Heidelberg Institute for Theoretical Studies

HITS

13th IEEE International Workshop on High Performance Computational Biology (HiCOMB 2014)

Efficient Computation of the Phylogenetic Likelihood Function on the Intel MIC Architecture

Alexey M. Kozlov, Christian Goll and Alexandros Stamatakis

Exelixis Lab
HITS gGmbH
Heidelberg, Germany

May 19, 2014

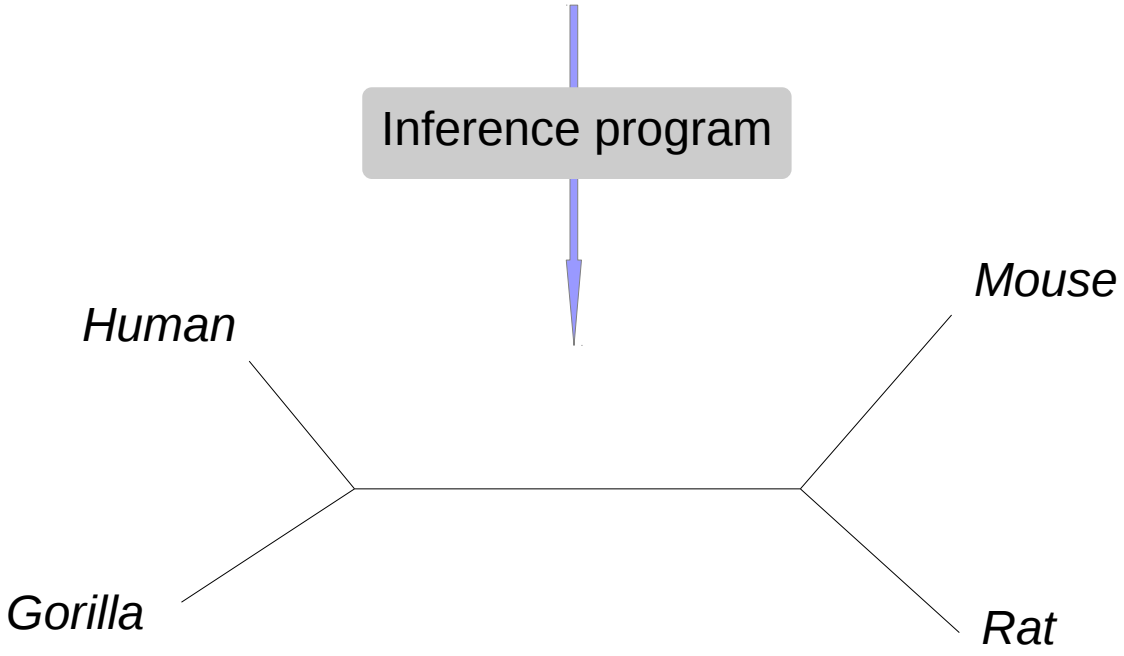
Outline

- Introduction
- Implementation
  - Porting of likelihood kernels
  - Integration to ExaML
- Experiments
- Conclusion & Future work

Phylogenetics

```
Human    ACGTACGTAACT
Gorilla  ACG-ACGTTACT
Mouse    ACGGA--TCAGT
Rat      ACGAA--TCTGT
```

MSA



Phylogenetic tree

Phylogenetic inference methods

- Neighbor Joining
- Parsimony Criterion

} "fast & dirty"

- Maximum Likelihood (ML)
- Bayesian Inference

} "slow & precise"

rely on tree LH calculation

Phylogenetic tools by Exelixis Lab

- Highly optimized codes for phylogenetic inference (ML and Bayesian)
- Likelihood evaluation is *the* bottleneck
  - Memory-intensive
  - >90% execution time
- Previous work:
  - SSE3/AVX + OpenMP/Pthreads + MPI
  - FPGA
  - GPU

Intel Xeon Phi (aka Intel MIC)

- Hardware
  - PCIe extention card
  - ~60 Pentium (P54C) cores
  - 512b SIMD
- Software
  - Intel C compiler
  - Pthreads, OpenMP
  - MPI
- Availability
  - Several supercomputers from Top500 list
  - Coming soon: SuperMUC (Top500 #10, Munich, Germany)

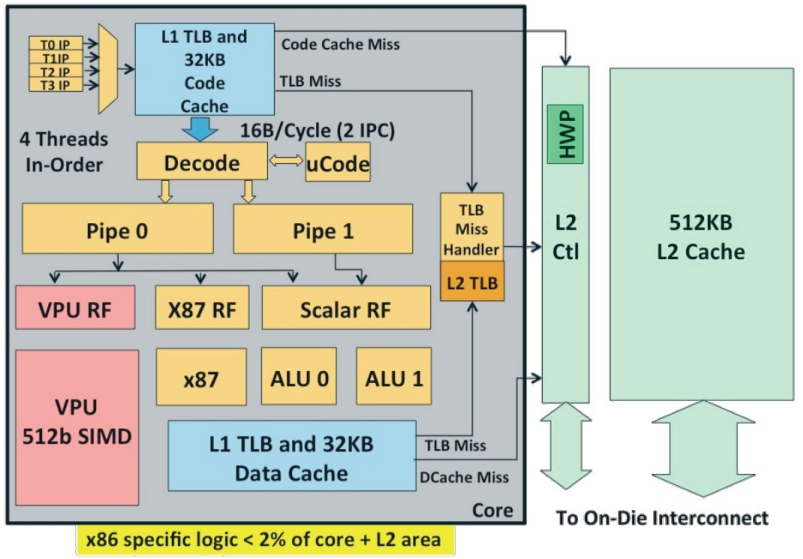
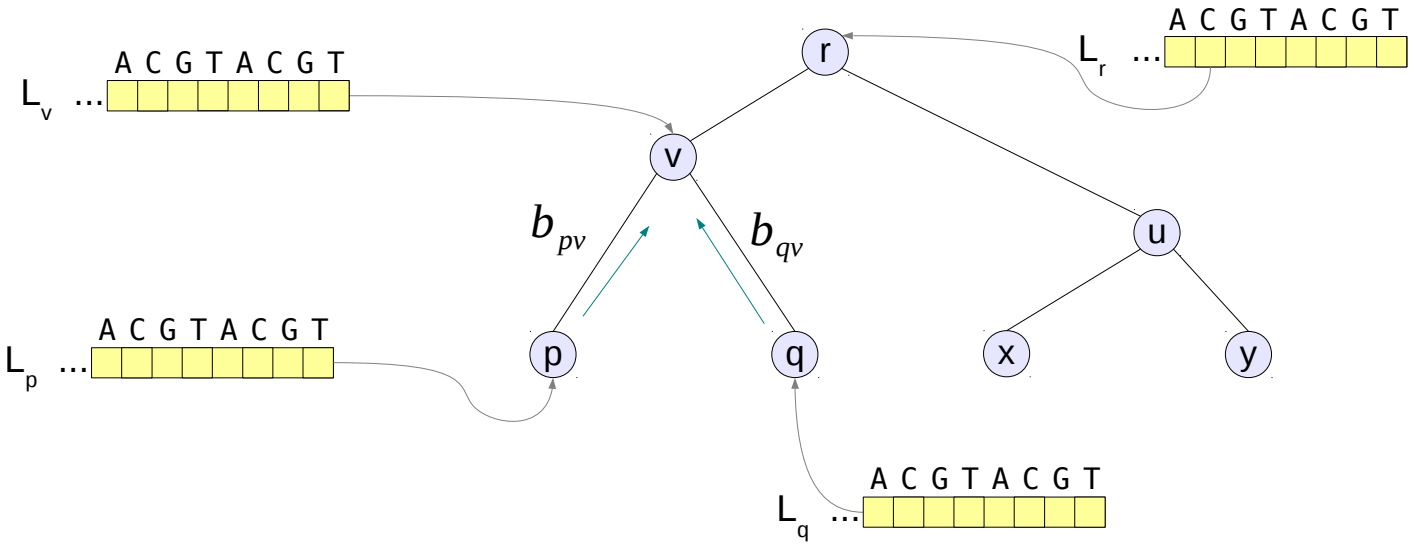


Image source:
http://semiaccurate.com/2012/08/28/intel-details-knights-corner-architecture-at-long-last/intel xeon phi core/

Outline

- Introduction
- Implementation
  - Porting of likelihood kernels
  - Integration to ExaML
- Experiments
- Conclusion & Future work

Likelihood kernels



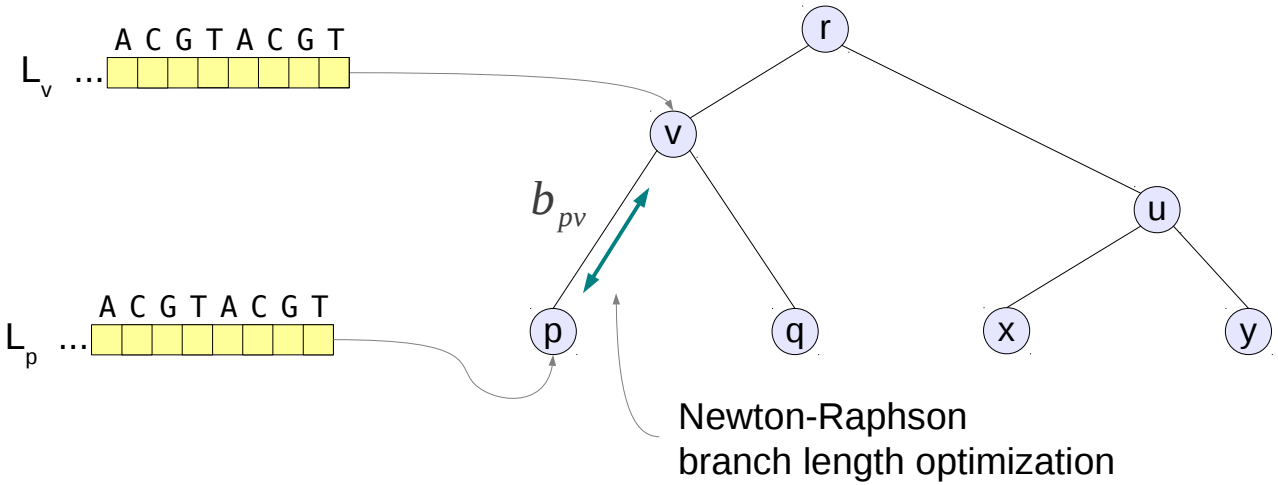
- `evaluateNode()`

$$L_v(i) = \sum_{j=1}^{4} P_{ij}(b_{pv}) \cdot L_p(j) \times \sum_{j=1}^{4} P_{ij}(b_{qv}) \cdot L_q(j), \quad i=1\ldots4$$

- `evaluateRoot()`

$$LH = \sum_{j=1}^{4} \pi_j \cdot L_r(j)$$

Likelihood kernels (2)



- `derivativeInit()`

$$S(i) = L_p(i) * L_v(i), \quad i = 1 \ldots 4$$

- `derivativeCore()`

$$\frac{dL(b_{pv})}{db_{pv}} = \sum_{i=1}^{4} \lambda_i \cdot e^{\lambda_i \cdot b_{pv}} \cdot S(i) \qquad \frac{d^2 L(b_{pv})}{db_{pv}^2} = \sum_{i=1}^{4} \lambda_i^2 \cdot e^{\lambda_i \cdot b_{pv}} \cdot S(i)$$

Kernel tuning: Loop vectorization

- Vector unit: 512b = 16 SP = 8 DP values
- Full utilization is critical for high performance!
- Loop vectorization prerequisites:
  - I/O arrays are aligned to 64-byte boundary
  - I/O arrays are independent
  - Ideally: unit-stride access

Kernel tuning: Loop vectorization (2)

- Kernel vectorization
  - evaluateRoot, derivativeInit: memalign+pragmas only
  - evaluateNode: manual scatter, loop fusion
  - derivativeCore: site blocking

```
#pragma ivdep
#pragma vector aligned
for (int i = 0; i < 16; i++) {
 prod[i] = left[i] * right[i];
}
```

Arrays are independent

Arrays are properly aligned

Example: A perfectly vectorized loop

C code

```
#pragma ivdep
#pragma vector aligned
for (int i = 0; i < 16; i++) {
 prod[i] = left[i] * right[i];
}
```

Assembly

```
vmovapd (%rsp,%r10,1), %k0, %zmm0
vmovapd 0x40(%rsp,%r10,1), %k0, %zmm1

vmulpd (%rcx,%rdi,8), %zmm0, %k0, %zmm2
vmulpd 0x40(%rcx,%rdi,8), %zmm1, %k0, %zmm3

vmovapd %zmm2, %k0, (%rsi,%rdi,8)
vmovapd %zmm3, %k0, 0x40(%rsi,%rdi,8)
```

Kernel tuning: Data prefetching

- All 4 kernels read large input arrays from memory
- Prefetching: cache data in advance to hide memory access latency
- Despite automatic prefetching being implemented in Phi, manual tuning led to significant improvement

```
#pragma ivdep
#pragma vector aligned
#pragma prefetch left:0:2
#pragma prefetch left:1:8
for (int i = 0; i < 16; i++) {
 prod[i] = left[i] * right[i];
}
```

Prefetch to L1: 2 iterations ahead

Prefetch to L2: 8 iterations ahead

Kernel tuning: Streaming stores

- `evaluateNode()` and `derivativeInit()` write large output arrays into memory
- Avoid loading original memory content into cache prior to overwriting
- Possible if writing the whole cache line (64B)
- Usage can be enforced by placing a pragma hint:

```
#pragma ivdep
#pragma vector nontemporal
for (int i = 0; i < 16; i++) {
 prod[i] = left[i] * right[i];
}
```

Outline

- Introduction
- Implementation
  - Porting of likelihood kernels
  - Integration to ExaML
- Experiments
- Conclusion & Future work

ExaML on MIC

- ExaML
  - ML inference tool for extra large datasets
  - Cluster support (MPI)
  - Currently used for analyzing data of the 1KITE project (http://www.1kite.org)
- Experimental MIC implementation
  - DNA data
  - Standard model of rate heterogeneity
  - No support for memory saving techniques

Native vs. Offload mode

Offload

```
double a[100];
double sum = 0.0;
for (int i=0; i<100; i++)
      a[i] = i * 1.5;
```

Host CPU

Offloading runtime call

```
#pragma offload
      in(a: length(100))
      out(sum)
{
      for (int i=0; i<100; i++)
        sum += a[i];
}
```

Xeon Phi

Offloading runtime sync

```
printf("sum = %.5f", sum);
```

Host CPU

Native

```
double a[100];
double sum = 0.0;
for (int i=0; i<100; i++)
      a[i] = i * 1.5;
```

Xeon Phi

```
for (int i=0; i<100; i++)
      sum += a[i];
```

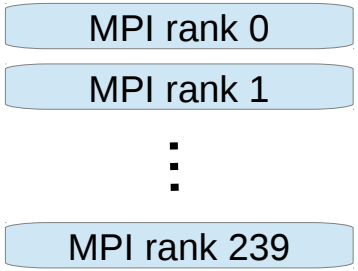
```
printf("sum = %.5f", sum);
```

Execution mode for ExaML

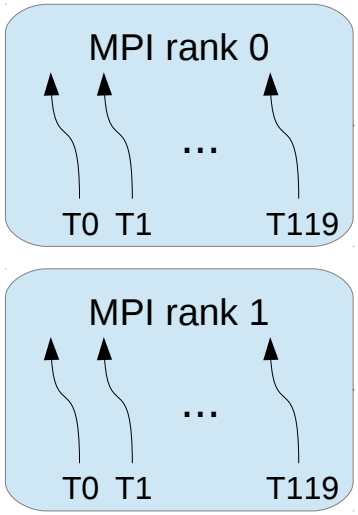
- Offload
  - \+ Only kernels must be ported to MIC
  - – Calling offload runtime has significant latency
    - 1000s kernel calls per second → Show-stopper!
- Native
  - \+ Kernel invocations are simple function calls
  - \+ Code simplicity
    - e.g. no MIC-specific memory management
  - – Whole application must be ported

Thread-level parallelism

- ExaML-CPU: MPI-only
  - One MPI rank per core
  - Works fine on CPUs (12-16 ranks)
  - But: becomes prohibitively slow on MIC (120-240 ranks)



- ExaML-MIC: Hybrid MPI/OpenMP
  - One or few MPI rank(s) per MIC card
  - Within each rank, kernel loops are parallelized with OpenMP



Outline

- Introduction
- Implementation
  - Porting of likelihood kernels
  - Integration to ExaML
- Experiments
- Conclusion & Future work

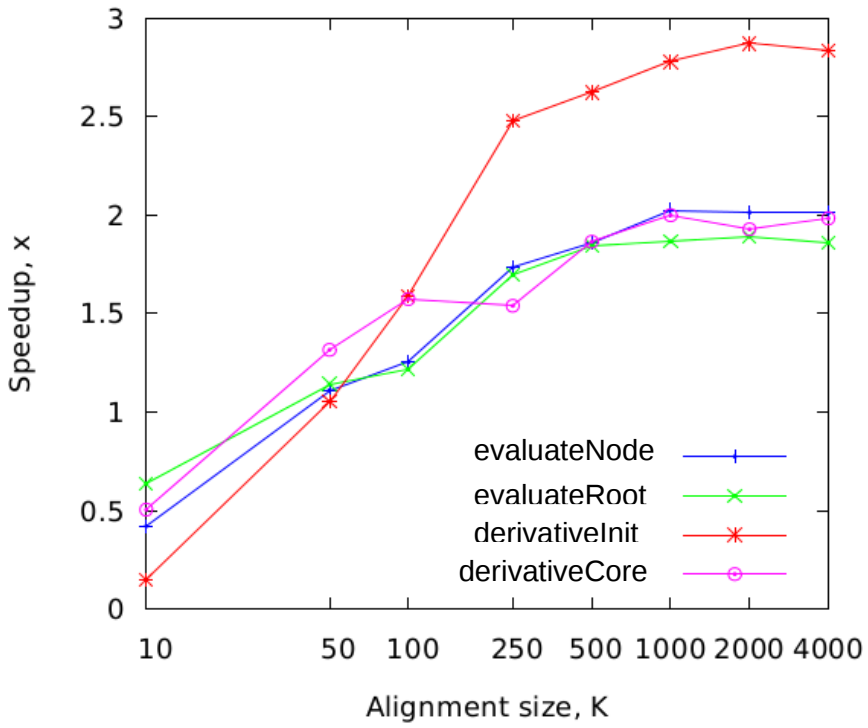
Experimental setup

- MIC card: Intel Xeon Phi 5110P
  - 60c @ 1Ghz, 8GB DDR5 RAM
  - 1074 GFLOPS DP
  - 225 W TDP
- Reference system: 2x Intel Xeon E5-2680
  - 16c @ 2.70 GHz, 32 GB RAM
  - 346 GFLOPS DP
  - 260 W TDP
- Test datasets:
  - Simulated alignments, 15 taxa, 10K ÷ 4M sites

Kernel performance

Individual kernel speedups

(1x5110P vs. 2xE5-2680)



- Better speedups for larger alignments
- Simplified `derivativeInit()` kernel:

```
for (int i = 0; i < n; i++)
{
  for (int j = 0; j < 16; j++)
  {
    prod[i*16+j] = left[i*16+j] * right[i*16+j];
  }
}
```

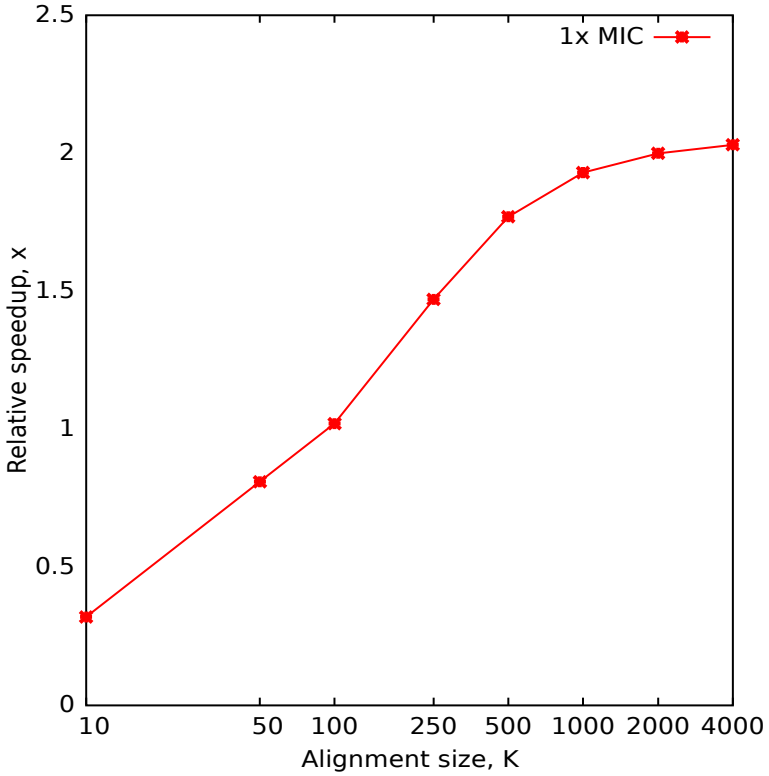
Trivial multiplication → near-optimal speedup (2.9x)

- Other kernels
  - More complicated computation
  - Speedup: 1.8x – 2.0x

Application performance

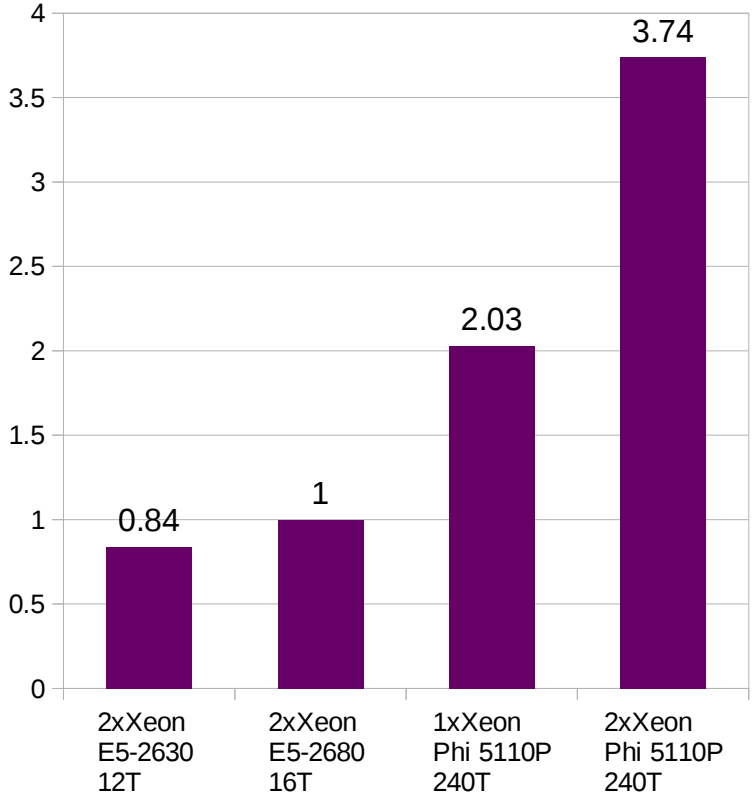
Whole application speedup

(1x5110P vs. 2xE5-2680)



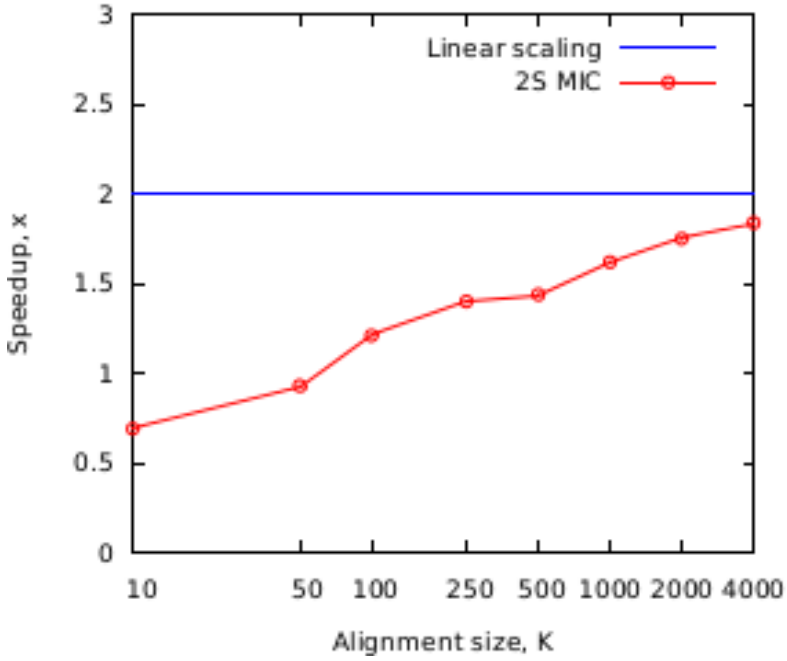
ExaML maximum speedup

(4000K alignment sites)



Scalability with MPI

- Tested with 2 cards on the same node
- MPI `AllReduce` latency:
  - 20 μs Phi-to-Phi (PCIe)
  - 5 μs Host-to-Host (IB)
- Max. speedup with 2 cards: 1.84x
- Better MPI implementations are under way
  - e.g. MVAPICH2



Outline

- Introduction
- Implementation
  - Porting of likelihood kernels
  - Integration to ExaML
- Experiments
- Conclusion & Future work

Conclusions

- Results:
  - + Optimized PLF kernels for MIC implemented
  - + Integrated with state-of-the-art tree inference program
  - + Significant speedups to modern CPUs shown
- Problems:
  - – High number of cores → need for large alignments
  - – Limited on-card memory (≤16 GB)
  - – Suboptimal MPI performance

Future work

- Support protein data
- Improve OpenMP parallelization
- Production-level integration with:
  - ExaML, ExaBayes, PLL
- Analyze data from the next phase of 1KITE project
  - using MIC cluster of SuperMUC

Q & A

Thanks for your attention!

Questions?