Performance Evaluation of ParalleX Execution model on Arm-based Platforms

Nikunj Gupta[1,3] Rohit Ashiwal[1] Bine Brank[2] Sateesh K. Peddoju[1] Dirk Pleiter[2]

भारतीय प्रौद्योगिकी संस्थान रूड़की
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE [1]

[2]

STE||AR GROUP [3]

Background

Asynchronous Many-Task (AMT) Programming model

- Define work in form of tasks and express any data and/or control constraints
- Let runtime extract as much or as little parallelism, provided they follow the user constraints
- Each task runs off on a (light-weight) thread
  - Synchronization handled by the runtime
  - Runtime schedules the threads
- Scheduler deals with the load imbalance

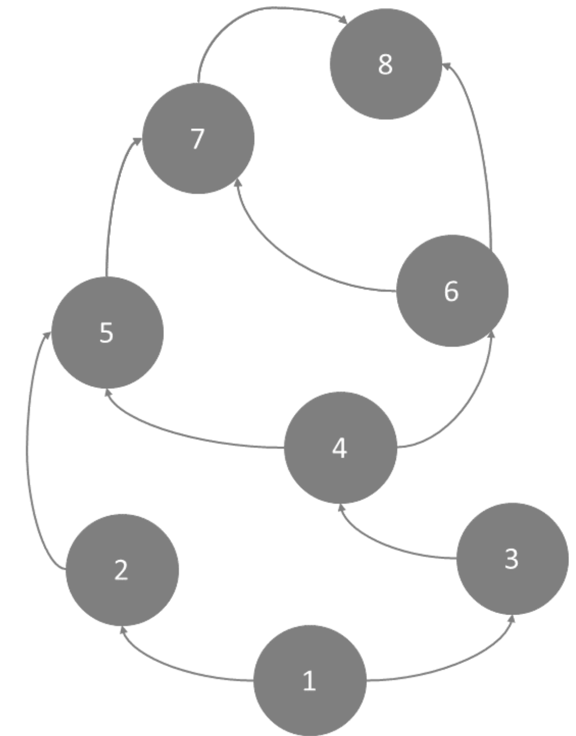

Fig 1. Task Dependency in AMT

HPX – ParalleX Execution model

- ParalleX model addresses the critical bottlenecks of exascale HPC systems like Starvation, Latency, Overheads and Contention
- HPX - allows wait-free asynchronous parallel programming
    - Active Global Address Space (AGAS)
    - Threading subsystem
    - Lightweight Control Objects synchronization
    - Parcelport

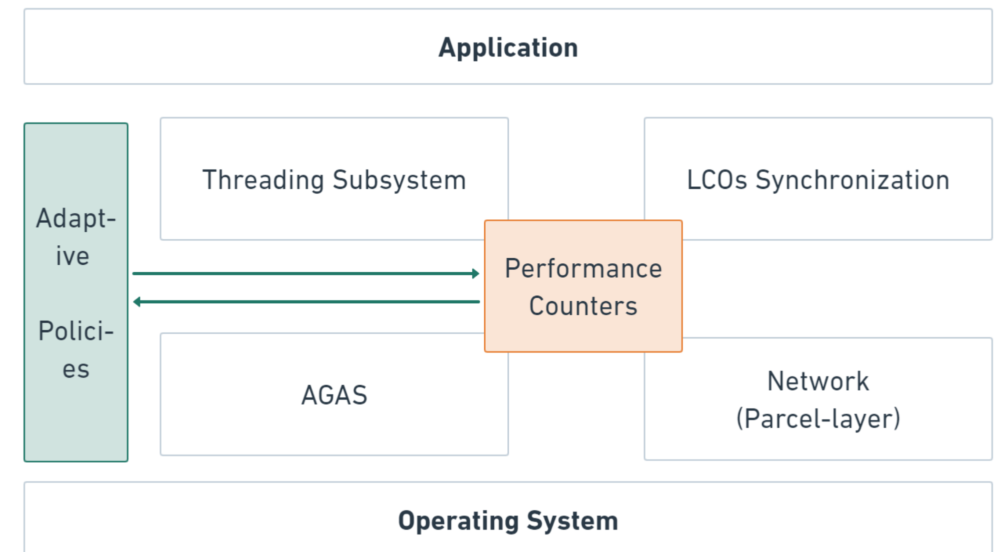

Fig 2. HPX architecture

The Roofline Model

- Provides performance estimation by leveraging the properties of peak computational performance (CP) and I/O Bandwidth (BW)
- Uses Arithmetic Intensity (AI) to glue the two properties
  - Reveals memory bound or compute bound nature
- Peak Attainable performance given by:

$$Attainable\ Performance = min(CP,\ AI \times BW)$$

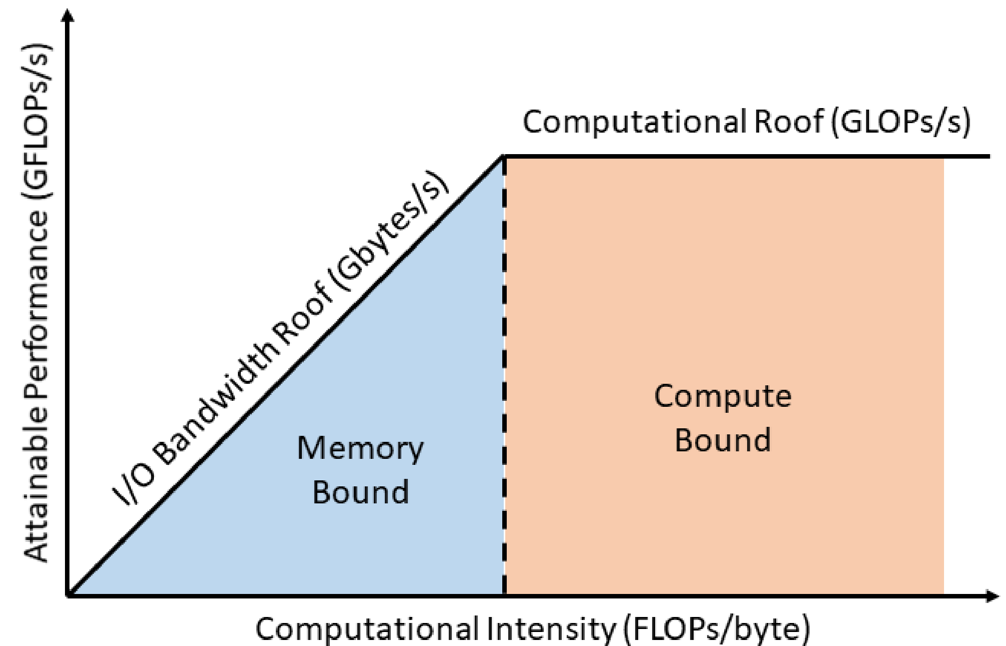

Fig 3. Roofline Model

Benchmarks

1D Stencil Distributed

- Fully distributed 1D heat equation solver.
- Reports total execution time for the application
  - Idea is to observe distributed scaling
- Strong scaling: 1.2 billion stencil points iterating over 100 time steps
- Weak scaling: 480 million stencil points per node iterating over 100 time steps

2D Stencil Shared Memory

- 2D stencil implementing Jacobi method
- Explicit vectorization (Virtual Node Grid) vs auto-vectorization
- Assumes three memory transfers per iteration
  - Arithmetic Intensity (floats): 1/12 LUP/Byte
  - Arithmetic Intensity (doubles): 1/24 LUP/Byte
- Strong scaling (Kernel): Grid size of 8192x131072 iterating over 100 time steps.

Explicit Vectorization – 2D Stencil

- Use NSIMD to add explicit vector types
  - Portability allows single code to work on all architectures!
- HPX “for_each” to parallelize each iteration
- Custom Grid class to abstract data layout

```
template <typename Container>
void stencil_update(array_t<Container>& U,
                    size_t ny,size_t t)
{
  Grid<Container>& curr = U[t % 2];
  Grid<Container>& next = U[(t + 1) % 2];

  size_t row_length = curr.row_size();

  #pragma unroll
  for (size_t nx = 1; nx < row_length-1; ++nx)
  {
  // Stencil operation
  next.in(nx, ny) = (curr.in(nx-1, ny)
                    + curr.in(nx+1, ny)
                    + curr.in(nx, ny-1)
                    + curr.in(nx, ny+1))
                    * 0.25f;
}
```

Listing 1. 2D stencil kernel

```
// Maintain the halo in case of simd
if (std::is_same<
        typename Container::value_type,
        nsimd::pack<
                typename get_type<
                typename Container::value_type
                >::type>
        >::value)
  helper<Container>::shuffle(next, ny);
}

// Call to stencil_update
hpx::util::high_resolution_timer t;
for (size_t t = 0; t < steps; ++t)
{
  hpx::parallel::for_each(
      policy, begin(range), end(range),
      [&U, t] (size_t i)
      {
      stencil_update<Container>(U, i, t);
      });
}
t.elapsed();
```

Listing 2. Halo update and HPX parallel for loop

System Setup

- **Arm based:** Fujitsu FX1000 A64FX, Marvell ThunderX2 and HiSilicon Hi1616; **x86 baseline:** Intel Xeon E5-2660 v3

- Compiler: GNU GCC
  - Arm compilers (GCC, Arm HPC compiler, Fujitsu Compiler) implement SVE types by utilizing __*sizeless_struct*
  - __*sizeless_struct* allows analysis of SVE type and length at runtime
  - __*sizeless_struct* doesn't work with custom classes unless class defined as __*sizeless_struct*!
  - Bad for us! We use custom grid class type and STL features!
  - Only GCC allows us to pass SVE length at compile time. Reduces portability but allows code to compile!

- Measured memory bandwidth using STREAM COPY benchmark for 2D stencil application. (Array size: 120 million)
  - Stencil codes are known to be memory bound
  - Can compute expected peak using computed memory bandwidth (Roofline Model!)

Results

1D Stencil Distributed

- Strong scaling:
  - Xeon E5 and A64FX shows almost linear scaling
  - Very low network performance for Hi1616
- Weak scaling:
  - No visible increase in execution times for Xeon E5 and A64FX
  - Significant increase in execution time for Hi1616

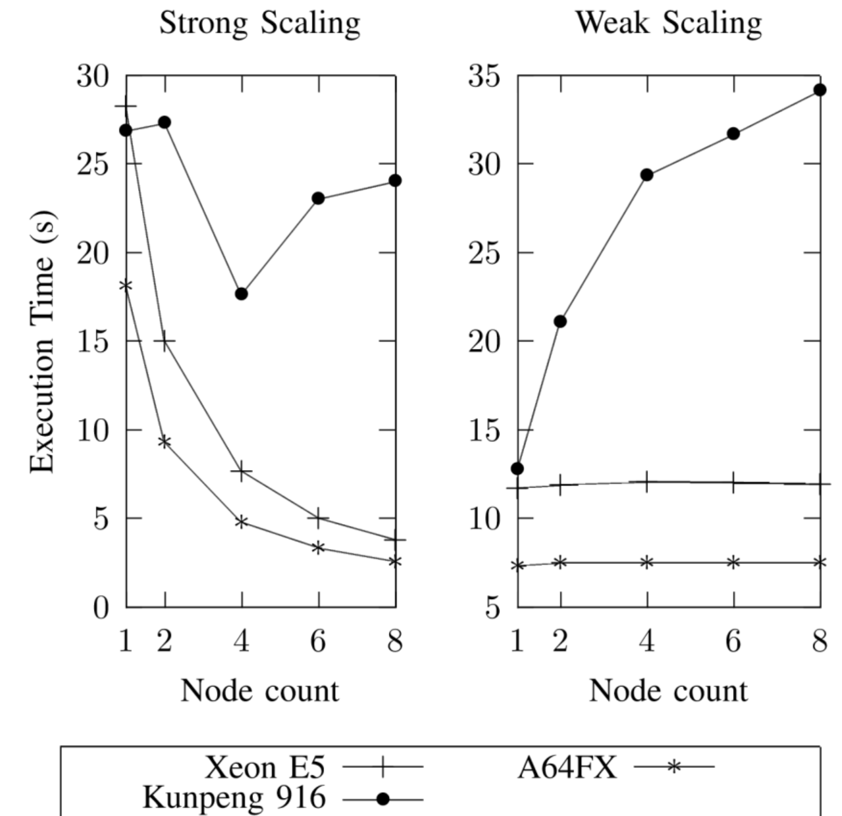

Fig 4. 1D stencil distributed performance

STREAM COPY Results (Array size: 120 million)

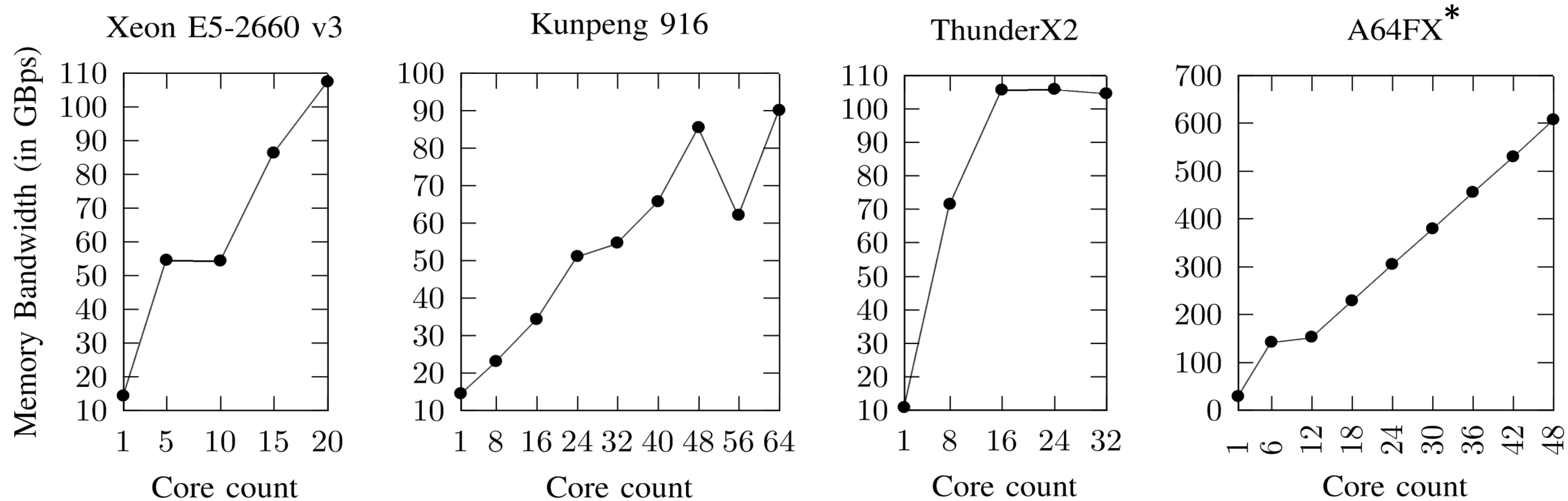

Fig 5 Memory bandwidth results using STREAM COPY benchmark.

*Higher memory bandwidth results are possible with A64FX using special cache initialization techniques. We do not add techniques that can't be added to the 2D stencil code. This ensures correct computation wrt optimal results

2D stencil – Intel Xeon E5 and HiSilicon Hi1616

- Xeon E5-2660 v3:
  - Minimal gains with explicit vectorization
  - 40% Less instructions, 75% More cache-misses with explicit vectorization
- HiSilicon Hi1616:
  - 50-80% gains with explicit vectorization
  - Similar instruction counts, 20% Less cache-misses with explicit vectorization
  - Performance decrease going from 32 to 40 and 48 to 56 cores!
    - Partial saturation of a NUMA domain believed to be at fault

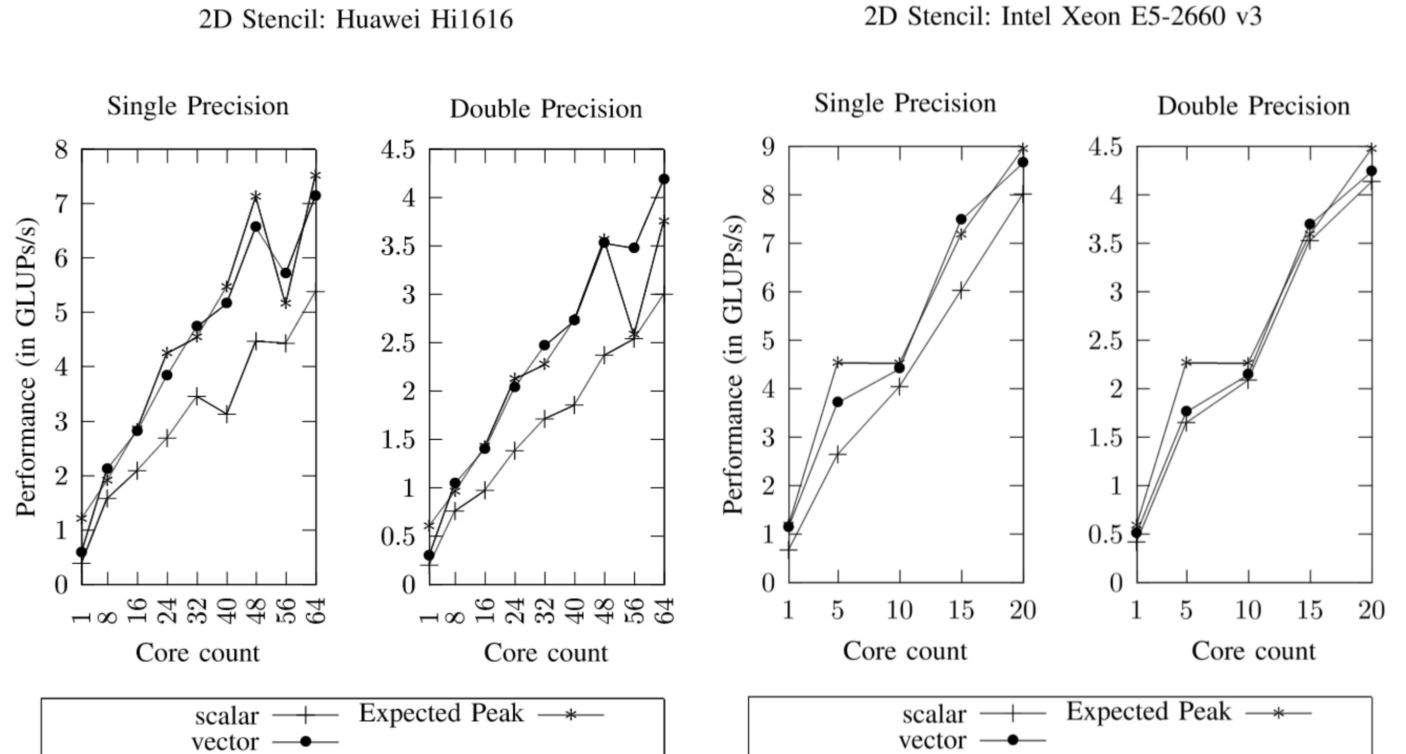

Fig 6. Results for Hi1616 and Xeon E5

2D stencil – Fujitsu A64FX and Marvell ThunderX2

- Large caches results in inherent cache-blocking
  - AI (floats): 1/12 (Min) -> 1/8 LUP/Byte (Max)
  - AI (doubles): 1/24 (Min) -> 1/16 LUP/Byte (Max)
- Fujitsu A64FX:
  - Minimal gains with explicit vectorization
  - 20% more instructions, 16% less backend stalls with explicit vectorization
- Marvell ThunderX2
  - Upwards of 50% gains with explicit vectorization
  - 10% more instructions, 60% (floats) and 15% (doubles) less backend stalls with explicit vectorization

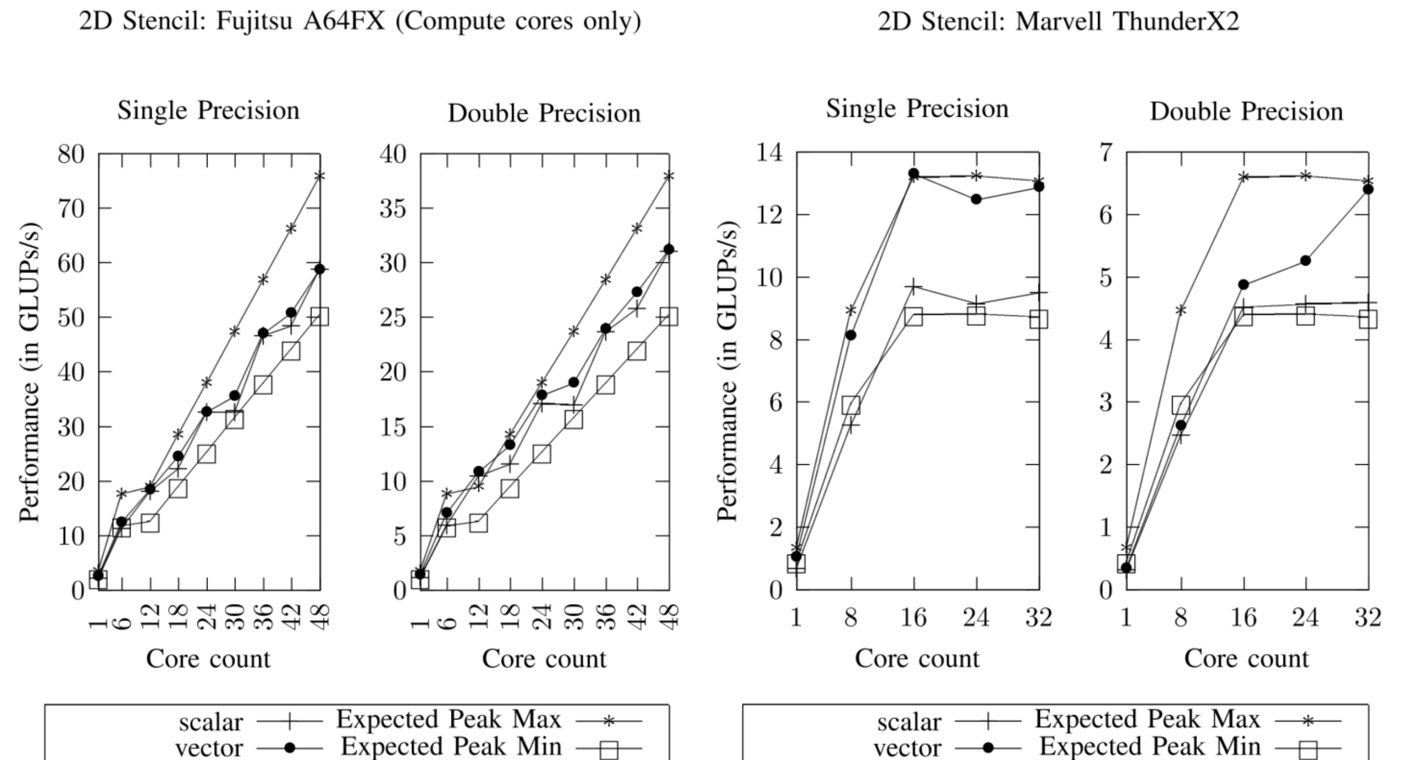

Fig 7. Results for A64FX and ThunderX2

Conclusion

- Porting is mostly straightforward
  - Further work required to integrate custom containers to work with __*sizeless_struct*!
- General performance as good or better than x86
- 1D stencil results proves easy distributed scaling on Arm
  - Poor network performance to be blamed for Hi1616
- Large caches allowed ~50% excess performance due to inherent cache-blocking in 2D stencil
- Minimal gains in performance for Xeon E5 and A64FX using explicit vectorization
- Significant gains with ThunderX2 and Hi1616

Questions?