

Porting to Hybrid, Multi-core Systems

Heidi Poxon

**Manager & Technical Lead, Performance Tools
Cray Inc.**

- When code is network bound
 - Look at collective time, excluding sync time: this goes up as network becomes a problem
 - Look at point-to-point wait times: if these go up, network may be a problem

- When MPI starts leveling off
 - Too much memory used, even if on-node shared communication is available
 - As the number of MPI ranks increases, more off-node communication can result, creating a network injection issue

- When contention of shared resources increases

- Reduce number of MPI ranks per node
- Add parallelism to MPI ranks to take advantage of cores within a node while minimizing network injection contention
- Maximize on-node communication between MPI ranks
- Relieve on-node shared resource contention by pairing threads or processes that perform different work (for example computation with off-node communication) on the same node
- Accelerate work intensive parallel loops

- Determine where to add additional levels of parallelism
 - Assumes MPI application is functioning correctly on X86
 - Find top work-intensive loops (perftools + CCE loop work estimates)

- Split loop work among threads
 - Do parallel analysis and restructuring on targeted high level loops
 - Use CCE loopmark feedback, Reveal loopmark and source browsing

- Add parallel directives and acceleration extensions
 - Insert OpenMP directives (Reveal scoping assistance)
 - Run on X86 to verify application and check for performance improvements
 - Convert desired OpenMP directives to OpenACC

Steps to Porting to Hybrid Multi-core Systems (2)

- Run on X86 + GPU and get performance feedback
 - perftools profiling analysis

- Optimize for data locality and copies to the GPU
 - perftools accelerator statistics

- Optimize kernel on GPU
 - perftools GPU counter statistics
 - perftools Kernel statistics

- Optimize core performance on CPU
 - Automatic profiling analysis with CPU HW counter threshold feedback

**Determine where to add additional
levels of parallelism –
loop work estimates**

- Helps identify loops to optimize (parallelize serial loops):
 - Loop timings approximate how much work exists within a loop
 - Trip counts can be used to approximate work and help carve up loop on GPU
- Enabled with CCE `-h profile_generate` option
 - Should be done as separate experiment – **compiler optimizations are restricted with this feature**
- Loop statistics reported by default in `pat_report` table
- *Coming soon*: integrated loop information in profile
 - Get exclusive times and loops attributed to functions

- Access CCE and perftools software
module load PrgEnv-cray perftools
- Compile **AND** link with `-h profile_generate`
`cc -h profile_generate -c my_program.c`
`cc -h profile_generate -o my_program my_program.o`
- Instrument binary for tracing
`pat_build -u my_program` OR
`pat_build -w my_program`
- Run application
- Create report with loop statistics
`pat_report my_program+pat.xf > loops_report`

Example Report – Loop Work Estimates



Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group	Function
	Time	Time%				
						PE=HIDE
						Thread=HIDE
100.0%	176.687480	--	--	17108.0	Total	

85.3%	150.789559	--	--	8.0	USER	

85.0%	150.215785	24.876709	14.4%	2.0	jacobi_.LOOPS	
=====						
12.2%	21.600616	--	--	16071.0	MPI	

11.9%	21.104488	41.016738	67.1%	3009.0	mpi_waitall	
=====						
2.4%	4.297301	--	--	1007.0	MPI_SYNC	

2.4%	4.166092	4.135016	99.3%	1004.0	mpi_allreduce_(sync)	
=====						

Example Report – Loop Work Estimates (2)

Table 3: Inclusive Loop Time from -hprofile_generate

Loop Incl	Loop	Loop	Loop	Function=/.LOOP[.]
Time	Hit	Trips	Trips	PE=HIDE
Total		Min	Max	

...				
175.676881	2	0	1003	jacobi_.LOOP.07.li.267
0.917107	1003	0	260	jacobi_.LOOP.08.li.276
0.907515	129888	0	260	jacobi_.LOOP.09.li.277
0.446784	1003	0	260	jacobi_.LOOP.10.li.288
0.425763	129888	0	516	jacobi_.LOOP.11.li.289
0.395003	1003	0	260	jacobi_.LOOP.12.li.300
0.374206	129888	0	516	jacobi_.LOOP.13.li.301
126.250610	1003	0	256	jacobi_.LOOP.14.li.312
126.223035	127882	0	256	jacobi_.LOOP.15.li.313
124.298650	16305019	0	512	jacobi_.LOOP.16.li.314
20.875086	1003	0	256	jacobi_.LOOP.17.li.336
20.862715	127882	0	256	jacobi_.LOOP.18.li.337
19.428085	16305019	0	512	jacobi_.LOOP.19.li.338
=====				

**Do parallel analysis and restructuring
on targeted high level loops –
Reveal**

- Generate compiler program library with whole program analysis for more in-depth inter-procedural analysis
 - `% cc -hwp -h pl=/path_to_my_program_library/`

- Generate loopmark information, view .lst files
 - `% cc -rm -c my_program.c`

- Use Reveal to view loopmark information, compiler messages, browse source

New code restructuring and analysis assistant...

- Uses both the performance toolset and CCE's program library functionality to provide static and runtime analysis information
- Assists user with the code optimization phase by **correlating source code with analysis** to help identify which areas are key candidates for optimization

■ Key Features

- **Annotated source code** with compiler optimization information
 - Feedback on critical dependencies that prevent optimizations
- **Scoping analysis**
 - Identify, shared, private and ambiguous arrays
 - Allow user to privatize ambiguous arrays
 - Allow user to override dependency analysis
- **Source code navigation** based on performance data collected through CrayPat

Source Code – Loopmark

Compiler feedback

The screenshot displays a compiler interface with a source code editor on the right and a performance feedback panel on the left. The source code is a Fortran loop starting at line 66. The performance panel shows a tree view of code blocks, with 'Loop@66' highlighted in green and 'Loop@89' highlighted in yellow. A blue callout bubble labeled 'Performance feedback' points to the performance panel. Another blue callout bubble labeled 'Compiler feedback' points to the source code editor. An 'Info' window at the bottom left provides details for line 66.

```
32.33% calc2.F
└─ 32.33% CALC2
   └─ Loop@66
      └─ Loop@67
         └─ Loop@89
            └─ 17.34% calc1.F
               └─ 0.21% swim.F
```

```
66 DO 200 I=1,M
67 DO 200 J=js,je
68   UNEW(I+1,J) = UOLD(I+1,J)+
69   1   TDTS8*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J+1)+CV
70   2   +CV(I+1,J))-TDTSDX*(H(I+1,J)-H(I,J))
71   if(j.gt.1)then
72     VNEW(I,J) = VOLD(I,J)-TDTS8*(Z(I+1,J)+Z(I,J))
73     1   *(CU(I+1,J)+CU(I,J)+CU(I,J-1)+CU(I+1,J-1))
74     2   -TDTSDY*(H(I,J)-H(I,J-1))
75   endif
76   if(j.eq.n)then
77     VNEW(I,J+1) = VOLD(I,J+1)-TDTS8*(Z(I+1,J+1)+Z(
78     1   *(CU(I+1,J+1)+CU(I,J+1)+CU(I,J)+CU(I+1,J))
79     2   -TDTSDY*(H(I,J+1)-H(I,J))
80   endif
81     PNEW(I,J) = POLD(I,J)-TDTSDX*(CU(I+1,J)-CU(I,J))
82     1   -TDTSDY*(CV(I,J+1)-CV(I,J))
83   200 CONTINUE
84
85 CME-----
86 C
```

Info
Line 66:
Loop unrolled 2 times.
Loop interchanged with loop
at line 67.

**Add parallel directives and
acceleration extensions -
Reveal**

Display Scoping Information for Selected Loop

```
290 ldir$ omp_analyze_loop
291 DO K=2,kmax-1
292   DO J=2,jmax-1
293     DO I=2,imax-1
294       S0=a(I,J,K,1)*p(I+1,J, K) &
295         +a(I,J,K,2)*p(I, J+1,K) &
296         +a(I,J,K,3)*p(I, J, K+1) &
297         +b(I,J,K,1)*p(I+1,J+1,K) &
298         -p(I+1,J-1,K) &
299         -p(I-1,J+1,K) &
300         +p(I-1,J-1,K)) &
301         +b(I,J,K,2)*p(I, J+1,K+1) &
302         -p(I, J-1,K+1) &
303         -p(I, J+1,K-1) &
304         +p(I, J-1,K-1)) &
305         +b(I,J,K,3)*p(I+1,J, K+1) &
306         -p(I-1,J, K+1) &
307         -p(I+1,J, K-1) &
308         +p(I-1,J, K-1)) &
309         +c(I,J,K,1)*p(I-1,J, K) &
310         +c(I,J,K,2)*p(I, J-1,K) &
311         +c(I,J,K,3)*p(I, J, K-1) &
312         +wrk1(I,J,K)
313       SS=(S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
314       WGOSA=WGOSA+SS*SS
315       wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
316     enddo
317   enddo
318 enddo
319   wgosaf = wgosaf + wgosaf
320 !!$ AH: pack buffers containing the halos
321 !!$ Could use acc_update here but non-contiguous array shapes currently
322 !!$ not supported
323 !!$ A hack to make sure we don't end up with an empty block
324 ldir$ omp_analyze_loop
```

Name	Type	Scope	F	L	Info
a	Array	Shared			
b	Array	Shared			
bnd	Array	Shared			
c	Array	Shared			
imax	Scalar	Shared			
jmax	Scalar	Shared			
kmax	Scalar	Shared			
omega	Scalar	Shared			
p	Array	Shared			
s0	Scalar	Private	N	N	
ss	Scalar	Private	N	N	
wgosaf	Scalar	Shared			
wrk1	Array	Shared			
wrk2	Array	Shared			

- Navigate by profile call tree with loops
- Initiate scoping analysis from within Reveal (no `omp_analyze` directives or compiler command-line option)
- Directive generation and insertion into source
- Focus on loops with unknowns
- Create OpenMP or OpenAcc directives
- Highlight “interesting” compiler feedback
 - Was call site flattened or not?
 - Was loop flattened or not?
 - Was loop or region pattern-matched?

How to use Reveal 0.1 (early alpha version)

- Use cce 8.0.3 or later
- Start with clean build
- Collect loop statistics with cce and perftools to identify loops to parallelize

- Add `!dir$ omp_analyze_loop` directive before each loop to parallelize
 - This directive only works with serial loops. Add `-x omp` or `-x acc` to your cce compile options if loop is already parallel
- Compile application for scoping analysis
 - `% ftn -homp_analyze -hwp -hpl=/full_path/program.pl`
- Launch reveal:
 - `% reveal program.pl`

How to use Reveal 0.1 (early alpha version)

- Expand files and functions to look for loops with scoping information (highlighted green)
- Scope any unknowns
- Dump scoping information to stderr (where you launched reveal) to copy and past into a directive in your source by clicking “Dump Data”

Questions

??