

High Performance Computing on *COSMOS*

Victor Travieso
COSMOS Parallel Programmer
University of Cambridge

Overview

- What is High Performance Computing
 - Of supercomputers, flops, benchmarks and the real world
- COSMOS facility
 - Dedicated HPC resource for UK cosmologists
- Practical supercomputing
 - Roads to performance and common pitfalls
- Conclusion

High Performance Computing

Why do we need supercomputers :

- Performance is a crucial requirement
 - Limit on the size of the problem that can be studied
 - Level of complexity of a model
 - Time to produce results might be limited – eg. Conference deadline
- “Super-” usually means more resources
 - Larger amount of memory and disk
 - Faster and multiple integrated CPUs
 - Dedicated, special purpose hardware

High Performance Computing

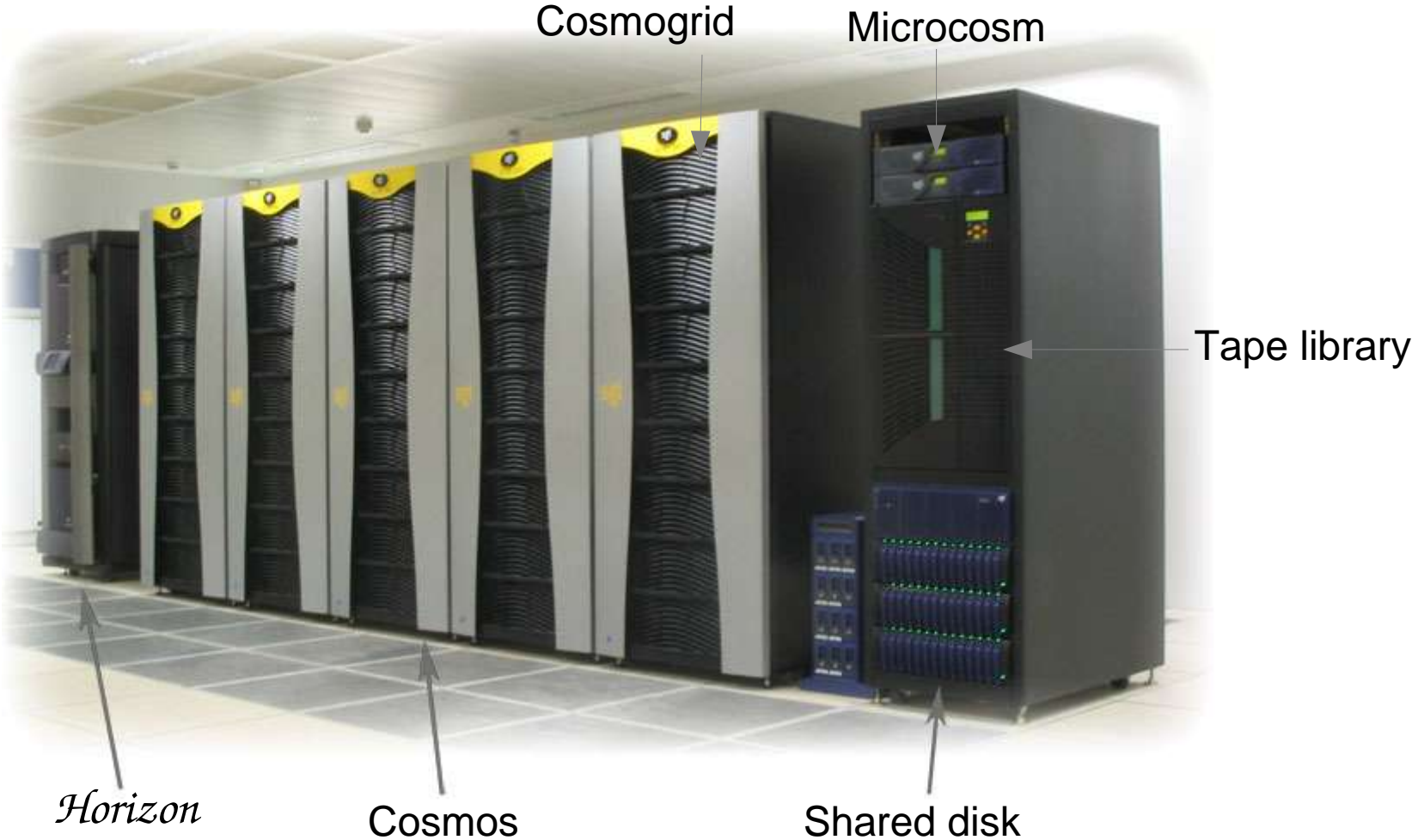
Does the program run super-fast ? :

- Peak performance misleading
 - Flop comparisons are more of a PR stunt
 - Real codes order of magnitude below “peak”
 - Alternative definition: “Performance the machine is guaranteed not to achieve”
- Benchmarks limited usefulness
 - Vendors usually “cook” results
 - Often not comprehensive, read the small print (sometimes useful info!)
- Real applications are what count
 - But might not be able to estimate performance a priori..

High Performance Computing

- “Supercomputers” enable HPC but are not a guarantee
 - Real applications will often benefit but might need some work
 - Codes don't usually have to be rewritten, but some porting & optimization is often needed
 - Performance libraries, optimizing compilers and parallel programmers make the task easier

COSMOS



Cosmos - Altix architecture

- High performance 'cluster' of IA-64 boxes
 - 152 Itanium-II CPUs – 1.3 Ghz, 64-bit
 - modular design – bricks and cables – NUMAflex
- SSI, 152 GB of global shared memory
 - ccNUMA – cache coherent non-uniform memory access
- Low latency, high bandwidth interconnect
 - NUMALink 3 (4) – Same bandwidth as the Origin but ~ 40% faster
- Fast I/O
 - CXFS, XFS filesystems > 200 MB/s

Altix architecture

Benefits of shared memory:

- Supports both parallel programming models
 - threads (OpenMP, posix) and message passing (MPI)
 - OpenMP is easier to do – incremental parallelization
- No need to decompose data or swap it
 - large data sets fit into memory
- Efficient load balancing
 - no need to move data between processes
- More efficient MPI implementation
 - can use 'shortcuts' to speed up communication (single-copy)
- Easier to administer

Altix architecture

- Compute brick in Cosmos

- Memory

 - 2 GB per node

 - 6.4 GB/s

- Numalink4

 - Inside a C-Brick

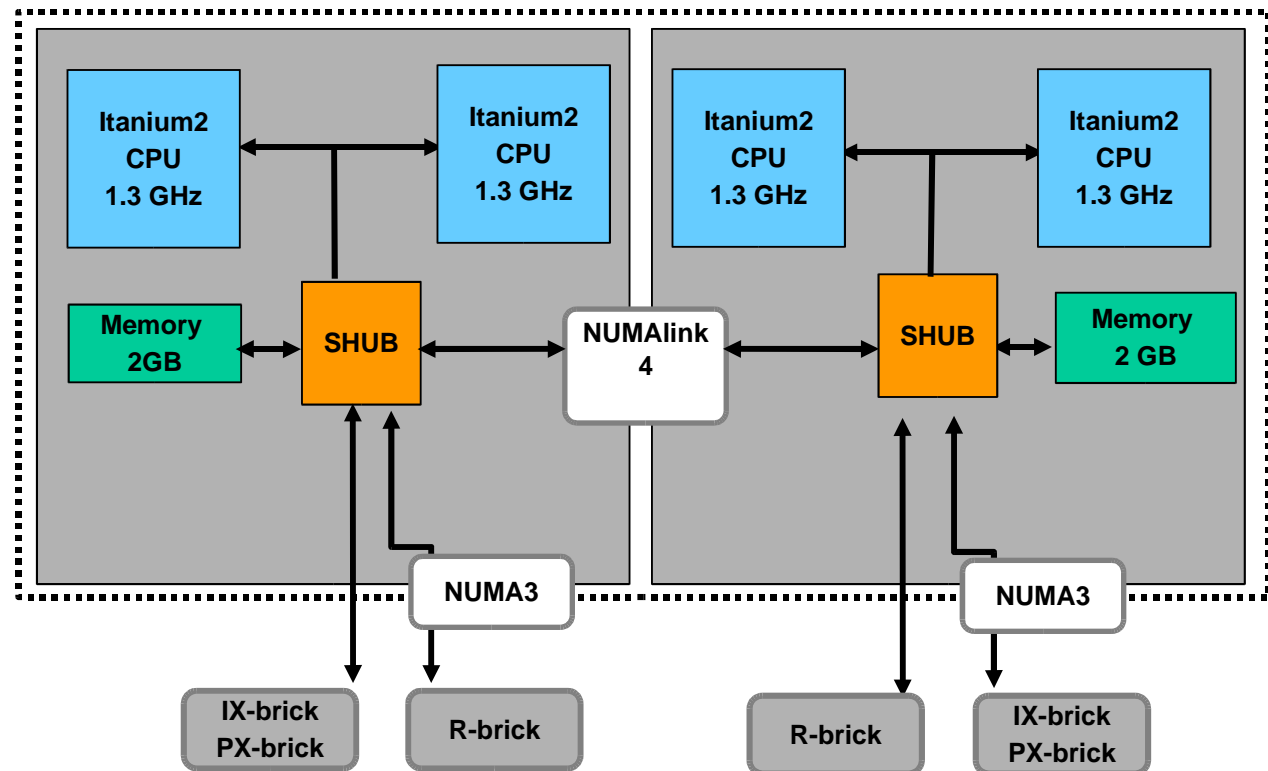
 - 6.4 GB/s

- Numalink 3

 - Inter brick

 - 3.2 GB/s

- Latencies: Local ~100 ns, nn ~ 240 ns + 50 ns per hop (fat tree topology, max hops = 6)



Itanium II processor

Intel's first 64-bit offer for HPC:

- Second iteration of the Itanium - Madison
 - Montecito is the dual core long overdue upgrade..
- Designed and optimized for scientific workloads
- Large and fast on-die cache (including L3)
 - L1I + L1D (16 Kb, integer), L2 (256 Kb, 6 cyc.), L3 (3 Mb, 12 cyc.)
- EPIC architecture – superscalar, up to 6 IPC
- Large amount of resources (registers, issue units, HPW)
- Logic simplified
 - So called 'logic transfer'... responsibility of good instruction selection in the compiler
 - No out of order execution

Itanium II processor

EPIC Architecture:

- Clock speeds are relatively slow (<1.5 GHz)
- Superscalar – up to 6 instructions per cycle
 - up to 2 fma sustainable, i.e. 4 flops per cycle
- Other features for floating point workloads
 - large amount of registers (128 float) and stack engine (reduce function call overhead)
 - support for software pipelining via rotating registers
 - TLB supplemented with Hardware Page Walker – reduce TLB misses via a second level
- But no out of order logic!
 - if no instructions can be issued at the moment (eg. might be waiting for memory) the processor stalls
 - pressure on the compiler to find parallelism and produce appropriate ordering of operations

User's View

- One monolithic system
 - Looks like a large SMP workstation
- Usual GNU/Linux environment
 - Open source development tools (gcc, gdb, emacs, etc.)
 - 100% binary compatible with RedHat or SuSE
 - SGI added software layer : Propack – cxfs, numa, mpt, etc.
- Underlying complexity is hidden to first order
 - But relevant to performance!

User's view

Running jobs:

- Cosmos accommodates a variety of environments
 - environmental variables and module interfaces for instant switching
- Interactive partition and express queue
 - 8 cpus reserved for interactive use – fast response time, code development and testing
- Platform LSF 6.0 for production jobs
 - Batch queues for fair allocation of resources and maximum throughput
 - At worst ~ 8 hours wait time

COSMOS

- **Emphasis on flexibility and ease of use**
 - run in any queue or interactively, any size of you need – cpus, memory, etc.
 - however, it has to accommodate many different jobs and users
 - some mechanisms to protect users from other users and jobs and to ensure every application gets enough resources
- **Missing features on Linux/LSF**
 - proper resource monitoring and accounting
 - memory placement and protection of a job's 'personal' space
 - implemented with our own software - watchdog

Performance

Performance analysis – measure first:

- Goal : reduce program run time
 - Strategies might include: using libraries, code optimizations, parallel processing
- Performance can be measured
 - Execution profile – guide optimization
 - Hardware counters – identify and quantify problems
- Profiling gives you a reliable picture of performance
 - Used to be complicated and lengthy.. now hardware support for it

Ahmdal's Law

Law of diminishing returns:

- General formulation for any performance improvement
 - F_p – Fraction of the program that can be improved by S_p
 - N-Parallelism is a special case with $S_p = N$



Ahmdal's Law

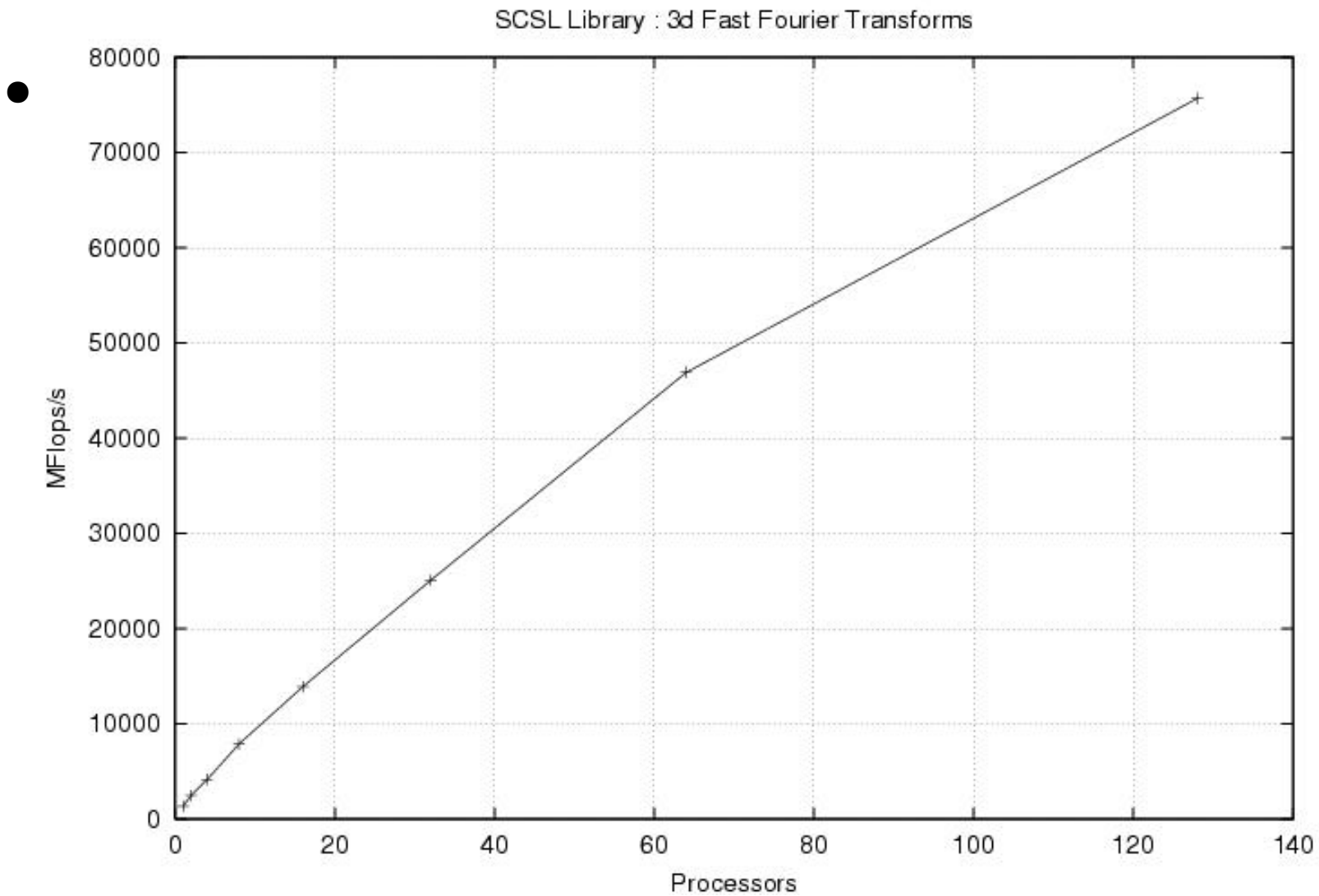
Pitfalls:

- LOS code with 80% of the time on 3D complex FFT
 - Speed up the FFT by 2 – eg. SCSL library, $S \sim 1.6$
 - Parallelize with 8 processors and $S < 3$
- Real code with FFT kernel, but changes gave no performance benefit at all
- Profiling revealed 90% of the time on repacking, 10% on FFTs
 - Figure out Ahmdal's for that one...

Libraries

- Numerical software highly optimized and robust
 - easiest way to get a very good performance - don't reinvent the wheel
 - less portability but big performance gains - if you require portable code customize the Makefile
 - use over freely available/generic routines (lapack, netlib, numerical recipes) when possible
- Most common numerical tasks are included
 - linear algebra (Lapack) implemented fully with the same calling convention - instant migration
 - FFTs - No agreed standard, so migrating needs a bit of work. Parallel enabled through OpenMP
 - MPI libraries (Scalapack) fully implemented
- Parallel routines are as good as they can get in terms of scalability

Libraries



Optimization

Is the implementation making full use of the hardware?

- Assuming the algorithm is well designed...
 - the pipeline might not be delivering instructions every cycle – lots of no-op
 - processor stalls waiting for memory
- Most optimizations have to do with hiding the memory latency
 - memory speed \ll processor speed
 - programs need to make efficient use of the memory hierarchy (caches)
- Compiler might need help to generate optimal code

Optimization

Code transformations:

- Software pipelining in loops
 - Eliminate dependencies and ambiguities, expose parallelism (loop fusion)
- Prefetching
- Cache friendly data access
 - Eg. Column order in Fortran, blocking
- Compiler flags will determine success to a great extent
 - Also one can help it by rewriting portions, directives, breaking up large programs or loops

Optimization

Example - CosmoMC:

- Cosmology MCMC code with MPI parallelism
- 90% of kernel CAMB spent in ode routine 'dverk'
- First optimization attempt – rksuite patch
 - Compiler was doing a mess, O2 faster, etc.
- Second optimization – Latest compiler (Intel 9.0) can get the same speedup with careful selection of flags
 - Selective optimization helps it generate very efficient code for dverk – key is software pipelining, no prefetching, etc..

Parallel programming

When to parallelize:

- There's enough work to compensate for parallel overheads
 - thread synchronization, communications, and time!
- A significant portion of the program can be parallelized
 - Ahmdal's law
- Things to consider:
 - adequate programming model : autoparallelization, OpenMP or MPI ?
 - OpenMP: locality and cache trashing
 - MPI: complexity, communication overhead, load balancing

OpenMP:

Parallel programming

- Very easy to program
 - code directives and incremental parallelization
- Memory placement still important
 - first touch applies, no migration! - poor placement leads to memory bottleneck and kills scalability
- Some hidden dangers: eg. cache trashing

MPI:

- Requires data decomposition and often rewriting the algorithms
 - but it is the most portable and widely used paradigm
- Communication is the bottleneck – and load balancing might be tricky

Conclusion

HPC is more than running code on a supercomputer :

- Achieving an acceptable efficiency requires some awareness of performance issues on part of the programmer
- Some optimization or parallelization effort might be required to squeeze flops
 - Libraries are very efficient and an easy route to multiprocessing, use them
 - Parallel programmers are there to help, use them too!

Conclusion

Further info :

- Contact details:

v.travieso@damtp.cam.ac.uk

cosmos_sys@damtp.cam.ac.uk

- Cosmos webpages

<http://www.damtp.cam.ac.uk/cosmos/>