Retargeting of the Open Community Runtime to Intel Xeon Phi

Jiri Dokulil and Siegfried Benkner
Research Group Scientific Computing,
University of Vienna, Austria
Open Community Runtime (OCR)

- runtime system for extreme scale parallel systems
- work in progress
  - specification 0.9
  - implementation targeted for this year
    - https://github.com/01org/ocr
- task based
  - all work is performed by event driven tasks (EDTs)
- explicit management of data
  - in data blocks (DBs) managed by the runtime
- fault tolerance
OCR basic concepts

- execution controlled using events and dependencies

- tasks and events have pre-slots and post-slots, which connect to form dependencies

- data block may be passed along dependencies or a DB may be used to satisfy a pre-slot directly
OCR basic concepts cont.

- A task can only run once all pre-slots have been satisfied.
- Any DB that the task wants to access has to be passed through one of the pre-slots.
  - An access mode has to be specified:
    - RO: read only
    - EW: exclusive write
    - NCR: non-coherent read
    - ITW: intent to write (“non-coherent write”)
- The access modes also need to be dealt with before the task starts:
  - Locking, but also possibly multiple copies.
db initial_task() {
    (t1, e1) = create_task(process, 1);
    (t2, e2) = create_task(process, 1);
    (t_final, e_final) = create_task(finalize, 3);
    db data = create_data();
    add_dependency(data, t_final, EW);
    add_dependency(e1, t_final, EW);
    add_dependency(e2, t_final, EW);
    add_dependency(data, t1, MODE1);
    add_dependency(data, t2, MODE2);
    return NULL;
}

db process(db arg) {
    work(arg);
    return NULL;
}

db finalize(db arg, db arg1, db arg2) {
    print_result(arg);
    return NULL;
}
db initial_task() {
    (t1,e1) = create_task(process,1); //Create task t1 and event e1. Task t1 has one pre-slot, executes the function process, and triggers event e1 when finished.
    (t2,e2) = create_task(process,1); //The pair (t2,e2) is analogous to (t1,e1)
    (t_final,e_final) = create_task(finalize,3); //Three pre-slots, runs finalize.
    db data = create_data(); //Create a new data block.
    add_dependency(data,t_final,EW); //Bind the data block to a pre-slot of t_final
    add_dependency(e1,t_final,EW); //Bind post-slot of e1 to a pre-slot of t_final;
    add_dependency(e2,t_final,EW); //the mode is irrelevant as no data is passed.
    add_dependency(data,t1,MODE1); add_dependency(data,t2,MODE2);
    return NULL;
}

db process(db arg) { //arg contains the data block
    work(arg);
    return NULL;
}

db finalize(db arg, db arg1, db arg2) {
    print_result(arg);
    return NULL;
}
Our shared memory implementation

- implements most of the 0.9 specification
- built on top of TBB
  - mainly it’s task scheduler
- custom dependence tracking
  - TBB not sophisticated enough
  - locks similar to RW-locks to cover the 4 OCR modes
    - block required by a task acquired in order to prevent deadlocks
    - if a block cannot be acquired, the task is added to a waitlist
  - copies of data not used to optimize access
  - once dependencies are satisfied, task submitted to TBB for execution
- no centralized locking needed
Applications

- **SPH**
  - smoothed particle hydrodynamics
  - n-body simulation of interacting particles
  - time-step loop
  - all-to-all interaction within a cutoff radius
    - variable task duration

- **Seismic**
  - example used in the TBB
  - propagation of seismic waves in 2D
  - stencil operation, 2 phases
  - each phase processed by several parent tasks
    - each task spawns more tasks to do the actual work
Experiments

- CPU
  - dual Xeon X5680 (3.3GHz, 6 cores, 12MB cache)
- Xeon Phi
  - 7120P (1.238GHz, 61 cores, 16 GB RAM)

- OCR and OpenMP implementation
- more compute-intensive variants (x-2)

<table>
<thead>
<tr>
<th>Host</th>
<th>OCR time</th>
<th>SPH</th>
<th>SPH-2</th>
<th>Seismic</th>
<th>Seismic-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>0.760 (0.016)</td>
<td>25.470 (0.538)</td>
<td>0.046 (0.002)</td>
<td>52.394 (1.126)</td>
<td></td>
</tr>
<tr>
<td>OCR time</td>
<td>0.842 (0.030)</td>
<td>25.577 (0.523)</td>
<td>0.158 (0.063)</td>
<td>57.035 (3.871)</td>
<td></td>
</tr>
<tr>
<td>speedup</td>
<td>1.107</td>
<td>1.004</td>
<td>3.462</td>
<td>1.089</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.955 (0.047)</td>
<td>4.408 (0.049)</td>
<td>0.528 (0.036)</td>
<td>20.106 (0.189)</td>
<td></td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>1.843 (0.010)</td>
<td>3.484 (0.006)</td>
<td>0.280 (0.005)</td>
<td>19.985 (0.071)</td>
<td></td>
</tr>
<tr>
<td>OCR time</td>
<td>0.942</td>
<td>0.790</td>
<td>0.529</td>
<td>0.994</td>
<td></td>
</tr>
<tr>
<td>speedup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Execution trace (CPU)

- SPH

- Seismic
Execution trace (Xeon Phi)
Conclusion

- the OCR should be usable even on the Xeon Phi
- distributed version can mostly be based on what we already have
  - almost there...
- OCR could play a similar role to TBB scheduler
  - simple, straightforward interface
  - only the most necessary features
  - good efficiency
  - the “clever” scheduling done by higher levels
    - consider the parallel algorithms provided by TBB