A UPC++ Actor Library and its Evaluation on a Shallow Water Application

Alexander Pöppl\textsuperscript{1}, Scott Baden\textsuperscript{2}, Michael Bader\textsuperscript{1}

\textsuperscript{1}Department of Informatics
Technical University of Munich

\textsuperscript{2}Computational Research Division
Lawrence Berkeley National Laboratory
Department of Computer Science and Engineering
University of California, San Diego

Parallel Applications Workshop, Alternatives To MPI+X
November 18th 2019
Denver, Colorado
Motivation

Invasive Computing:
- Dynamic resource allocation
- Predictability through exclusive resource usage
- Heterogeneous compute tiles

Actor-based Modelling
- Good fit for architecture, enables exploration of different mappings of actors to compute tiles
- SWE-X10 as sample application

Transfer to larger-scale applications

*Is it feasible to program an actor library using standard languages and frameworks?*

*If so, how does performance compare, both to our X10-based library, and BSP?*

✓ Tools: C++, OpenMP, UPC++
UPC++

Asynchronous Partitioned Global Address Space (APGAS) Model

Reliance on one-sided communication

Asynchronous, continuation-based API

Based on GASNet-EX, makes direct use of InfiniBand and (some) Cray interconnects

**UPC++**

**RPCs**

- Executed asynchronously
- Serialization and transfer of parameters, return value
- Completion events available after the local part (or overall RPC execution) is finished
RPCs
- Executed asynchronously
- Serialization and transfer of parameters, return value
- Completion events available after the local part (or overall RPC execution) is finished

Global Pointers
- Point to data in Shared segment
- May be used as target for RMA operations
RPCs
- Executed asynchronously
- Serialization and transfer of parameters, return value
- Completion events available after the local part (or overall RPC execution) is finished

Global Pointers
- Point to data in Shared segment
- May be used as target for RMA operations

Distributed Objects
- Created collectively
- Same handle points to different objects on each rank
Actors
- Encapsulate specific functionality, data and behavior
- Behavior defined through finite state machines
- No data sharing between actors
- Defined communication endpoints (Ports)
- Have the ability compute whenever data in their ports (InPorts or OutPorts) changes
  - Actors are being triggered

Application Developers…
- …subclass and implement act() method (actor FSM)
- …use ports as communication endpoints
- …specify which ports are connected
Channels

- Unidirectional connection between two ports
- FiFo semantics
- Operations: `read()`, `write(T)`, `peek()`
- Guards: `available()`, `freeCapacity()`
UPC++ Actor Library – Write

1:RPC (insert Data)

2:LPC (trigger Actor)

3:LPC (track RPC completion)

A1::Out A1

A2::In A2

Channel

Rank N

Rank M
UPC++ Actor Library – Read

A1

A1::Out  

Channel

A2

A2::In

1:read  
(dequeue Data)

2:RPC  
(update capacity)

3:LPC  
(trigger Actor)

4:LPC  
(track RPC completion)
Rank-based Execution Strategy

One thread per UPC++ rank, one rank per (logical) core

One event loop:
- Query runtime for progress
- Execute RPCs, mark affected actors
- Execute act() on affected actors

May use sequential UPC++ code mode

Low number of actors per rank
Thread-based Execution Strategy

One thread per actor, and one communication thread, low number of ranks per node

Two event loops:
- Communication thread queries runtime and executes RPCs
- Actor threads query runtime for progress and execute LPCs, execute act

Requires balancing of communication thread against number of actors
Task-based Execution Strategy

Map act() executions on OpenMP tasks

One event loop:
- Master thread queries Runtime
- Performs any incoming RPCs and triggers affected actors
- Schedules OpenMP task for each invocation of act. Dependencies between act invocations of same actor

Large number of actors per rank possible
Pond – A Shallow Water Proxy Application

Based on prior applications
- **SWE**, a BSP-based code written using MPI and OpenMP
- **SWE-X10**, an actor-based X10 application written using the actorX10 library

Parallelized using our actor library

Possible to auto-vectorize with AVX512 with Intel Compiler (v18.0)
Pond – A Shallow Water Proxy Application

\[
\begin{bmatrix}
    h \\
    hu \\
    hv
\end{bmatrix}_t + \begin{bmatrix}
    hu \\
    hu^2 + \frac{1}{2}gh^2 \\
    hv
\end{bmatrix}_x + \begin{bmatrix}
    hv \\
    huv \\
    hv^2 + \frac{1}{2}gh^2
\end{bmatrix}_y = S(t, x, y)
\]
Pond – A Shallow Water Proxy Application

Finite volume scheme on a Cartesian grid with piecewise constant unknown quantities and Euler time step

\[ Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-\frac{1}{2},j}^n + A^- \Delta Q_{i+\frac{1}{2},j}^n \right) \]

\[ - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-\frac{1}{2}}^n + B^- \Delta Q_{i,j+\frac{1}{2}}^n \right) \]

Finite volume scheme on a Cartesian grid with piecewise constant unknown quantities and Euler time step

\[
Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x} \left( A^+ \Delta Q_{i-\frac{1}{2},j}^n + A^- \Delta Q_{i+\frac{1}{2},j}^n \right) - \frac{\Delta t}{\Delta y} \left( B^+ \Delta Q_{i,j-\frac{1}{2}}^n + B^- \Delta Q_{i,j+\frac{1}{2}}^n \right)
\]
Pond – A Shallow Water Proxy Application

Subdivision into rectangular, equally-sized patches with Halo regions
Pond – A Shallow Water Proxy Application

One actor per patch

Actors connected with direct neighbors
Pond – Simulation Actor

\[ t_{\text{cur}} < t_{\text{end}} \land \text{mayRead()} \land \text{mayWrite()} \]

\[ \text{receiveData(); computeFluxes(); applyUpdates(); sendData()} \]

\[ t_{\text{cur}} \geq t_{\text{end}} \]

\[ \text{stop()} \]

\[ \text{mayWrite()} \]

\[ \text{sendData()} \]
Evaluation

Performed on NERSC Cori
- Single socket Intel Xeon Phi (Knights Landing) nodes
- 68 cores (272 hyperthreads) per node
- 16GB MCDRAM
- 6TFlop/s (SP)
- Intel Compiler 18, Vectorization using AVX512

Comparison of
- SWE-X10, prior X10 application, based on actorX10, an X10 actor library
- SWE, prior MPI+OpenMP application, follows the BSP model
- Pond using our actor library (using the three available execution strategies, Rank, Thread and Task)

All subjects follow same numerical approach, same Riemann solver used in all cases
Evaluation – Weak Scaling

Radial Dam Break Scenario

$4096^2$ grid cells per node
Evaluation – Weak Scaling

Radial Dam Break Scenario

16384² grid cells per node

Patch sizes from 512x512 down to 64x64 grid cells
Conclusion

Competitive with OpenMP and MPI

UPC++ enables overlap of communication and computation

Higher abstraction level for application programmer

Flexibility regarding backend
Questions + Acknowledgements

- This research was funded by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) - Project number 14671743 - TRR 89 Invasive Computing.

- This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

- 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.

- Scott Baden was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

UPC++ Tutorial at LBL
- December 16th
- At NERSC or Online

More Info at:
https://www.exascaleproject.org/event/upccpp
https://upcxx.lbl.gov  ➤ News