Owl's Nest – Past, Present, and Future

Dr. Axel Kohlmeyer

Research Professor, Dept. of Mathematics
Associate Director, ICMS
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

The HPC Team

- Formally research faculty in math department
- Offices in SERC building 7th floor:
  - Ershaad Basheer (Engineering, Python, Linux)
  - Richard Berger (Computer Science, Mechatronics)
  - Axel Kohlmeyer (Computational Chemistry, Linux, Scientific Software Development, HPC Education)
- With technical support from Computer Services:
  - Raymond Lauff, Scott Birl, Lonny Dash
- Preferred point of contact: hpc@temple.edu

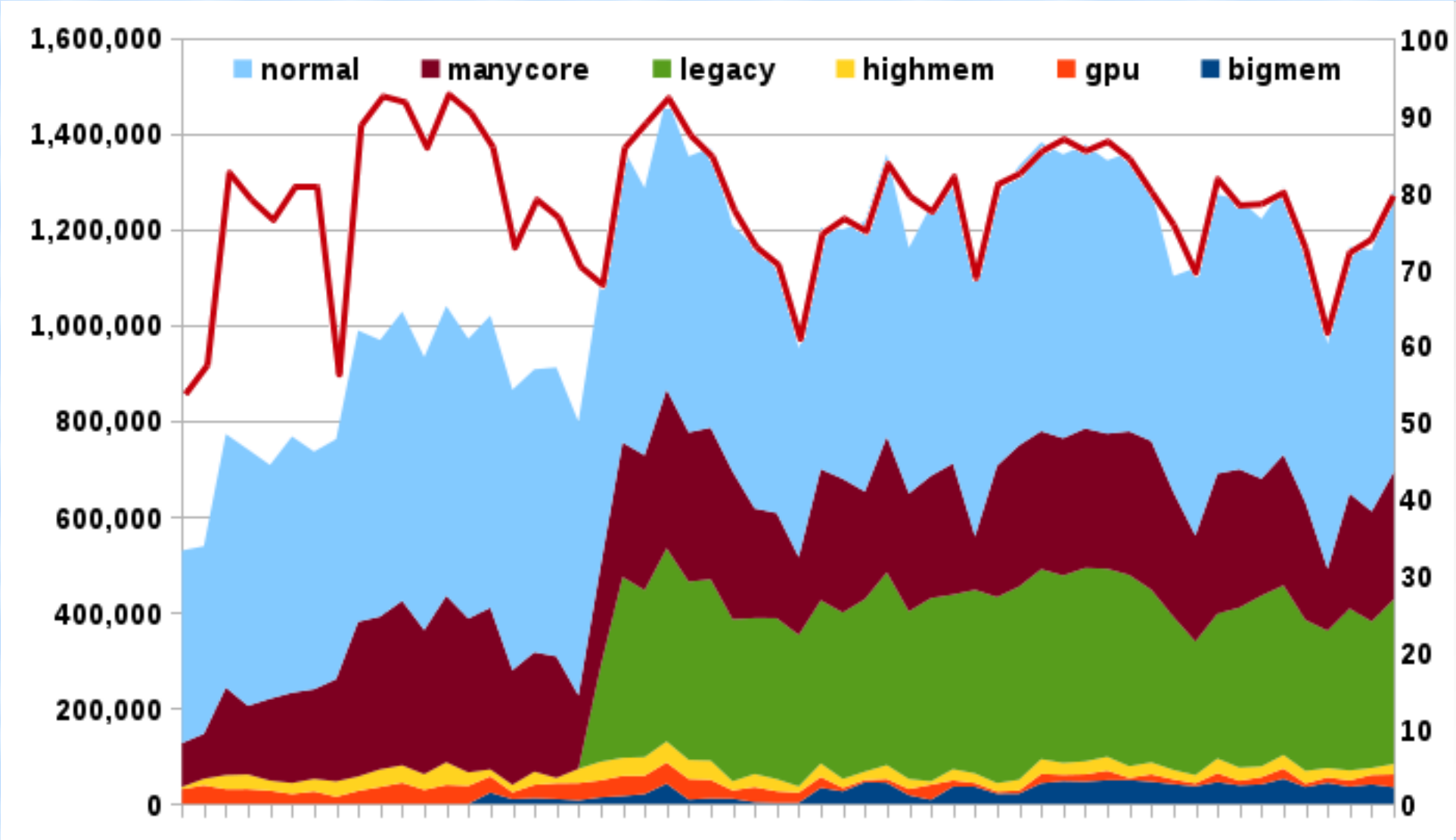
2009 – A Boost for HPC @ Temple

- Klein group moves from U Penn to Temple
- Institute for Computational Molecular Science
- Includes moving 72-node HPC cluster for ICMS
- MRI proposal to NSF for central HPC resource at Temple under the lead of CIS (J. Wu, Y. Shi)

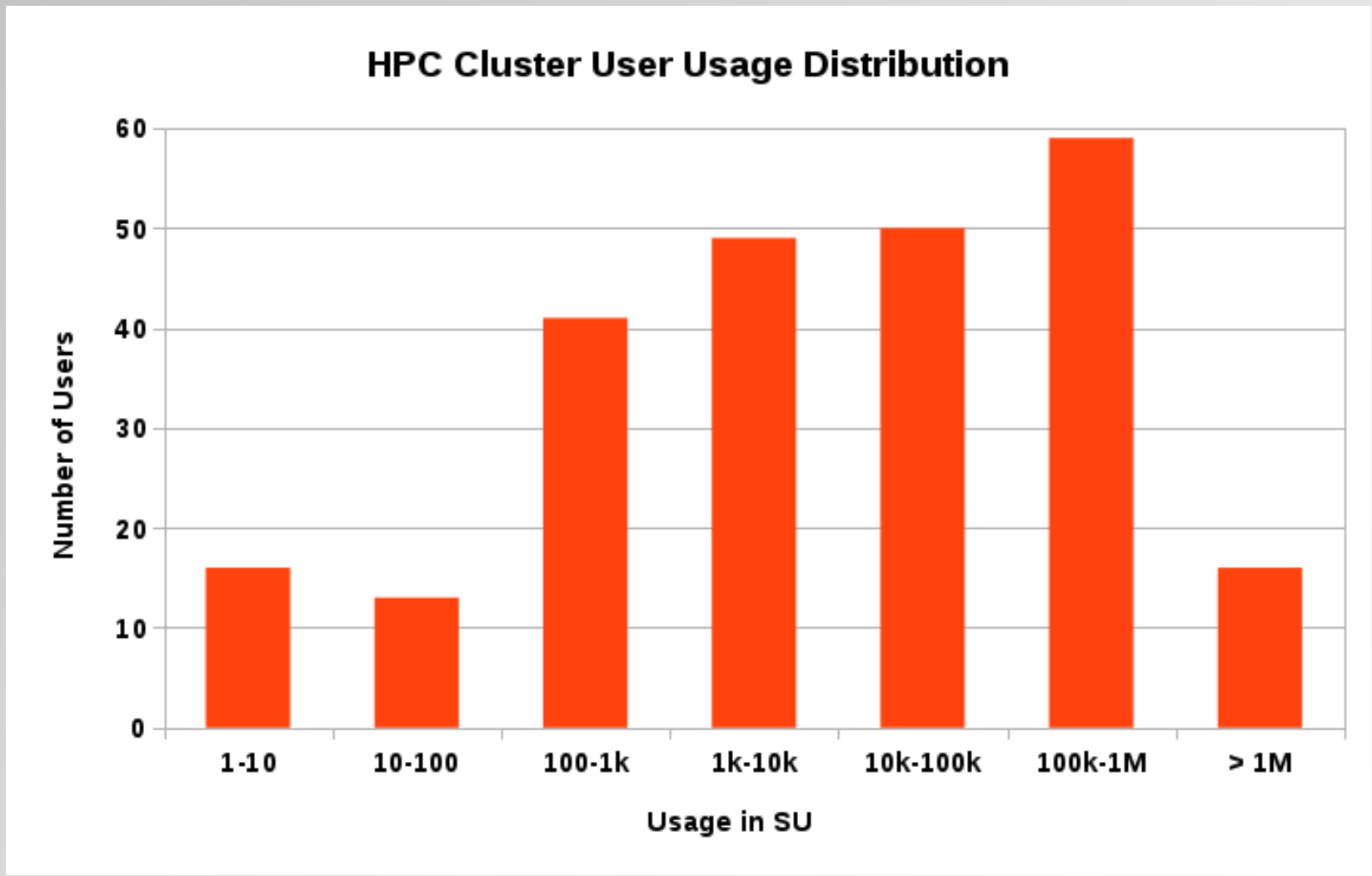
Since 2011: Owl's Nest

- Available since spring 2011
- Multiple funding sources (NSF, CST, Startups)
- Cluster of clusters → diverse hardware sections with shared front end, accounts and storage
- Hosted in Temple's JDC with support from CS
- 1x hardware expansion & ICMS cluster added
- Very reliable, high to good utilization
- Tuned for message passing parallel codes

Owl's Nest Utilization



270 Active Users



Adapting to Changing Needs

- Upgrade of front ends → bigmem queue
- Deployment of “compute” server for non-batch usage, various single node licensed software
- New MRI proposal January 2016:
  - New scalable storage systems
  - Very large memory nodes for big data, genomics
  - Improvements to aging infastructure
  - Modest upgrade for oldest legacy hardware
- Collaboration with PSC on Campus Bridging

Graduate Level HPC Classes

- Math department and HPC team currently work on establishing graduate level classes with HPC related content. Topics are:
  - Introduction to computer programming for scientists (Python, C/C++, Fortran)
  - Introduction to HPC technologies (Hardware, System Software, Deployment and Configuration)
  - Efficient Algorithms for Scientific Computing
  - Programming Parallel Algorithms
- Long term goal: offer an HPC degree

Special Expertise of the HPC Team

- Operating HPC hardware (but rarely touching it)
- Consulting on optimal hardware configurations
- HPC education and training (local, remote)
- Optimizing software and workflows
- Connecting Temple PIs with supercomputing resources (e.g. XSEDE, INCITE)
- Collaborating on porting, optimizing and parallelizing scientific applications
  (this is what we like to do the most)

Optimization Example 1

- LAMMPS MD code currently uses an approximation for erfc(x) in coulomb forces
  → only accurate to single precision FP math
  → using erfc(x) from libm => run takes 2x time
- Accurate erfc(x) computed from scaled error function complement erfcx(x) = exp(x*x)*erfc(x)
  → erfc(x) = erfcx(x)*exp(-x*x)
  → need to compute exp(-x*x) for derivative
- Use fast and precise approximation for erfcx(x)
- Precompute exp(-x*x) using fast exp2(x) code

Fast Implementation of exp()

- Range reduction: $x = f + n;\ \ n \in \mathbb{Z}, -0.5 \leq f < 0.5$

$$2^x = 2^{f+n} = 2^f \cdot 2^n$$

- Get 2^n from setting IEEE-754 exponent: zero mantissa bits (=1), exponent is *n* + 1023
- Padé Approximation: $2^f = 1.0 + \left(\frac{2f \cdot P_3(f^2)}{P_3(f^2) + Q_3(f^2)} \right)$
- Unroll & interleave $P_3(f^2)$ and $Q_3(f^2)$ evaluation
- Store coefficients for P/Q at aligned address
- exp(*x*) = exp2($\log_2(e)$**x*)

Optimization Example 2

- From the “HPC Helpdesk”: hpc@temple.edu
  User requests access to HPC resource because his self-written program needs too much memory and runs too slow on desktop
- Next, the user asks for parallel programming assistance to handle large matrices
- Application is one file with ~1000 lines C code
  => could be perfect showcase for a “minimum effort” optimization and parallelization study

  => “The game is afoot...”

Optimization 1: Reduce Memory

- The by far most time consuming step is the calculation of a “connection matrix”
- The matrix elements are either 1 (if two nodes are connected) or 0 (if the are not connected)
- Storage element was **`unsigned long int`**

  => use **`char`** instead

  => 4x (32-bit) to 8x (64-bit) memory savings

  => 1.5-2x performance increase

Optimization 2: Compiler

- The reference executable was compiled with gcc using default settings, i.e. no optimization
- Using compiler optimizations leads to significant performance increase
- Compiler optimization can be improved through using **`const`** qualifiers in the code wherever possible and local code changes
- Hide complex data types with **`typedef`**

  => 2.5 – 3.5x speedup

Optimization 3: Parallelization

- The construction of the connection matrix has no data dependencies => multi-threading
- Using OpenMP requires only adding one directive and a little bit of code reorganization
- Speedup going from serial to 2 threads: 1.5x
- Speedup levels out at 6-8 threads: 2.5x total

  => very little computation, mostly data access
  => performance limited by memory contention
- Total improvement: 8x-12x with 8 threads

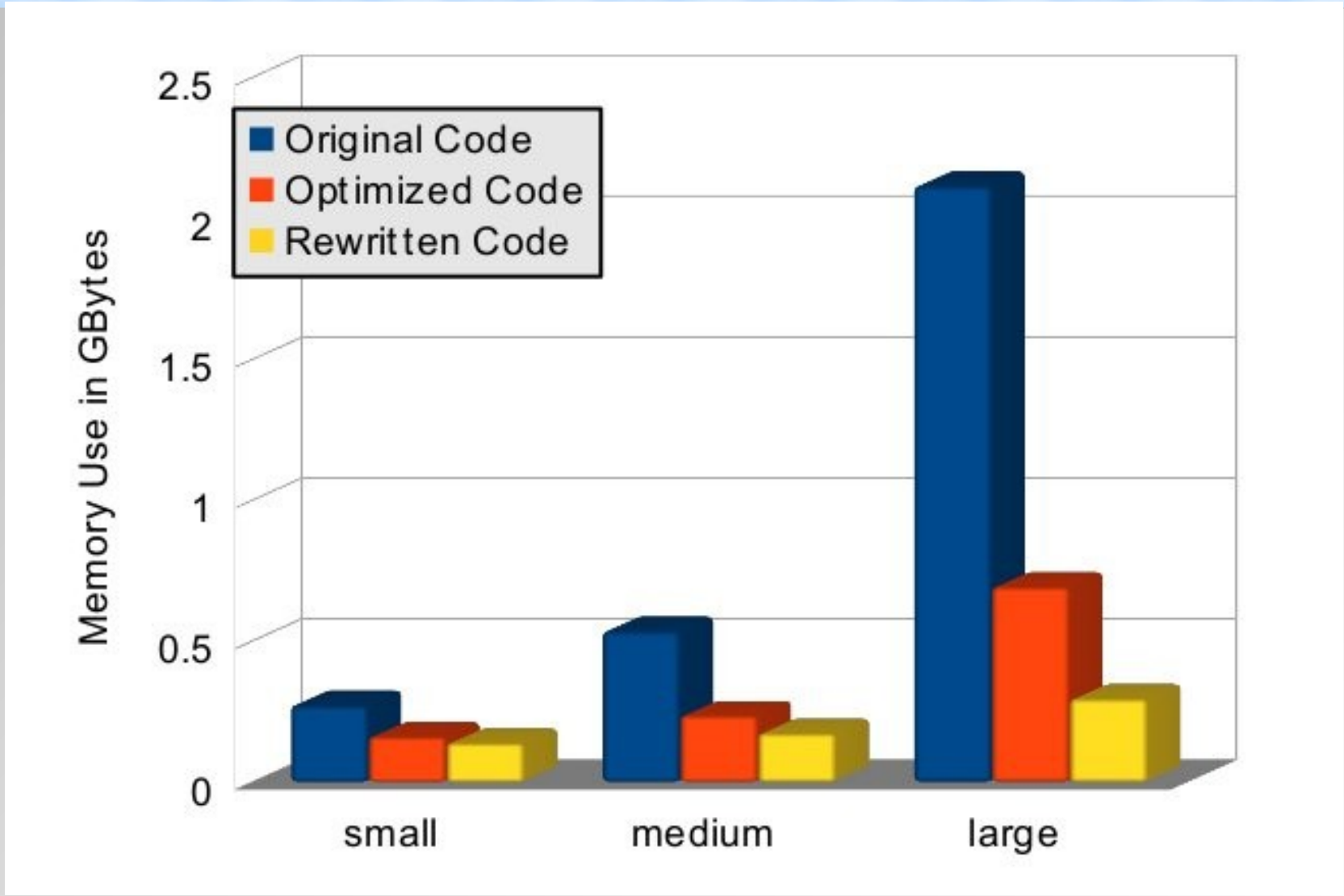
Proper Optimization or: The Power of the Rewrite

- Quick'n'dirty optimizations of T-CLAP resulted in significant improvements in a short time
- More optimization potential with rewrite:
  - Connection matrix information requires only 1 bit
    => reduce storage by another factor of 8 (vs. **`char`**)
  - Network represented by structs and lists of pointers
    => pointers require more storage in 64-bit mode
    => many pointers point to the same data
    => C aliasing rules still require re-reading data
  - Pruning implementation uses memmove() to compact matrix rows => bottleneck for large data

The Rewrite

- Rewrite in C++ (more optimization hints than C)
- Use STL container classes
- std::vector<bool> uses single bit per entry
- Single list of structs for all network nodes,
  all references via index lists (std::vector<int>)
  => no more need to re-read data
- Leave data in place during pruning, maintain
  lists of valid rows and columns instead
- Rewrite piece-by-piece to reproduce original

Memory Usage After Rewrite



Parallel Performance After Rewrite