HPX integration: APEX
(Autonomic Performance Environment for eXascale)

Kevin Huck,
With contributions from Nick Chaimov and Sameer Shende, Allen Malony

khuck@cs.uoregon.edu
http://github.com/khuck/xpress-apex
APEX Runtime Adaptation: Motivation

- **Controlling concurrency**
  - Energy efficiency
  - Performance

- **Parametric variability**
  - Granularity for this machine / dataset?

- **Load Balancing**
  - When to perform AGAS migration?

- **Parallel Algorithms (for_each...)**
  - Separate *what* from *how*

- **Address the “SLOW(ER)” performance model**

- **Post-mortem performance analysis**
  - “secondary” goal, but has been useful
Introduction to APEX

- Performance awareness and performance adaptation
- Top down and bottom up performance mapping / feedback
  - Make node-wide resource utilization data and analysis, energy consumption, and health information available in real time
  - Associate performance state with policy for feedback control
- APEX introspection
  - OS: track system resources, utilization, job contention, overhead
  - Runtime (HPX): track threads, queues, concurrency, remote operations, parcels, memory management
  - Application timer / counter observation
APEX architecture

APEX Introspection
- Synchronous
- Asynchronous

APEX State
- Triggered
- Periodic

System Info
(RCR Toolkit, /proc, getrusage, etc.)

Application

HPX

events

meta-events

actuators
APEX Introspection

• APEX collects data through “inspectors”
  – *Synchronous* uses an event API and event “listeners”
    • Initialize, terminate, new thread – added to HPX runtime
    • Timer start, stop, yield*, resume* - added to HPX task scheduler
    • Sampled value (counters from HPX)
    • Custom events (meta-events)
  – *Asynchronous* do not rely on events, but occur periodically

• APEX exploits access to performance data from lower stack components
  – Reading from the RCR blackboard (i.e., power, energy)
  – “Health” data through other interfaces (/proc/stat, cpuinfo, meminfo, net/dev, self/status, lm_sensors, power*, etc.)
APEX Event Listeners

• Profiling listener
  – Start event: input name/address, get timestamp, return profiler handle
  – Stop event: get timestamp, put profiler object in a queue for back-end processing, return
  – Sample event: put the name & value in the queue
  – Asynchronous consumer thread: process profiler objects and samples to build statistical profile (in HPX, processed/scheduled as a thread/task)
  – Optional: build taskgraph, build task scatterplot, screen/CSV output, etc.

• TAU Listener (postmortem analysis)
  – Synchronously passes all measurement events to TAU to build an offline profile

• OTF2 Listener (postmortem analysis)
  – Synchronously passes all measurement events to libotf2 for trace analysis

• Concurrency listener (postmortem analysis)
  – Start event: push timer ID on stack
  – Stop event: pop timer ID off stack
  – Asynchronous consumer thread: periodically log current timer for each thread, output report at termination
APEX Policy Listener

• **Policies** are rules that decide on outcomes based on observed state
  – *Triggered* policies are invoked by introspection API events
  – *Periodic* policies are run periodically on asynchronous thread

• Polices are registered with the Policy Engine
  – Applications, runtimes, and/or OS register callback functions

• Callback functions define the policy rules
  – “If \( x < y \) then…”

• Enables runtime adaptation using introspection data
  – Feedback and control mechanism
  – Engages actuators across stack layers
  – Could also be used to involve online auto-tuning support*
    • Active Harmony [http://www.dyninst.org/harmony](http://www.dyninst.org/harmony)
Building HPX with APEX

$ cmake <usual HPX settings>...

-DHPX_WITH_APEX=TRUE \ 
-DAPEX_WITH_ACTIVEHARMONY=ON/OFF \ 
-DAPEX_WITH_PAPI=ON/OFF \ 
-DAPEX_WITH_MSR=ON/OFF \ 
-DAPEX_WITH_OTF2=ON/OFF \ 
..

$ make (as usual) i.e. “make –j 8 core examples tests”
APEX environment variables

APEX_DISABLE : 0
APEX_SUSPEND : 0
APEX_PAPI_SUSPEND : 0
APEX_PROCESS_ASYNC_STATE : 1
APEX_TAU : 0
APEX_POLICY : 1
APEX_MEASURE_CONCURRENCY : 0
APEX_MEASURE_CONCURRENCY_PERIOD : 1000000
APEX_SCREEN_OUTPUT : 1
APEX_PROFILE_OUTPUT : 0
APEX_CSV_OUTPUT : 0
APEX_TASKGRAPH_OUTPUT : 0
APEX_PROC_CPUINFO : 0
APEX_PROC_MEMINFO : 0
APEX_PROC_NET_DEV : 0
APEX_PROC_SELF_STATUS : 0
APEX_PROC_SELF_IO : 0
APEX_PROC_STAT : 1
APEX_PROC_PERIOD : 1000000
APEX_THROTTLE_CONCURRENCY : 0
APEX_THROTTLING_MAX_THREADS : 4
APEX_THROTTLING_MIN_THREADS : 1
APEX_THROTTLE_ENERGY : 0
APEX_THROTTLE_ENERGY_PERIOD : 1000000
APEX_THROTTLING_MAX_WATTS : 300
APEX_THROTTLING_MIN_WATTS : 150
APEX_PTHREAD_WRAPPER_STACK_SIZE : 0
APEX_OMPT_REQUIRED_EVENTS_ONLY : 0
APEX_OMPT_HIGH_OVERHEAD_EVENTS : 0
APEX_PIN_APEX_THREADS : 1
APEX_TASK_SCATTERPLOT : 1
APEX_POLICY_DRAIN_TIMEOUT : 1000
APEX_PAPI_METRICS :
APEX_PLUGINS :
APEX_PLUGINS_PATH : /
APEX_OTF2 : 0
APEX_OTF2_ARCHIVE_PATH : OTF2_archive
APEX_OTF2_ARCHIVE_NAME : APEX
Elapced time: 2.40514 seconds
Cores detected: 4
Worker Threads observed: 5
Available CPU time: 9.62057 seconds

<table>
<thead>
<tr>
<th>Timer</th>
<th>#calls</th>
<th>mean</th>
<th>total</th>
<th>% total</th>
</tr>
</thead>
<tbody>
<tr>
<td>someThread(void*)</td>
<td>2</td>
<td>1.39e+00</td>
<td>2.79e+00</td>
<td>28.965</td>
</tr>
<tr>
<td>foo(int)</td>
<td>131072</td>
<td>2.00e-05</td>
<td>2.63e+00</td>
<td>27.306</td>
</tr>
<tr>
<td>bar(int, apex::profiler**, void**)</td>
<td>131072</td>
<td>9.69e-06</td>
<td>1.27e+00</td>
<td>13.198</td>
</tr>
<tr>
<td>someUntimedThread(void*)</td>
<td>2</td>
<td>1.12e+00</td>
<td>2.24e+00</td>
<td>23.271</td>
</tr>
<tr>
<td>main</td>
<td>1</td>
<td>1.40e+00</td>
<td>1.40e+00</td>
<td>14.548</td>
</tr>
<tr>
<td>APEX MAIN</td>
<td>1</td>
<td>2.41e+00</td>
<td>2.41e+00</td>
<td>100.000</td>
</tr>
</tbody>
</table>

Total timers : 262,150
Concurrency View

Massively Parallel Task-Based Programming with HPX: Performance Analysis of HPX using TAU and APEX
Concurrency View

![Graph showing concurrency view with x-axis representing time (sample period) from 0 to 750, y-axis representing total threads executing from 0 to 60000, and power in watts from 0 to 200000. The graph illustrates the dynamic nature of threads executing and power consumption throughout the sample period.]
OTF2 View in Vampir

Massively Parallel Task-Based Programming with HPX: Performance Analysis of HPX using TAU and APEX
Profile View in TAU ParaProf

Metric: TIME
Value: Exclusive

Std. Dev. | Mean | Max | Min
---|---|---|---
node 0 | | | |
node 1 | | | |
node 2 | | | |
node 3 | | | |
node 4 | | | |
node 5 | | | |
node 6 | | | |
node 7 | | | |
node 8 | | | |
node 9 | | | |
node 10 | | | |
node 11 | | | |
node 12 | | | |
node 13 | | | |
node 14 | | | |
node 15 | | | |
Task Scatterplot Analysis (prototype)
Taskgraph View (prototype)

```
main: 0.134676s
```

```
count: 1
```

```
fib(int): 6.66041s
```
Example : HPX+APEX+TAU

- Heat diffusion
- 1D stencil code
- Data array partitioned into chunks
- 1 node with no hyperthreading
- Performance increases to a point with increasing worker threads, then decreases
Concurrence & Performance

• Region of maximum performance correlates with thread queue length runtime performance counter
  – Represents # tasks currently waiting to execute
• Could do introspection on this to control concurrency throttling policy (see next example)
1d_stencil_4 Baseline

- 48 worker threads (with hyperthreading, on Edison)
- Actual concurrency much lower
  - Implementation is memory bound
- Large variation in concurrency over time
  - Tasks waiting on prior tasks to complete

138 secs

Massively Parallel Task-Based Programming with HPX: Performance Analysis of HPX using TAU and APEX
1d_stencil w/optimal # of Threads

- 12 worker threads on Edison
- Greater proportion of threads kept busy
  - Less interference between active threads and threads waiting for memory
- Much faster
  - 61 sec. vs 138 sec.
1d_stencil Adaptation with APEX

- Initially 32 worker threads
- ActiveHarmony searches for minimal thread queue length
- Quickly converges on 20
Adapting Block Size

• Is 1000 partitions of 100000 cells the best partitioning?
• Parametric studies say “no”.
• Can we modify the example to repartition as necessary to find better performance?
1d_stencil: adapting block size

![Graph showing concurrency and grain size parameter over time](image)

- **grain_size_parameter**
- **other**
- **hpx_main**
- **hpx::lcos::local::dataflow::execute**
Next Steps (?)

- Measurement infrastructure in place
- Policies
- Policies
- Policies
  - Parcel coalescing
  - Auto-chunking / Parallel Algorithms
- Updated documentation
  - Quickstart
  - Howto
  - FAQ
  - User/Reference guides
• Support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (and Basic Energy Sciences/Biological and Environmental Research/High Energy Physics/Fusion Energy Sciences/Nuclear Physics) under award numbers DE-SC0008638, DE-SC0008704, DE-FG02-11ER26050 and DE-SC0006925.

• This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

• This material is (will be?) based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 079289626(?).

• DOD?

• The authors acknowledge that the Talapas HPC cluster and the High Performance Computing Research Core Facility at The University of Oregon have contributed to the research results reported within this paper.