- it collects profile data and generates a Counts file for each thread
- the output files are named with the original executable name appended with ".Counts" and with the pid (process id) of the thread appended to ".Counts"

ls *Counts*

- use this command to determine the names of the Counts files

Parallel Profiling Steps (Cont.)

Now run a separate profile for each thread

```
prof  -h  -pixie  a.out  a.out.Counts.pid0  >  profile0
prof  -h  -pixie  a.out  a.out.Counts.pid1  >  profile1
prof  -h  -pixie  a.out  a.out.Counts.pid2  >  profile2
```

You specify on the prof command:

- -h means give me “all” output in the prof output file
- -pixie means this is a Counts file created by pixie
- a.out is the name of the original executable
- a.out.Counts.pid0 is the Counts file for thread 0
- profile0 is the name of the prof output file for thread0

Parallel Profile Output

Prof gives the cycle count for each thread. Comparing the cycle counts shows whether there is a load imbalance problem. Here are the cycle counts for a run with 3 threads.

Thread	Cycles
0	8569588
1	7986149
2	7971092

Thread 0, the master thread will always have a higher cycle count than the other threads, because its does the scalar work as well as its share of the parallel work.