

Getting Ready to EuroHack

Alistair Hart (Cray) ahart@cray.com

COMPUTE | STORE | ANALYZE

Getting Ready to EuroHack



- EuroHack starts on Monday July 6th
- One intensive week is a great opportunity
- But you will get a lot further if you do some work first
- These slides describe some things you can do now
 - to give you a running start on day 1.



Learn a bit about OpenACC



- EuroHack is not a training course
 - We're all going to learn a lot about OpenACC
 - But we assume that you know the basics already
- Things you can do:
 - Watch some CSCS training videos, e.g.
 - https://www.youtube.com/playlist?list=PL1tk5lGm7zvQOLguaplvYUfkBCVx0GMyW
 - Read the OpenACC standard (it's not as bad as you think)
 - Download from http://www.openacc.org



Machine access



We will use the CSCS machine "Piz Daint"

- It has GPU-accelerated nodes
- It has OpenACC-aware compilers
- It has a full GPU development environment

You should already have access to daint

- We will use special course accounts for EuroHack
- These will avoid your jobs having to wait in the main queue

• Very first steps:

- Make sure you can log in with your course account
- Make sure you can (and know how to) submit a simple job



Compiler support



daint has two OpenACC compilers

- The Cray Compilation Environment (CCE)
- The PGI compiler
- (gcc will support OpenACC in the future, but not yet)

You will need to be able to compile your code with these

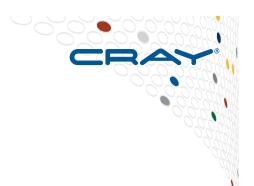
- At least one compiler
- Preferably both, for comparisons and debugging

Your next step:

- Understand how to use CCE and PGI compilers on daint
- Develop and test build scripts (Makefiles, etc.) for your code
 - using CCE and PGI
- Contact CSCS if there are any problems getting your code to compile



Test cases



- Code development is an iterative process
 - Repeated cycles of: edit, compile, run

You will need to prepare two testcases to run

- 1. A representative problem
 - Exercises all the relevant parts of the code
 - Should spend the right amount of time in the different parts of the code
 - Reduce this as much as you can (whilst still being representative)
 - weak scale to as few nodes as possible (but not too few)
 - reduce the runtime as much as possible (but not too short)
 - reduce the number of timesteps
 - speed up the initialisation
- 2. A rapid development testcase
 - Exercises all the relevant parts of the code
 - Needn't spend the right amount of time in the different parts of the code
 - Make this as quick to run as possible (ideally, a couple of minutes or less)



Correctness checking



- A successful OpenACC port is done incrementally
 - It will not be faster at every step
 - But it should be numerically correct at every step

Your next step

- Make sure your code has correctness checks (or add them)
 - e.g. known values in output file that don't change (for a given testcase)
 - array checksums, solver residuals...
 - or internal checks in the code
 - The "domain experts" in the team can help with this
- If you have time, write a small validation script
 - Parses the job output and checks for correctness





Profiling is important

- not just to measure code performance
- but also to guide the OpenACC porting process

CrayPAT is the Cray profiling tool

It works with all the compilers on daint

COMPUTE

It profiles CPU and GPU parts of your code

CrayPAT works with all the compilers on daint

It has some extra features with CCE, designed for OpenACC porting



Profiling your code



- First, a sampling profile is useful:
 - Build the code:
 - "make clean" to remove all the object files
 - "module load perftools"
 - "make" to build executable EXE
 - Instrument the code
 - "pat_build -f EXE" to create new executable EXE+pat
 - Now execute EXE+pat using usual jobscript
 - A file or directory is created; name ends in ".xf"
 - Create the profile
 - pat_report <xf file/directory>
 - Save this report in a text file







- Don't profile the initialisation
 - just the main computational steps
 - e.g. timestep loop
- Use the CrayPAT API to do this
 - Fortran, C, C++
- See "man pat_build" for details
- CPP macro: CRAYPAT
 - Useful so code compiles with and without perftools
 - CCE: automatically defines CRAYPAT when perftools loaded
 - PGI: need to manually include -DCRAYPAT when perftools loaded
- Repeat your profiles using the API

```
#ifdef CRAYPAT
include 'pat apif.h'
#endif
! declarations
#ifdef CRAYPAT
call PAT record(PAT STATE OFF, ipat)
#endif
! initialisation code
#ifdef CRAYPAT
call PAT record(PAT STATE ON,ipat)
#endif
! computation code
#ifdef CRAYPAT
call PAT record(PAT STATE OFF,ipat)
#endif
! finalisation code
```

Getting a calltree



- A calltree shows which routines call which
 - Potentially changes for different testcases
 - Provides a map for the OpenACC port
- The OpenACC port will start at the child routines
 - Then work back up the calltree
- Generate the calltree from existing CrayPAT data:
 - pat_report -O calltree -T <xf file/directory>
 - Save this report in a text file
- T includes all routines in the output
 - Not just those taking more than 2% (the default)



Loop level profiling



Profiling is typically at the routine level

Profiling at the loop level is very useful for OpenACC

- Which loopnests take the most time?
- How many iterations do these loopnests typically have?
- How much work is there per iteration

CrayPAT supports loop-level profiling with CCE only

- module load perftools
- compile with new compiler flag "-h profile_generate"
- pat build and execute the code as before
- pat_report now gives loop-level profiling information
- Save this report in a text file



Summary



Steps you can take now to get a running start at EuroHack:

- 1. Learn about OpenACC
- 2. Port your existing code to CCE and PGI compilers
- 3. Develop suitable testcases
- 4. Develop suitable correctness checks
- 5. Generate profiles of your existing code with CrayPAT
 - CrayPAT API
 - Flat profile
 - Calltree
 - Loop level profiling with CCE
- You should definitely aim to complete steps 1-4.
- It's even better if you can do step 5 as well.

