

Performance Results for Nested Parallelism on Multicore Architectures

David W. Walker and Martin Chorley
Cardiff School of Computer Science

Martyn F. Guest
Advanced Research Computing @ Cardiff

Overview of Talk

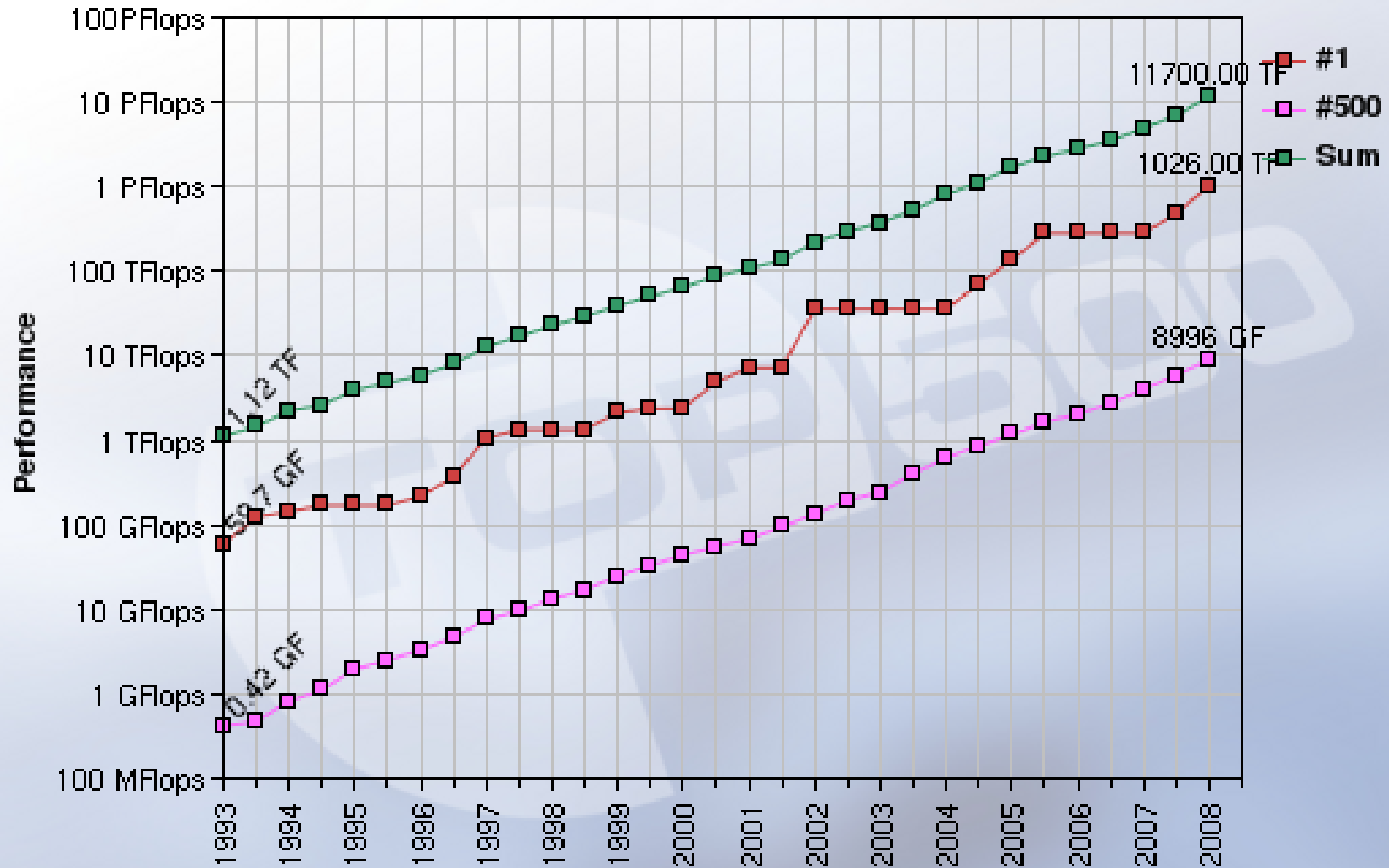
- The landscape of high performance computing and recent trends
- The Message-Passing Interface (MPI) and OpenMP programming styles.
- A hybrid parallel molecular dynamics simulation.
- Performance results and analysis.

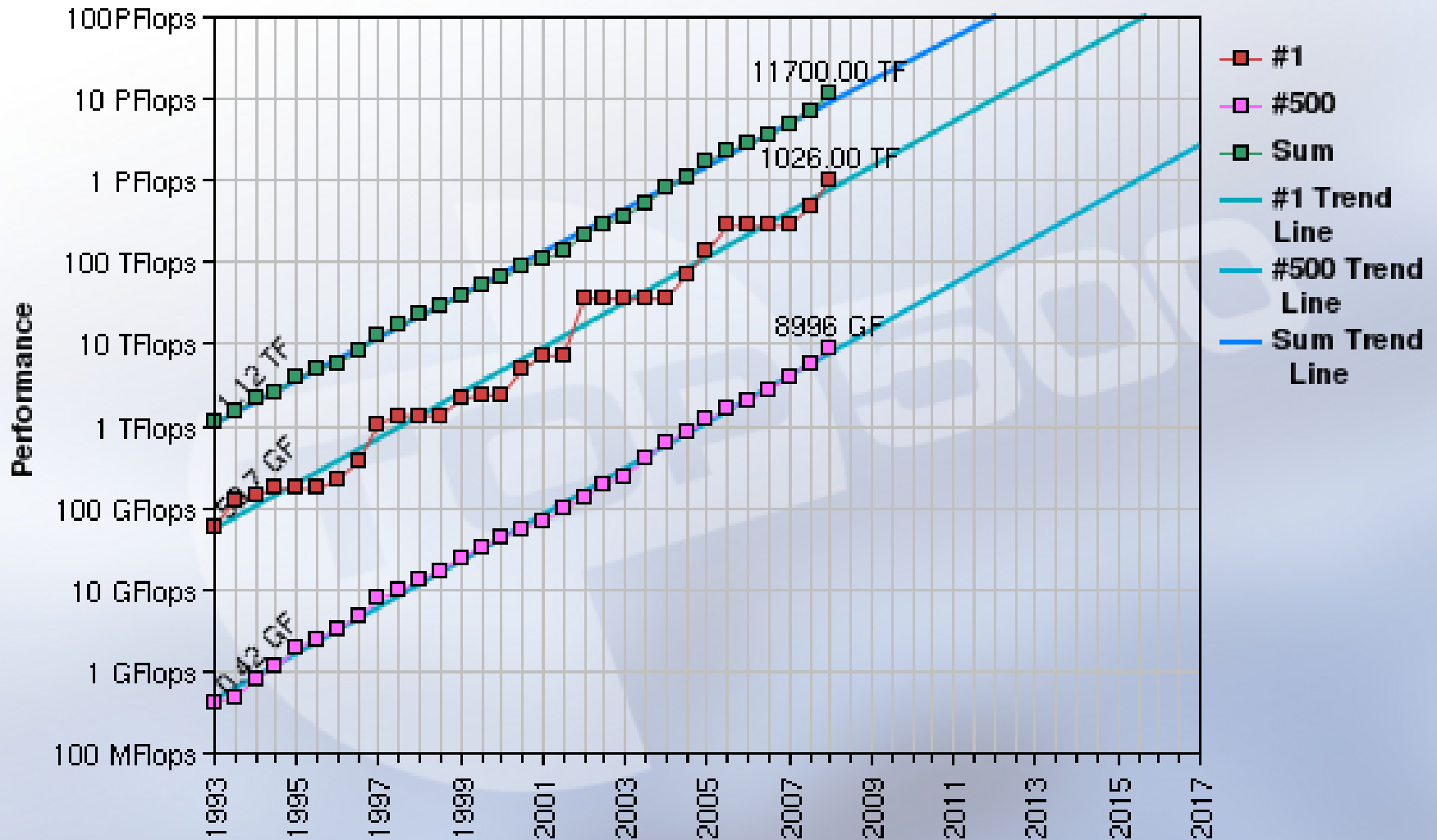
Computational Science

- Computational Science is the use of computer simulation to solve scientific problems.
- Often these problems require large amounts of computational power and memory.
- Thus, Computational Science is usually done on parallel computers.

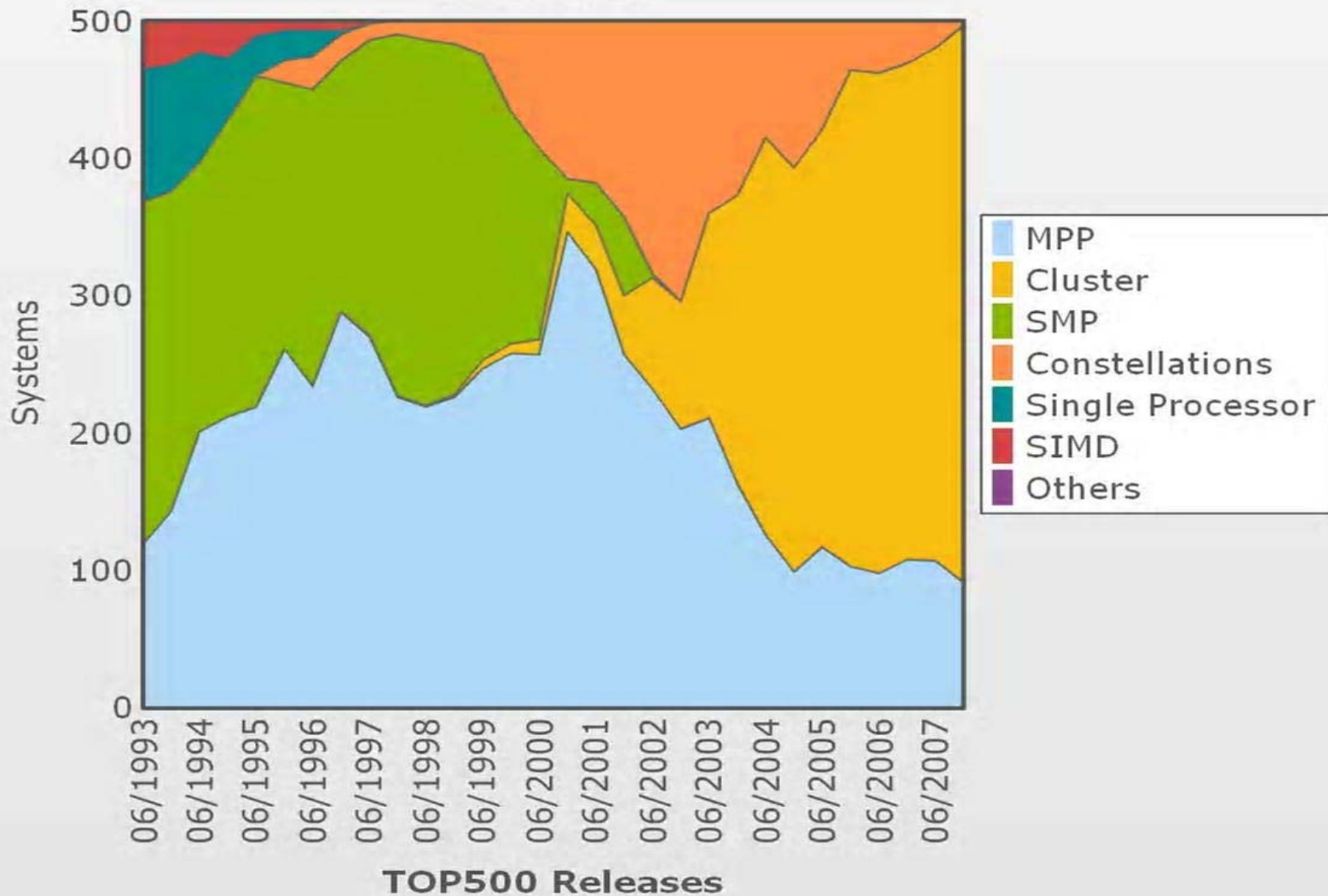
Parallelism and Memory

- Want more memory to solve bigger or more complex problems.
- Typical PCs have 2 Gbytes of RAM. Can fit an $16,384 \times 16,384$ array into 2 Gbytes of memory.
- The BlueGene/L parallel computer at Lawrence Livermore National Lab has 131,072 nodes with a total of 32.768 Tbytes of RAM. Can fit a $2,122,168 \times 2,122,168$ array into memory.

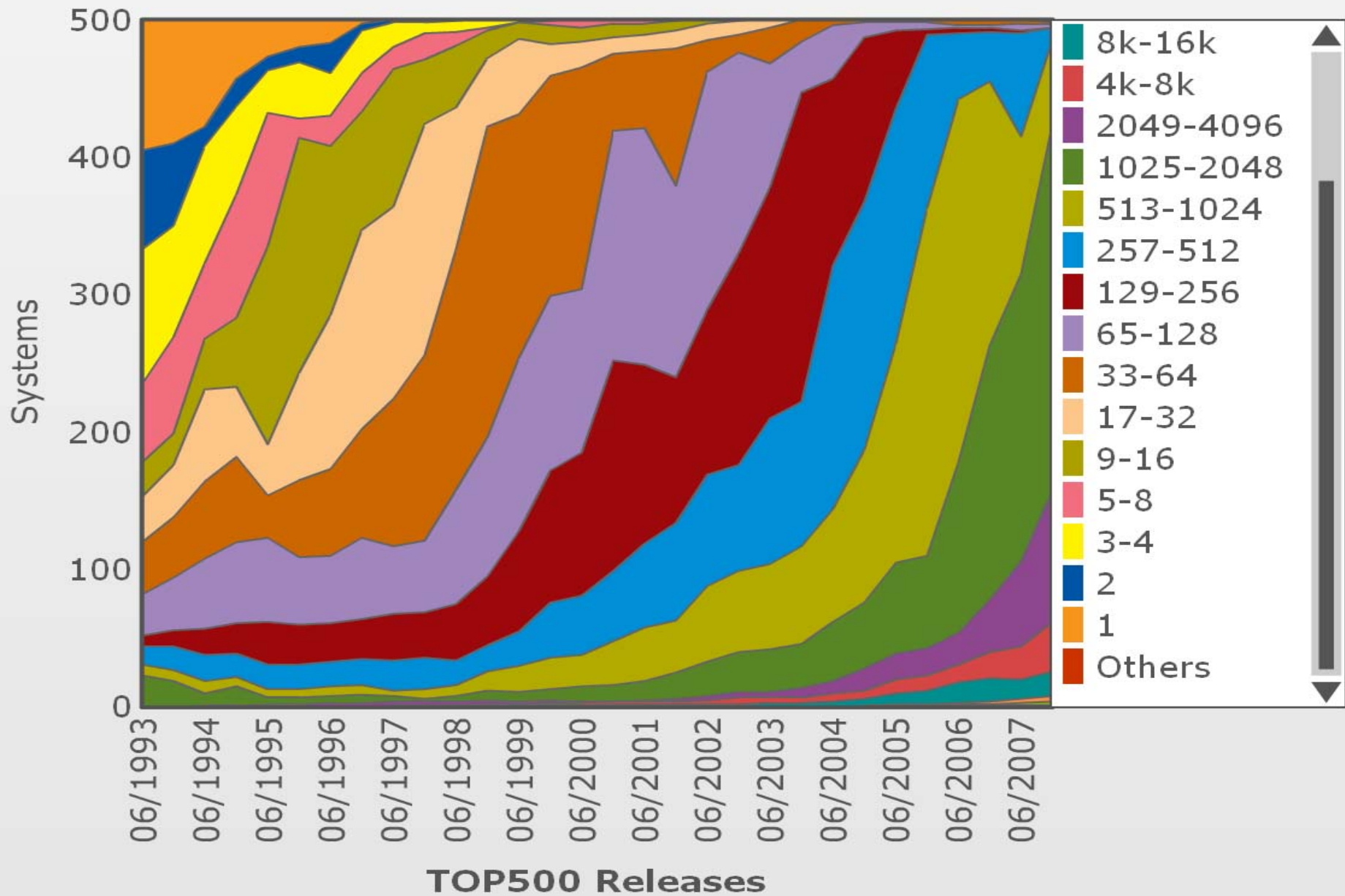




Architecture Share Over Time 1993-2007



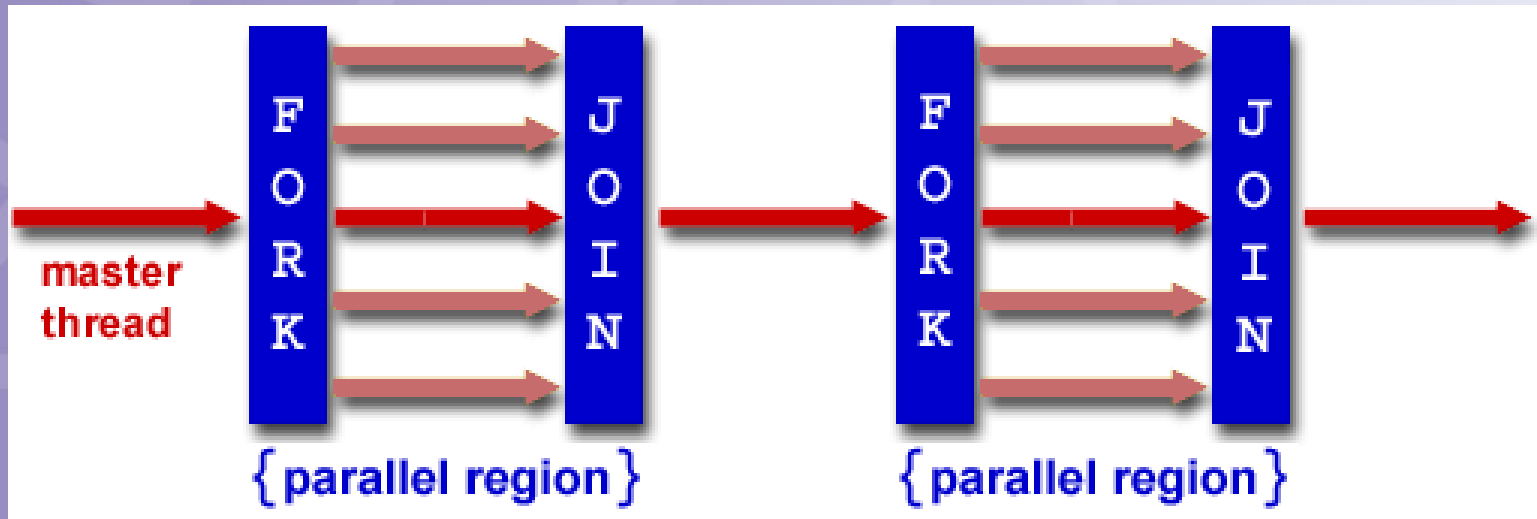
1993-2007



Quick Overview of OpenMP

- OpenMP can be used to represent task and data parallelism.
- In case of data parallelism, OpenMP is used to split loop iterations over multiple threads.
- Threads can execute different code but share the same address space.
- OpenMP is most often used on machines with support for a global address space.

OpenMP Fork/Join Model



OpenMP and Loops

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main () {
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++) c[i] = a[i] + b[i];
    }
    /* end of parallel section */ }
```

Hybrid Programming

- Uses both Message Passing (MP) and Shared Memory (SM) programming models.
 - Use Message Passing to communicate between nodes of a cluster.
 - Use Shared Memory programming to communicate within a node.
- Shared memory accesses within a node *should* be faster than using message passing.
- Exploit the hierarchical nature of modern HPC systems by using hierarchical parallelism.
- Previous work done focusing on SMP systems, found mixed results.
 - Performance depends on compiler, hardware and algorithms used.

Current Research

- Looking at a 3D Molecular Dynamics (MD) code.
 - Originally parallelised using MPI
- Have implemented a hybrid version.
- Added Shared Memory parallelism using OpenMP.
 - Main work loop parallelised with shared memory threading.
- Performance testing on different multi-core systems.
 - Dual core Intel Xeon system (Woodcrest).
 - Quad core Intel Xeon system (Harpertown).

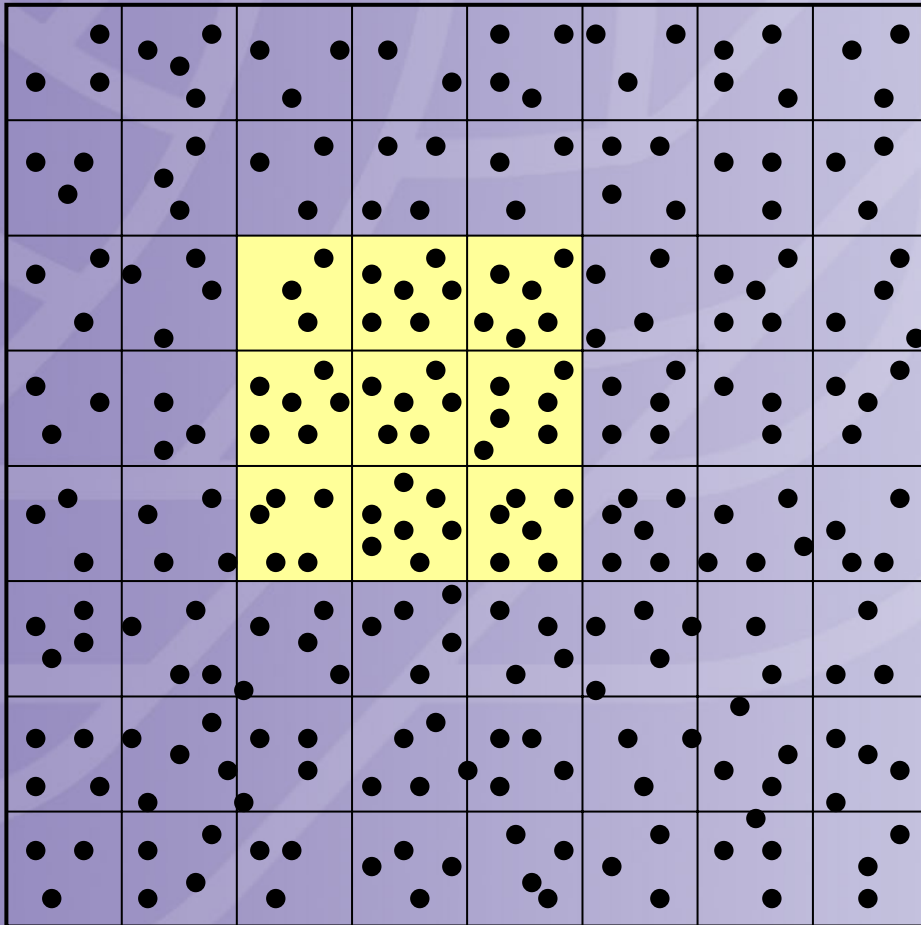
The Molecular Dynamics Application

- Shifted Lennard-Jones potential used to study molecular liquids.

$$v(r) = 4\epsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6\right) - A - B(r - r_c)$$

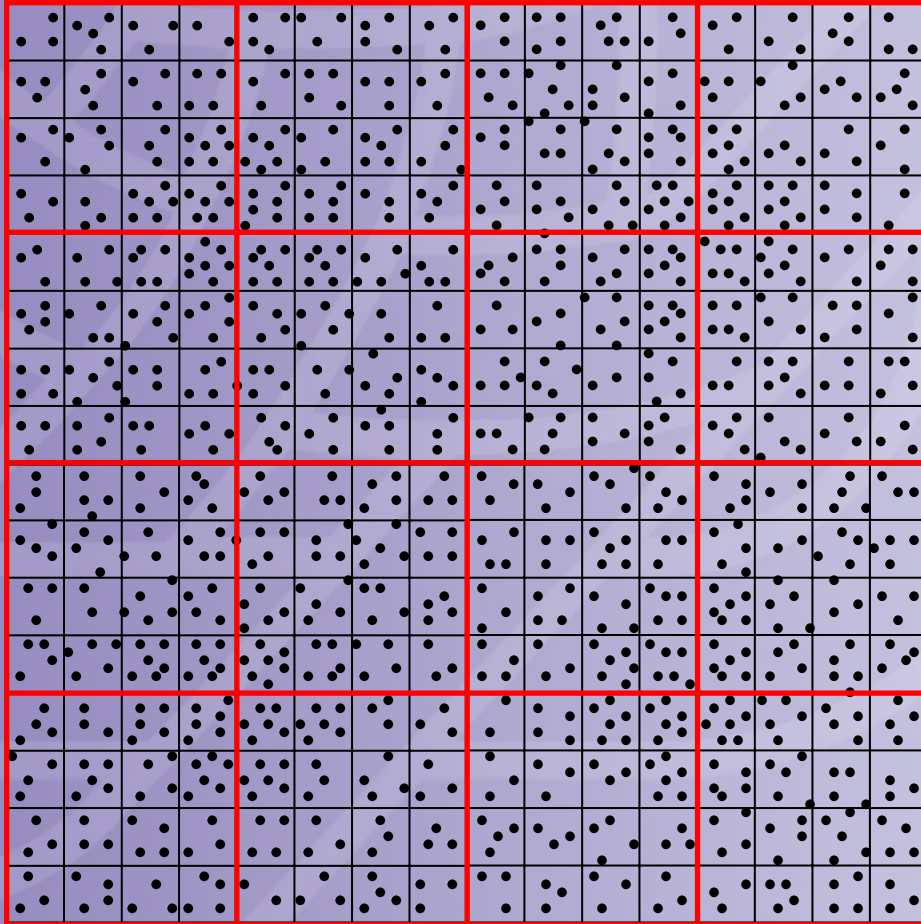
- Assume the potential between 2 particles is zero if they are more than r_c apart. Typically $r_c = 2.5\sigma$.
- A and B are constants that ensure the potential and the force go to zero at $r = r_c$.
- The velocity Verlet algorithm is used to update particle positions and velocities.

Cut-Off Distance



If we divide the domain of the problem into cells of size $r_c \times r_c$ each particle only interacts with the particles in its own cell and the 8 neighbouring cells.

Data Distribution



- The particles are distributed to processes by assigning a rectangular block of cells to each process.

Communication Requirements

- Each particle needs information about the particles in the neighboring cells in order to determine the force on it. So we need to communicate particles lying in cells along the boundary of each process
- When particles move they may travel from the set of cells owned by one process to those of another process. This is called *particle migration* and requires communication.

Determining the Forces

```
for (each particle, p, in this process){
    find out the location (i,j) of cell that p is in
    force[p] = 0;
    for (cell (i,j) and the 8 neighbouring cells){
        for (each particle q in cell){
            add force of q on p to force[p]
        }
    }
}
```

Structure of MD Code

for each time step

update positions and velocities

communicate using MPI (point-to-point)

update forces ←

update velocities

sum energies (reduction)

accumulate statistics

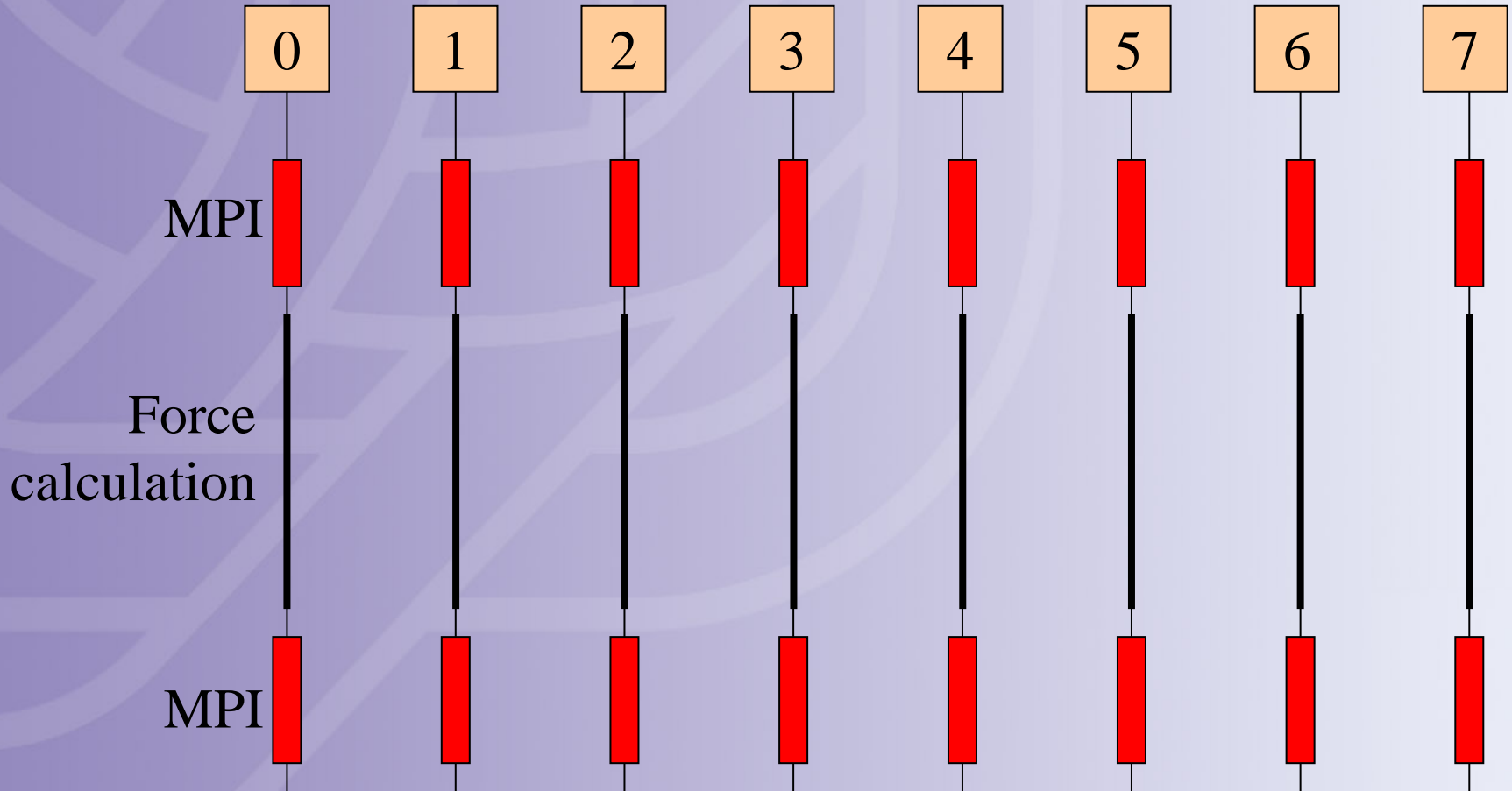
equilibrate if necessary

Most of the
computational
work is here

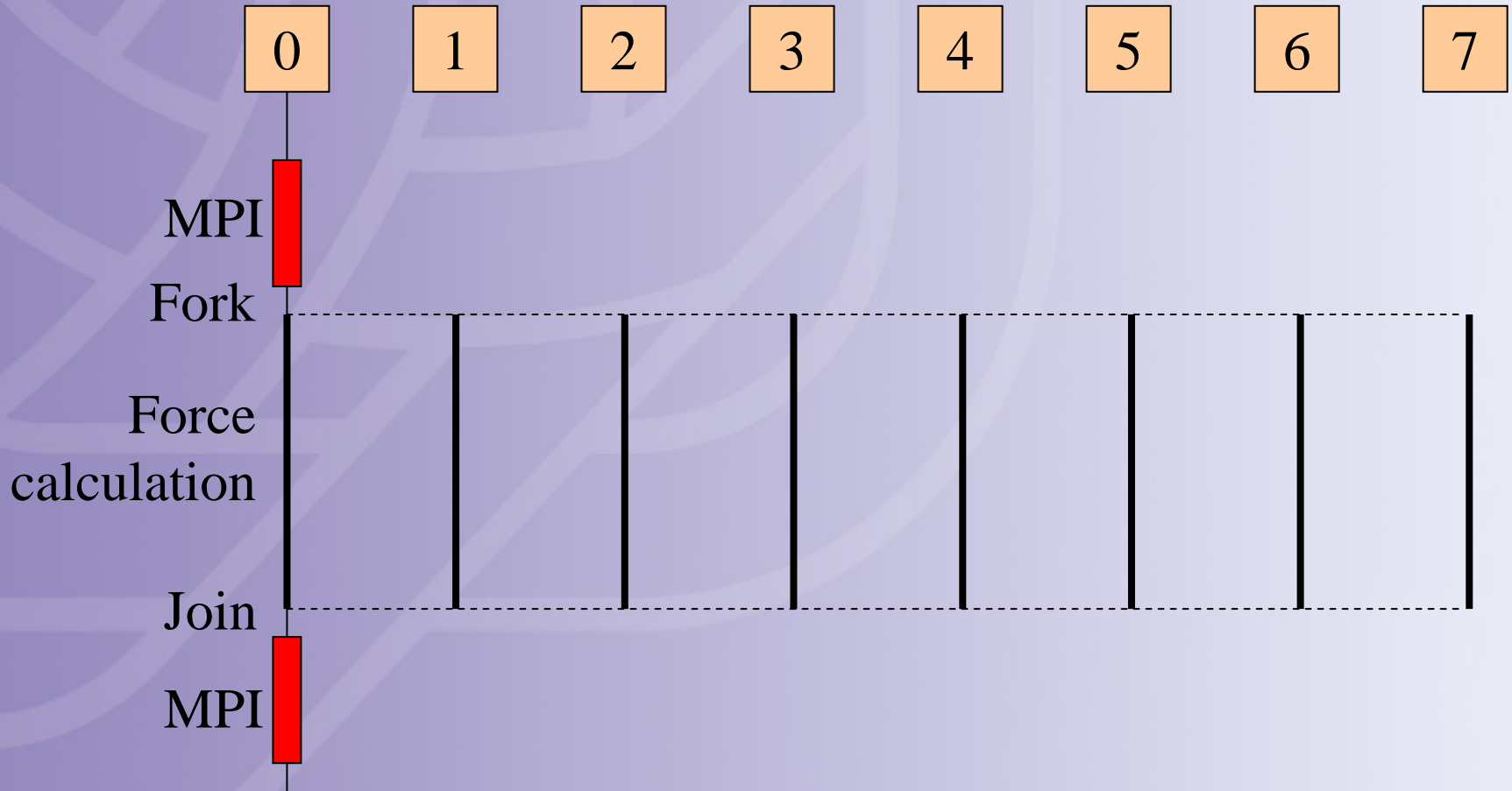
Pure MPI and Hybrid Codes

- Pure MPI: one MPI process runs on each core. Communication may be via
 - Sockets
 - Shared memory
- Hybrid1: each node has the SPMD code running on one core, and uses MPI to communicate particle data between nodes.
- Hybrid2: same as Hybrid1 except that each node has MPI running on two cores.
- In the force routine OpenMP is used to spread the force calculations over all cores in a node.

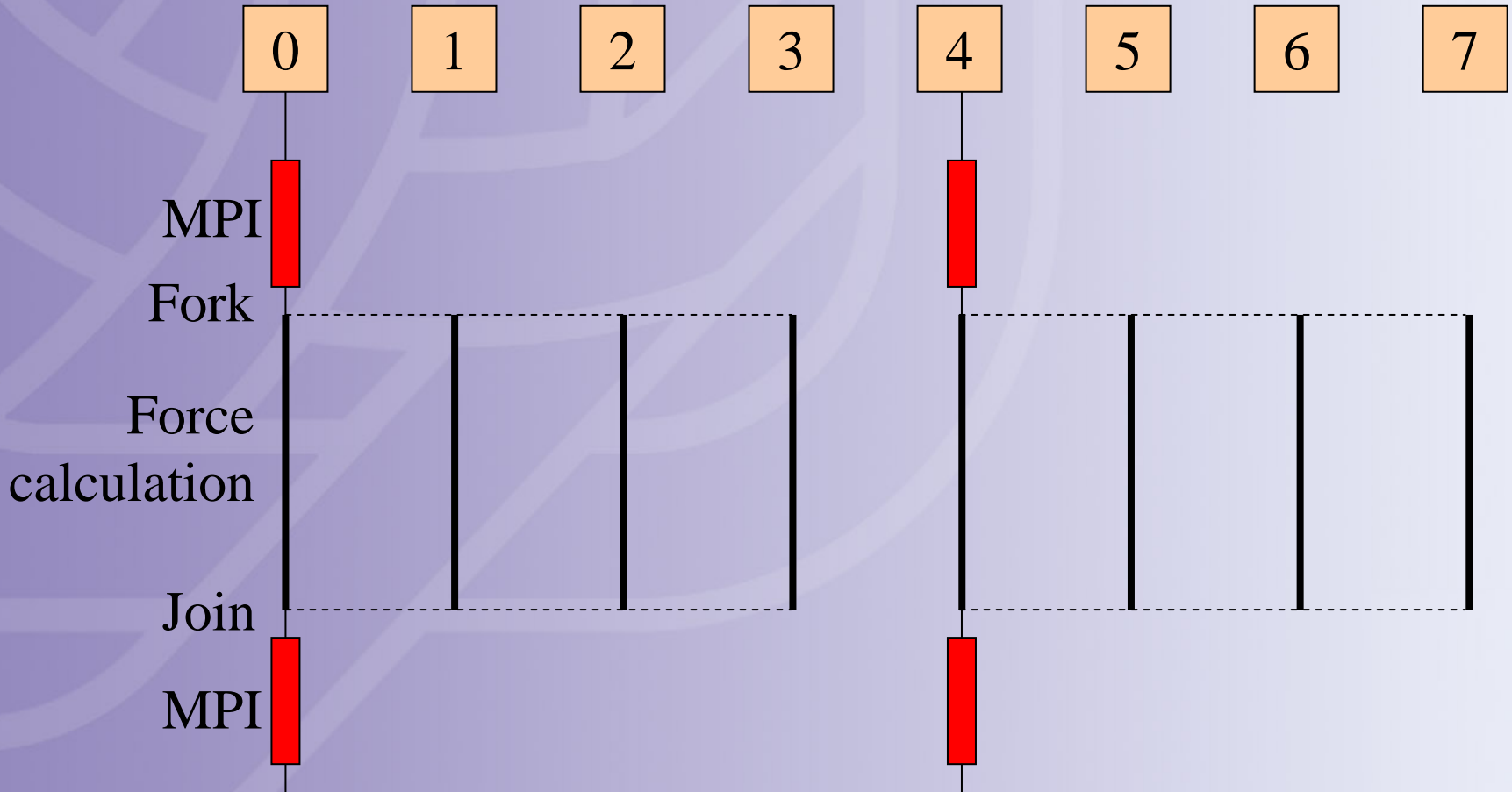
MPI Code



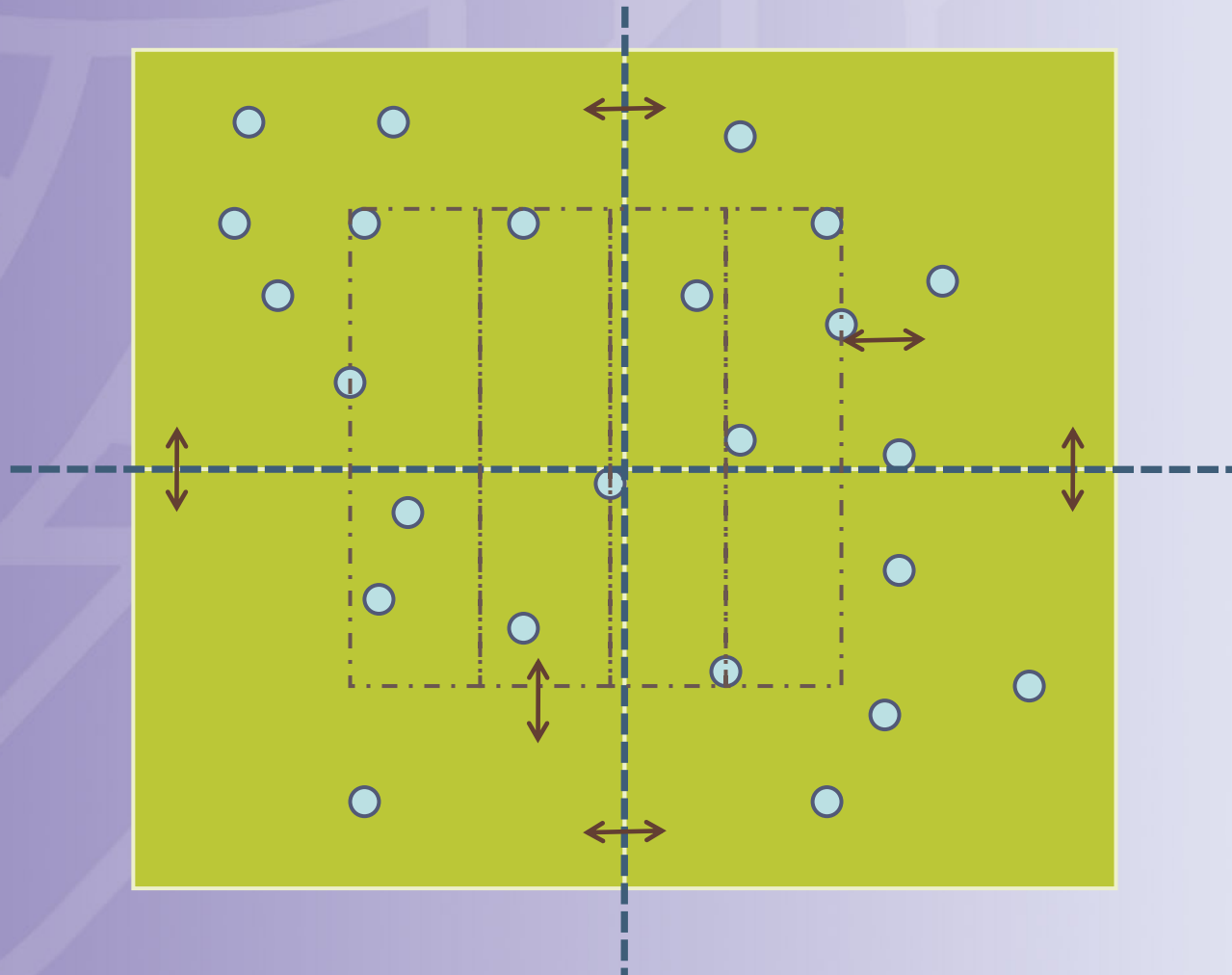
Hybrid 1 Code



Hybrid 2 Code



Hybrid Implementation



2D Space
Divide
between
nodes
Distribute
to nodes
Message
Passing
between
nodes
Shared
Memory
threading
within
node

Structure of Threaded Force Routine

```
#pragma omp parallel default(none) shared(v,p,fx,fy,fz,rx,ry,rz,...)
{
  #pragma omp for private(i)
  for (each particle, i){
    set forces on i to 0
  }
  #pragma omp for private(icell,...) reduction(+:v,p)
  for (each cell, icell, on this node){
    for (each particle, i, in this cell){
      for (each particle, j, in this and neighboring cells){
        Find force on particle i due to particle j, and add to force on
        particle i. Sum contributions to v and p.
      }
    }
  }
}
```

Advanced Research Computing @ Cardiff

- Merlin, the main ARCCA machine, consists of 256 nodes each with 2 processors. Each processor has 4 cores = 2048 cores.
- Linpack performance is 20.12 Tflop/s.
- One of the top 3 fastest machines in academia in the UK.

Merlin at ARCCA

- 256 Compute Nodes, each containing:
 - 2 x Xeon E5472 Quad-Core 3.0GHz Processors (8 cores per node)
 - 16GB Ram
 - 160GB HDD
- Infiniband 4X DDR network (20 Gbps)
 - 140 to 420 nanosecond latency
 - 288 port Voltaire switch.
- Gigabit Ethernet networks
 - 1 & 10 Gbps

Merlin - Xeon E5472 Processor

- Quad-core, Harpertown/Seaburg.
- 32kb Instruction Cache and 32kb Data Cache at L1 (per core).
- 6MB L2 cache per pair of cores.
- 1600 Mhz Front Side Bus.

Merlin - Software

- Red Hat Enterprise Linux 5.
- PBS Pro Job Scheduler.
- Intel 10.1.015 Compilers.
- Bull MPI 2.

CSEEM64T at Daresbury Lab

- 32 Compute Nodes, each containing:
 - 2 x Xeon Dual-Core 5160 3.0Ghz Processors (4 cores per node).
 - 8GB Ram.
- Gigabit Ethernet Network.
- Infinipath Network.

CSEEM64T – Xeon 5160 Processor

- Dual-Core, Woodcrest
- 32kb Instruction Cache and 32kb Data Cache at L1 (per core).
- 4MB L2 cache shared between both cores.
- 1333Mhz Front Side Bus

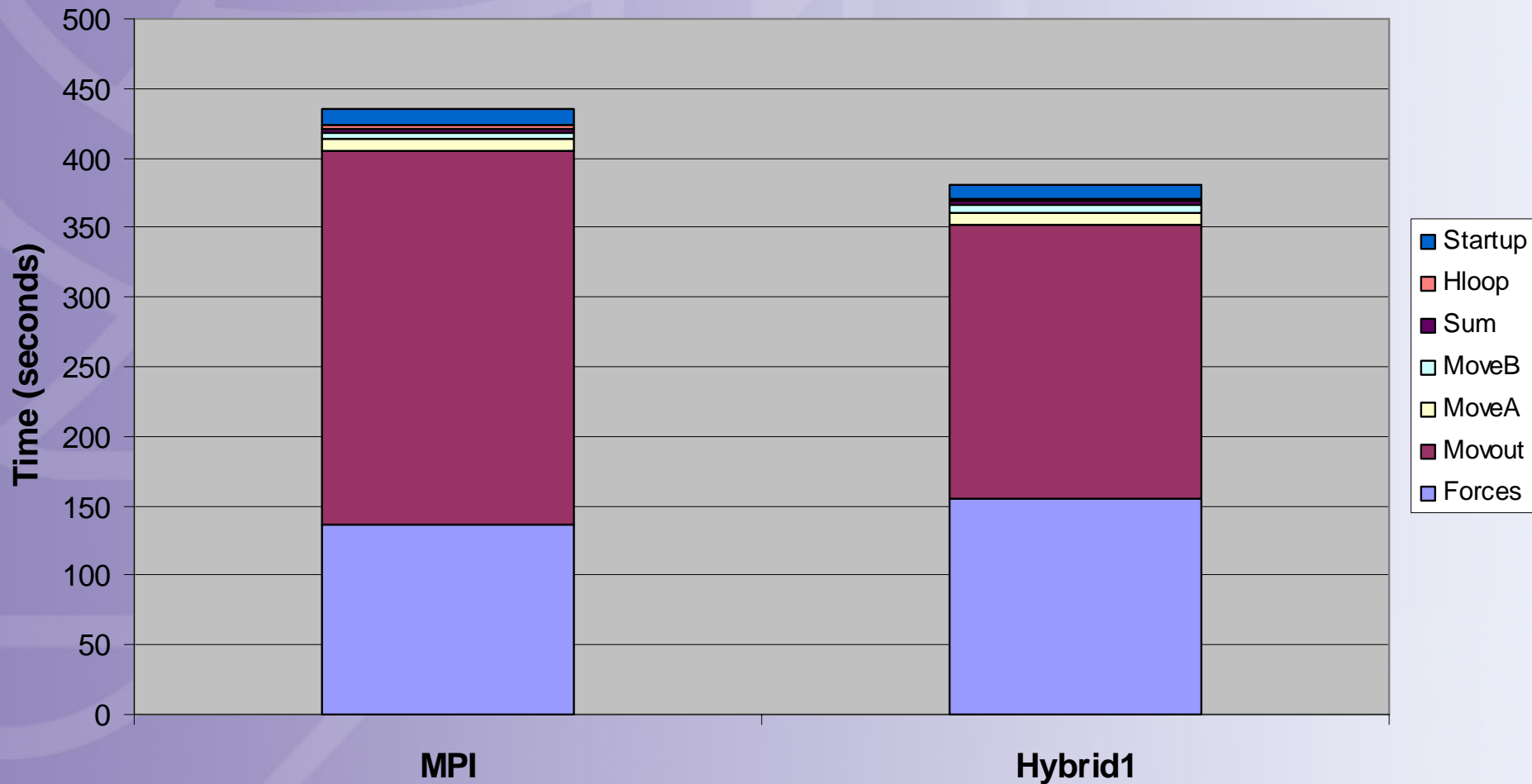
CSEEM64T - Software

- Suse Linux 10.1
- Intel 10.1.015 Compilers
- Intel MPI 3.0
- Sun Grid Engine Job Scheduler

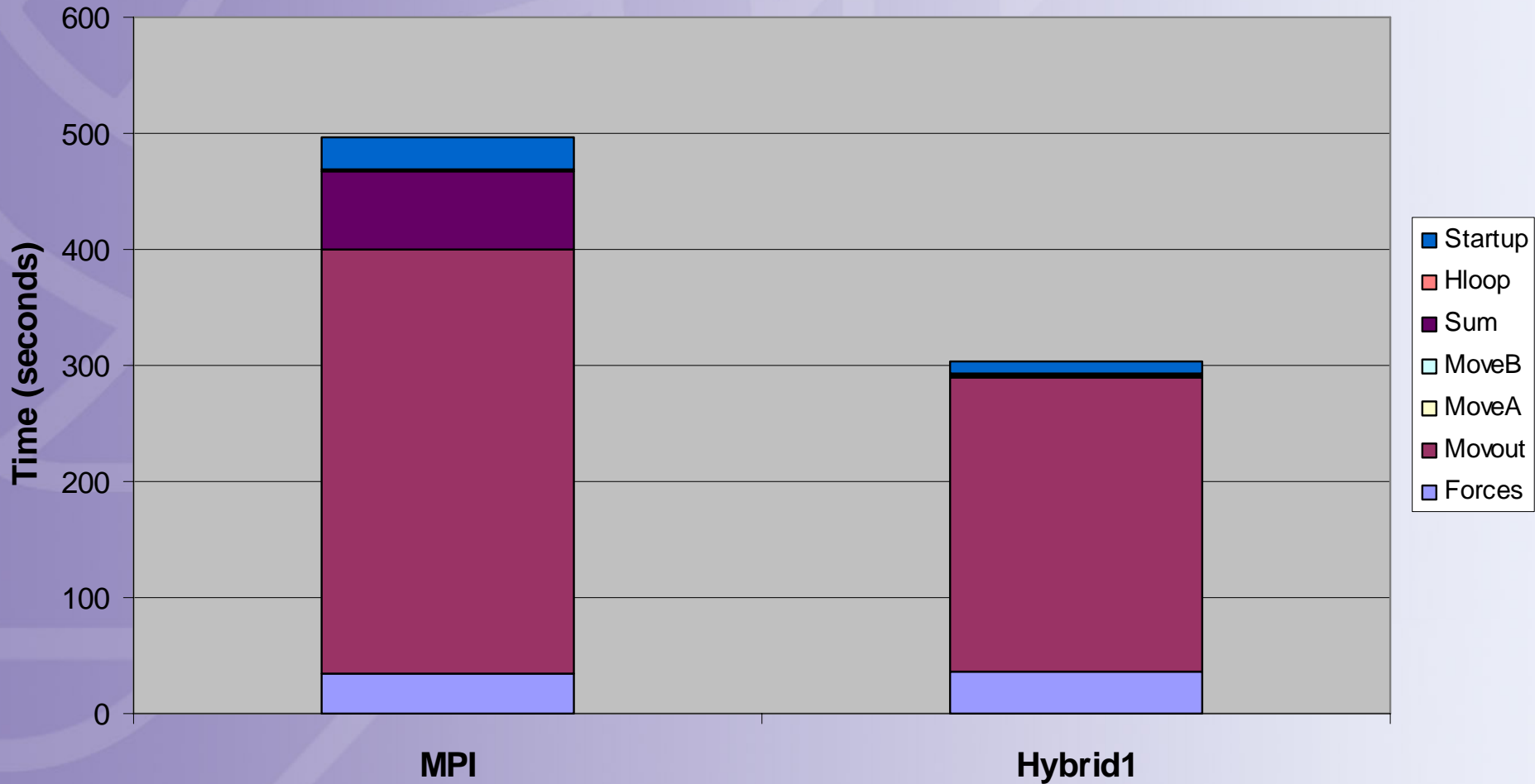
Further info:

- Merlin:
 - <http://www.cardiff.ac.uk/arcca/services/equipment/intro-merlin.html>
- CSEEM64T:
 - <http://www.cse.scitech.ac.uk/disco/cseem64t/cseem64t.shtml>

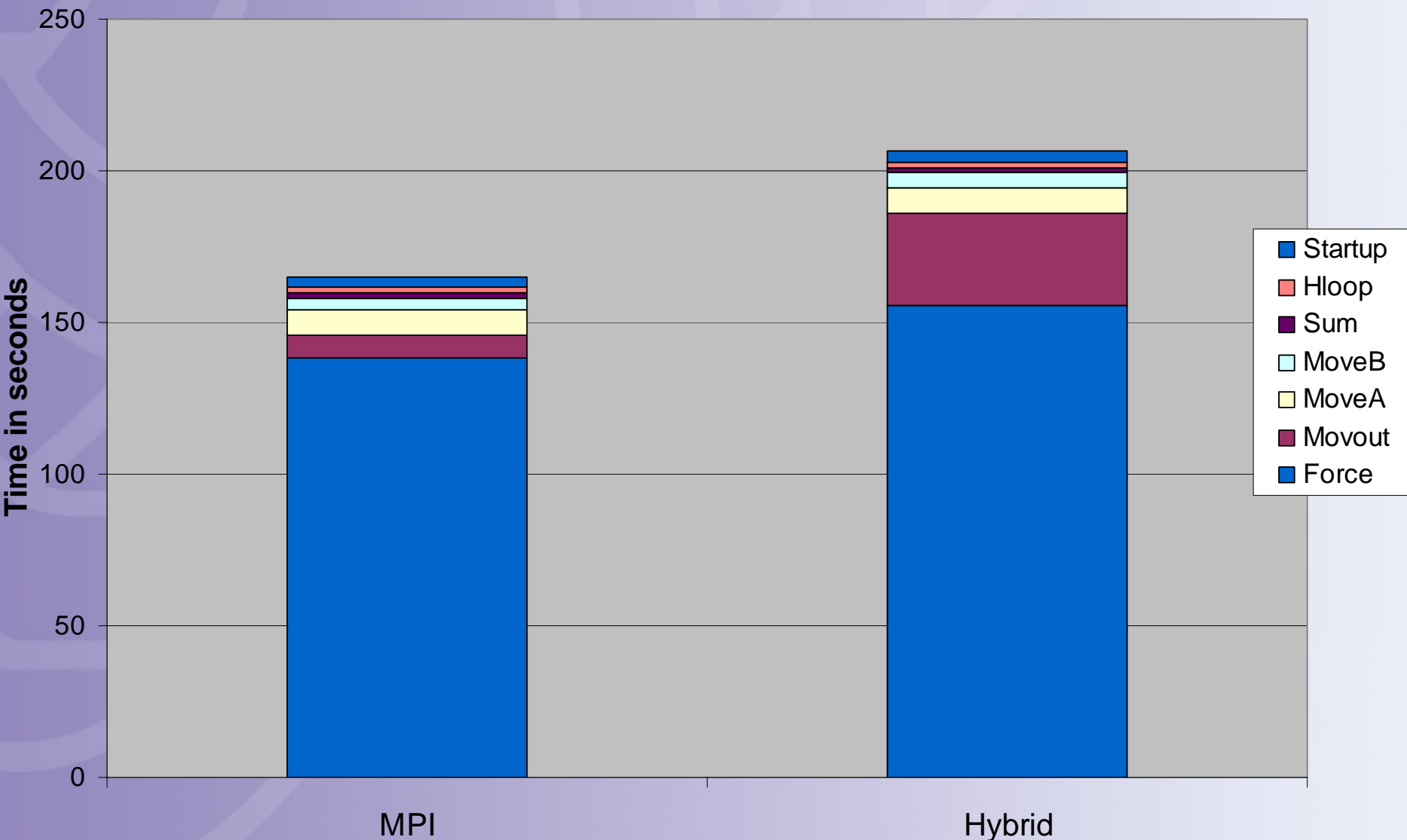
Merlin (GigE) - Time breakdown - 44,957,696 particles - 128 cores



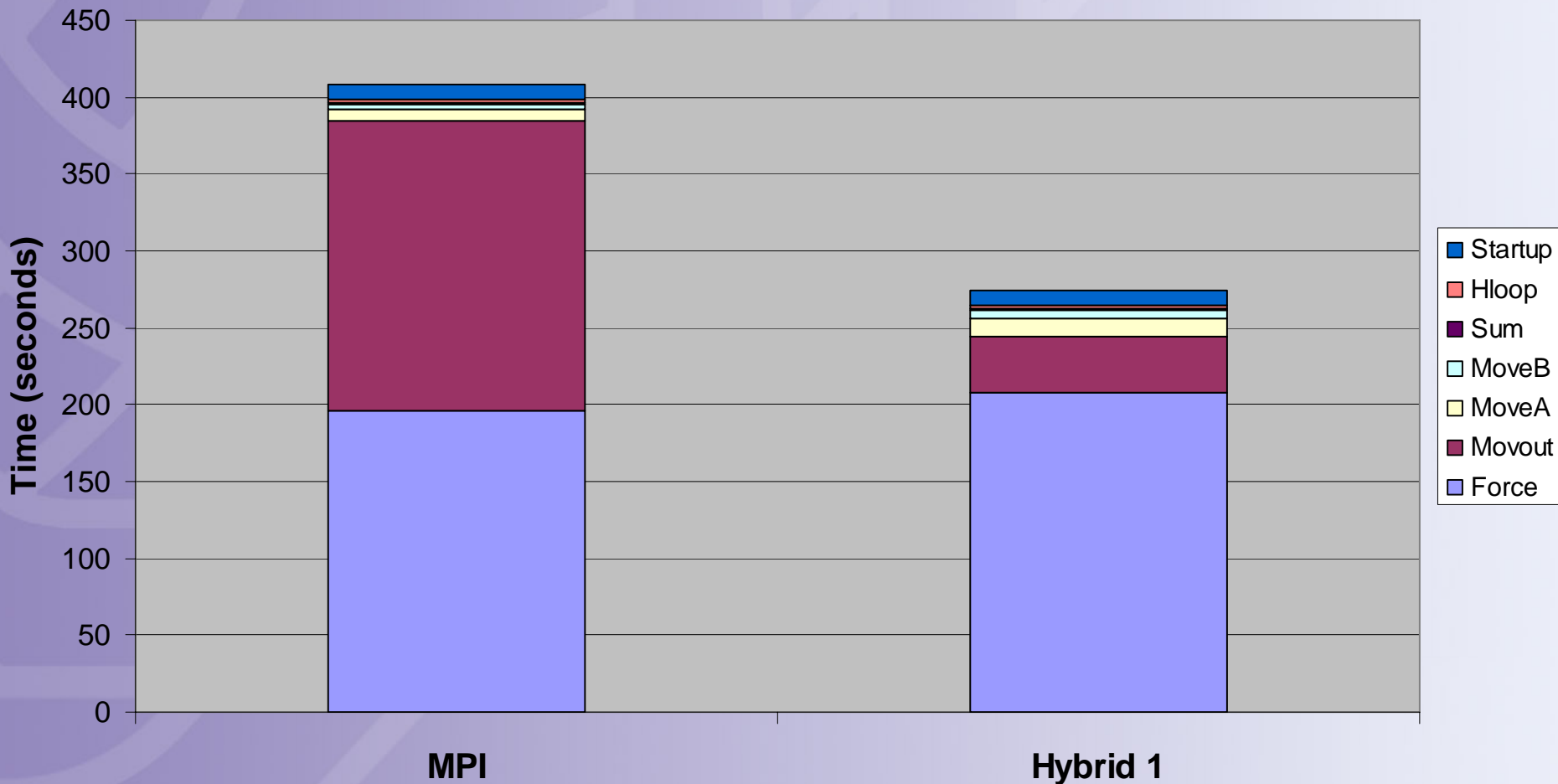
Merlin (GigE) - Time breakdown - 44,957,696 particles - 512 cores



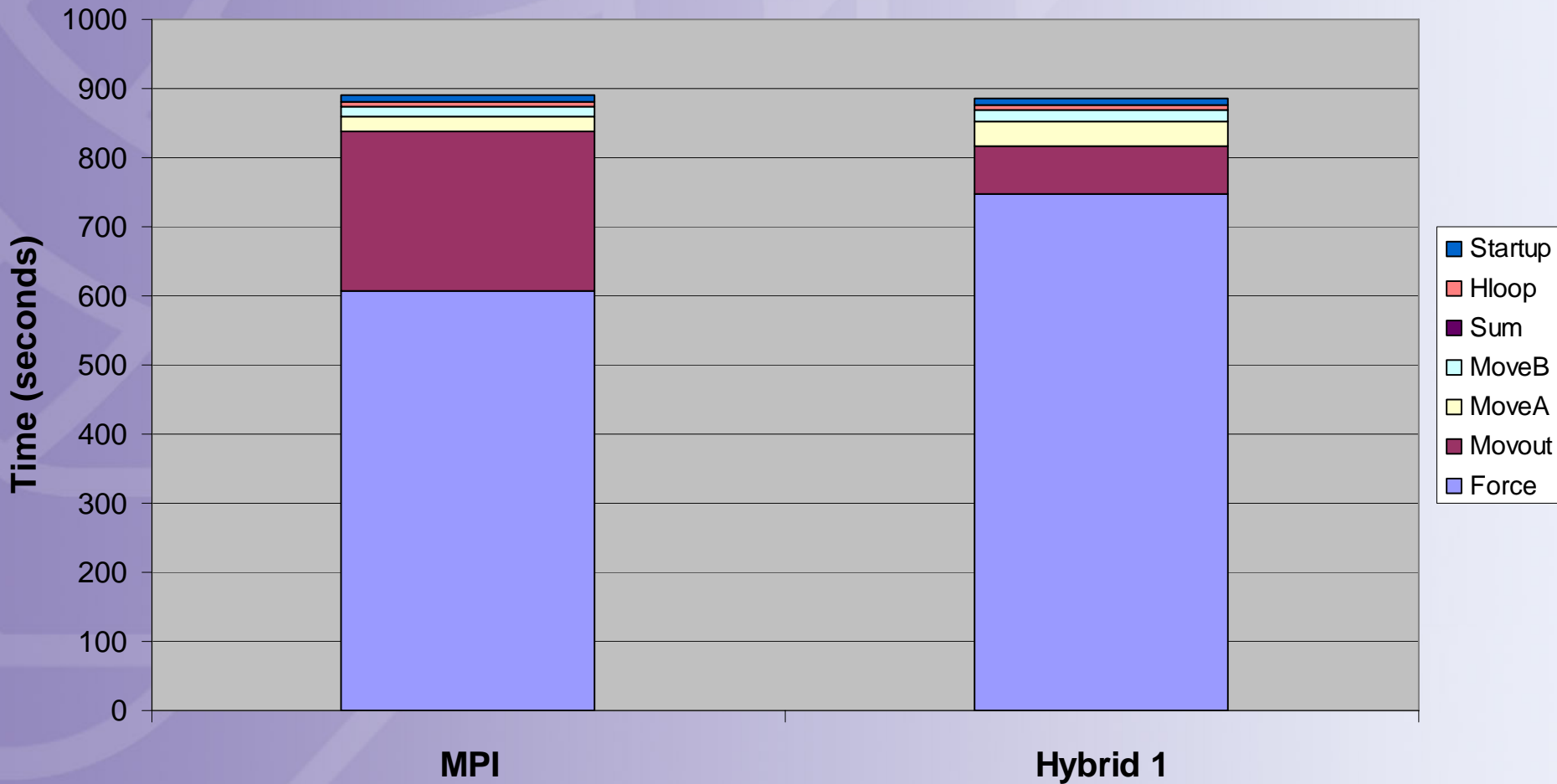
Merlin (Infiniband) - Time Breakdown - 44,957,696 Particles - 128 Cores



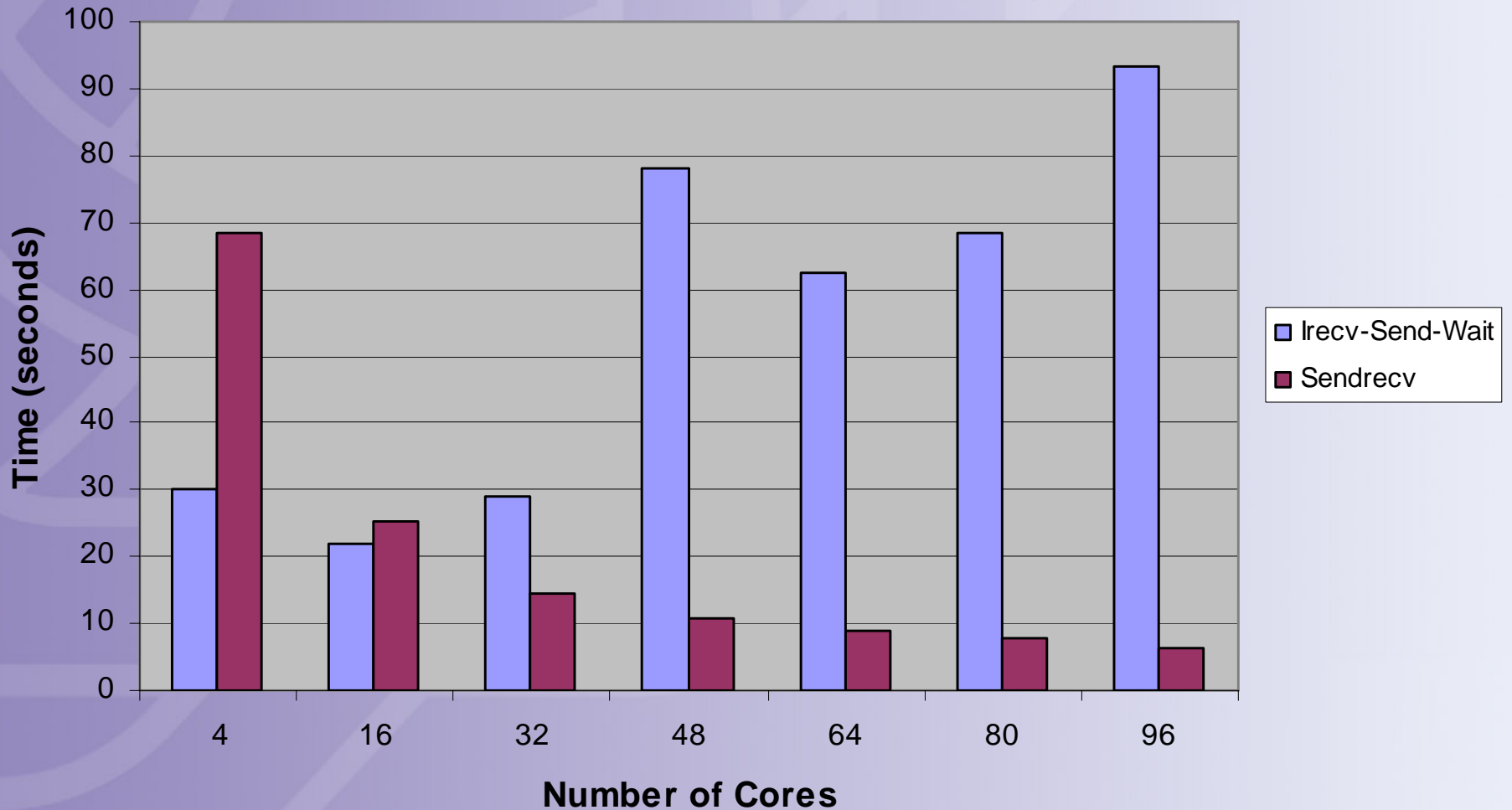
CSEEM64T - Time breakdown - 44,957,696 particles - 96 cores



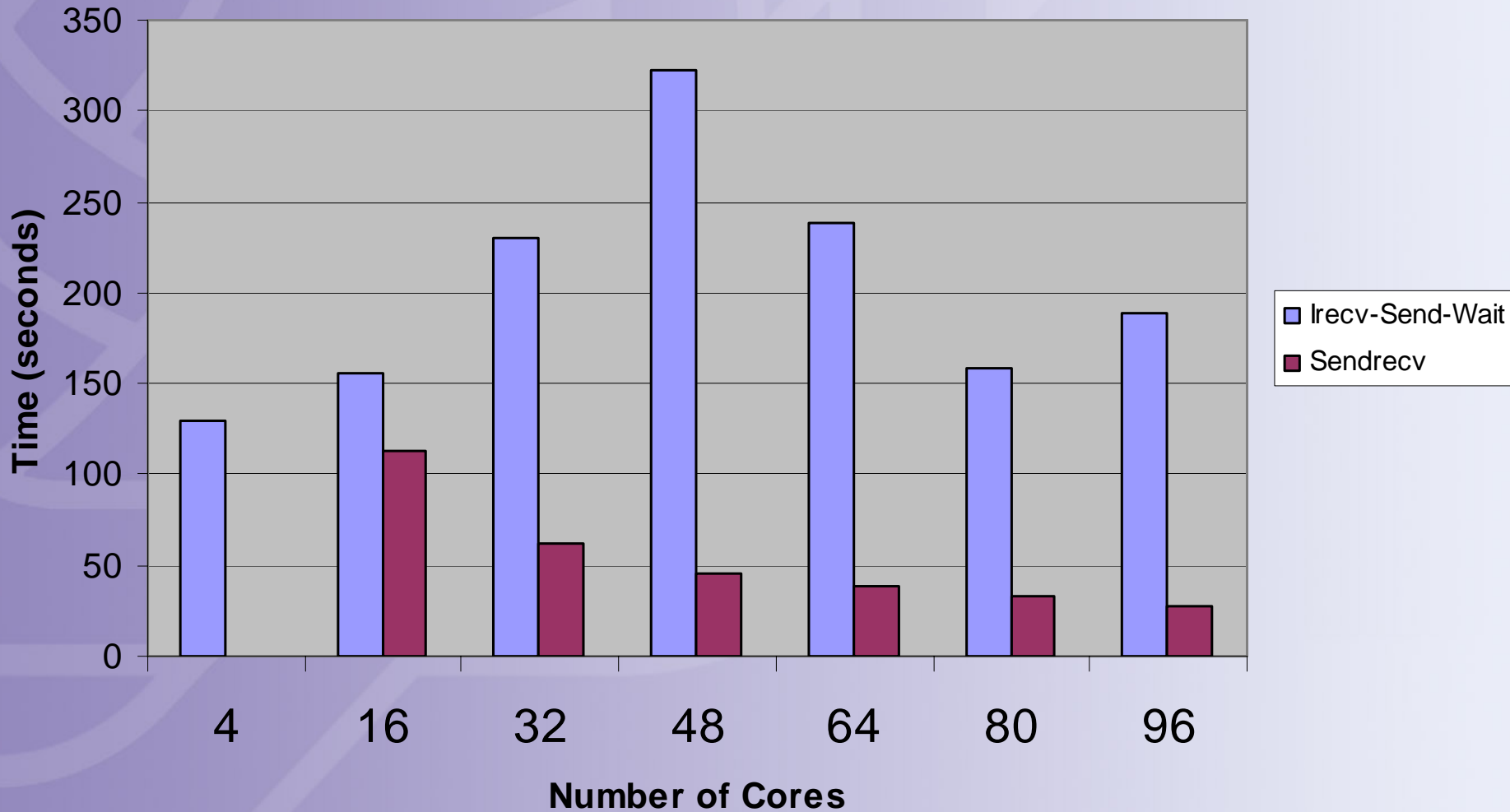
CSEEM64T - Time breakdown - 44,957,696 particles - 32 cores



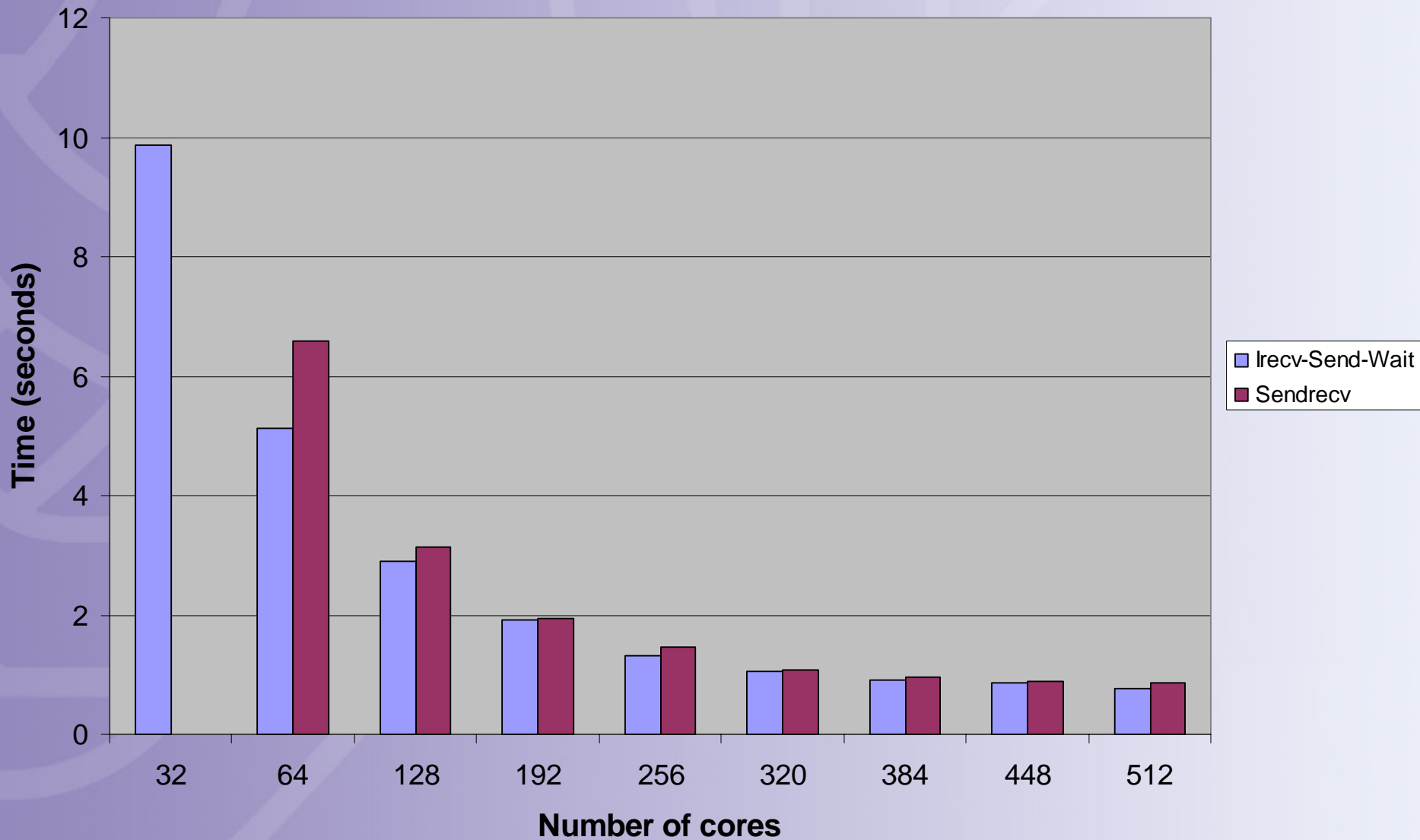
CSEEM64T (GigE) movout Times - 8,388,608 Particles



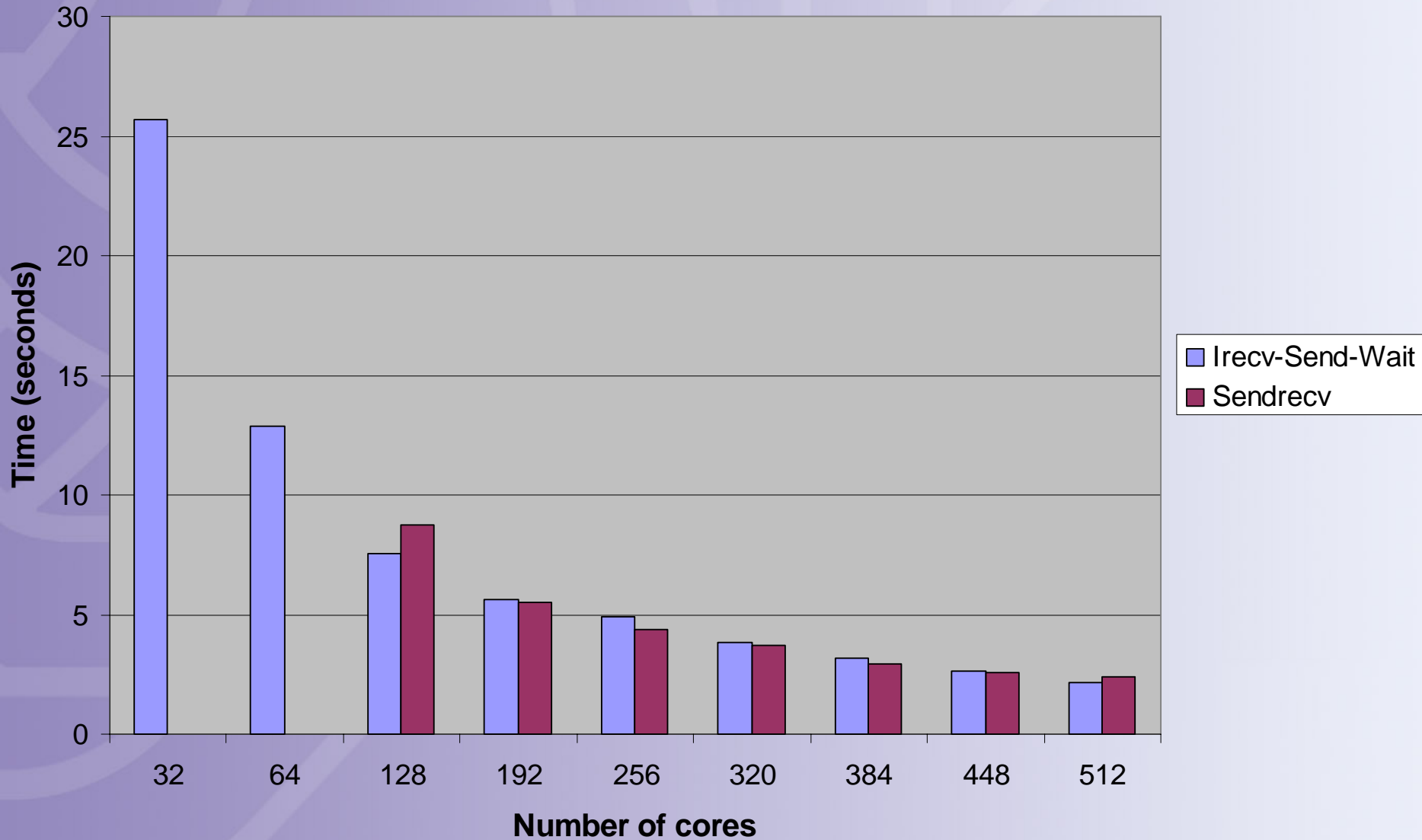
CSEEM64T (GigE) movout Times - 44,957,696 Particles



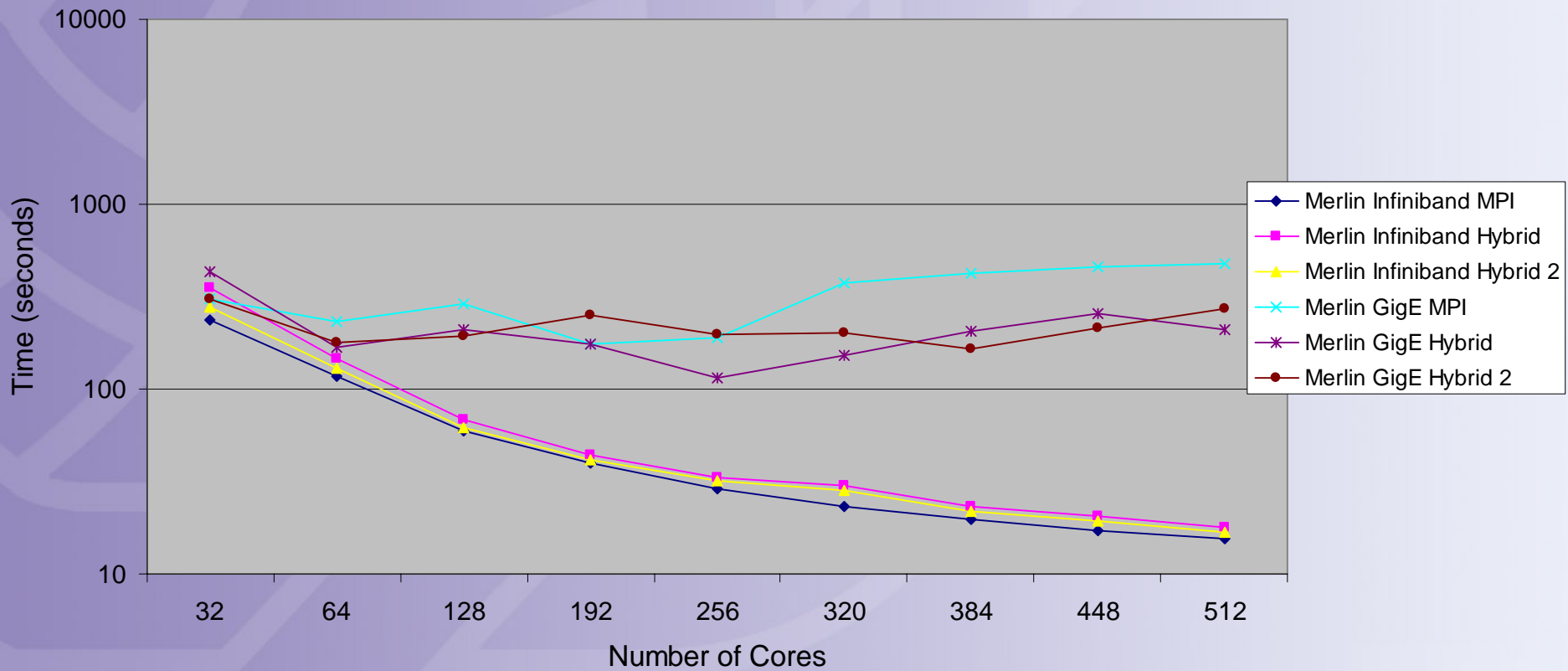
Merlin (InfiniBand) - movout timings - 16,384,000 particles



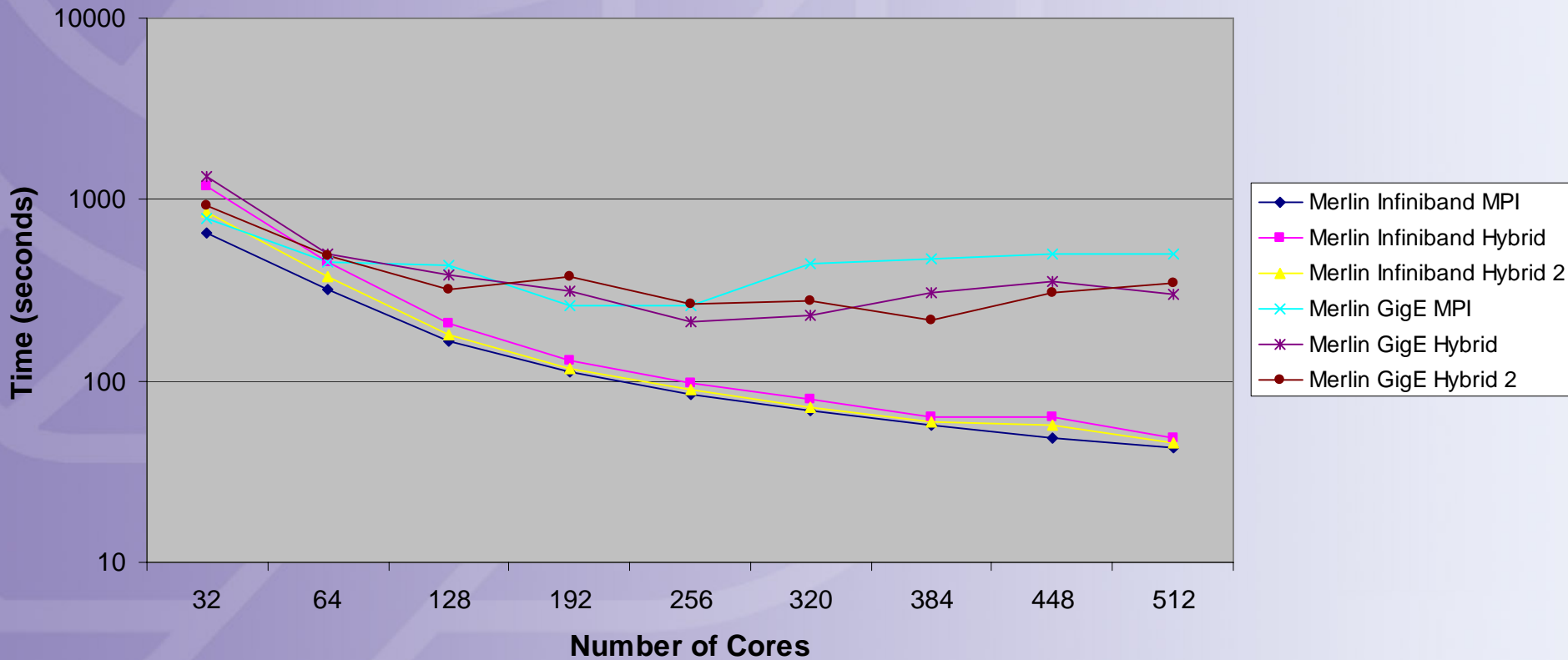
Merlin (InfiniBand) - movout timings - 44,957,696 particles



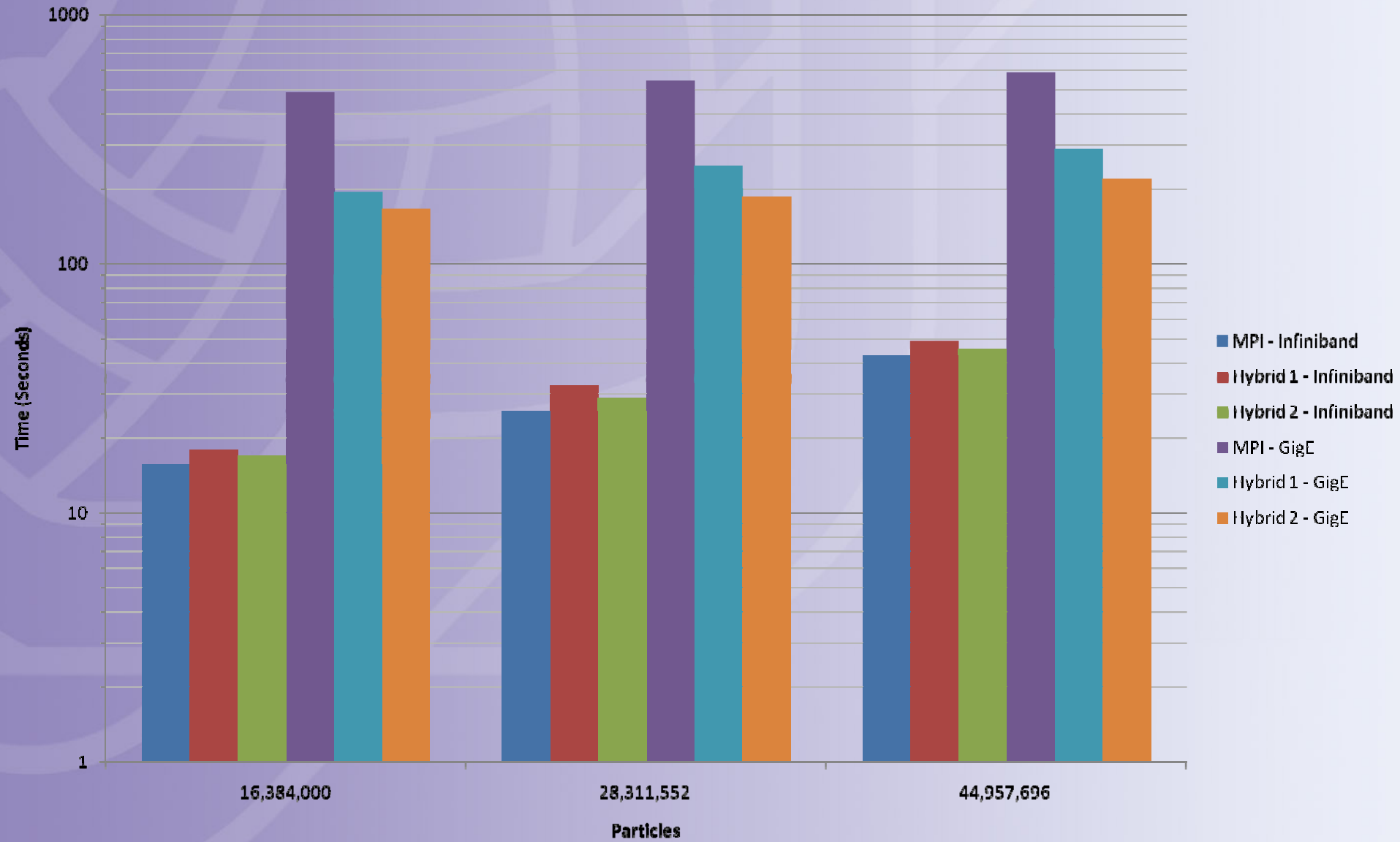
Timings for 16,384,000 Particles



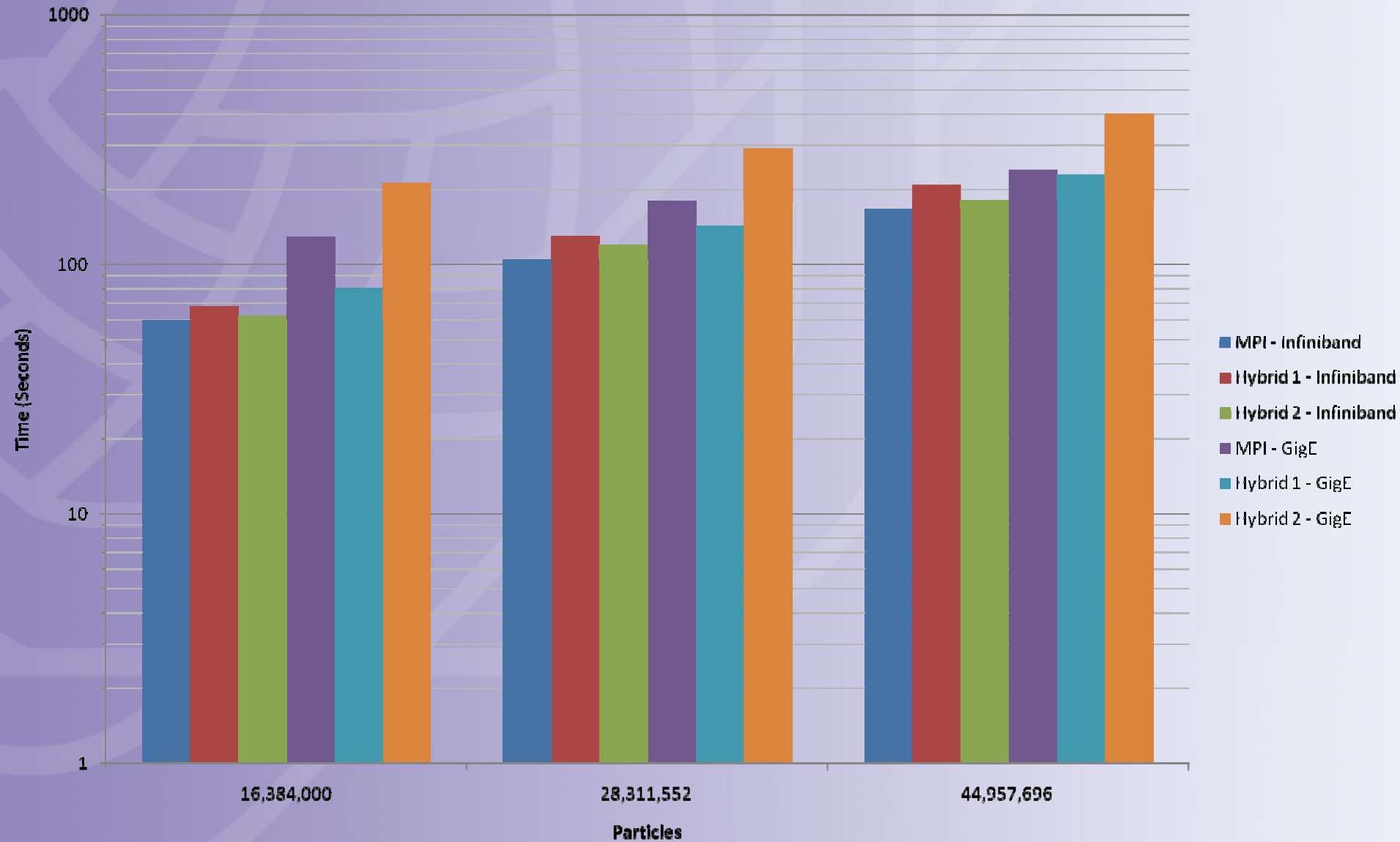
Timings for 44,957,696 Particles



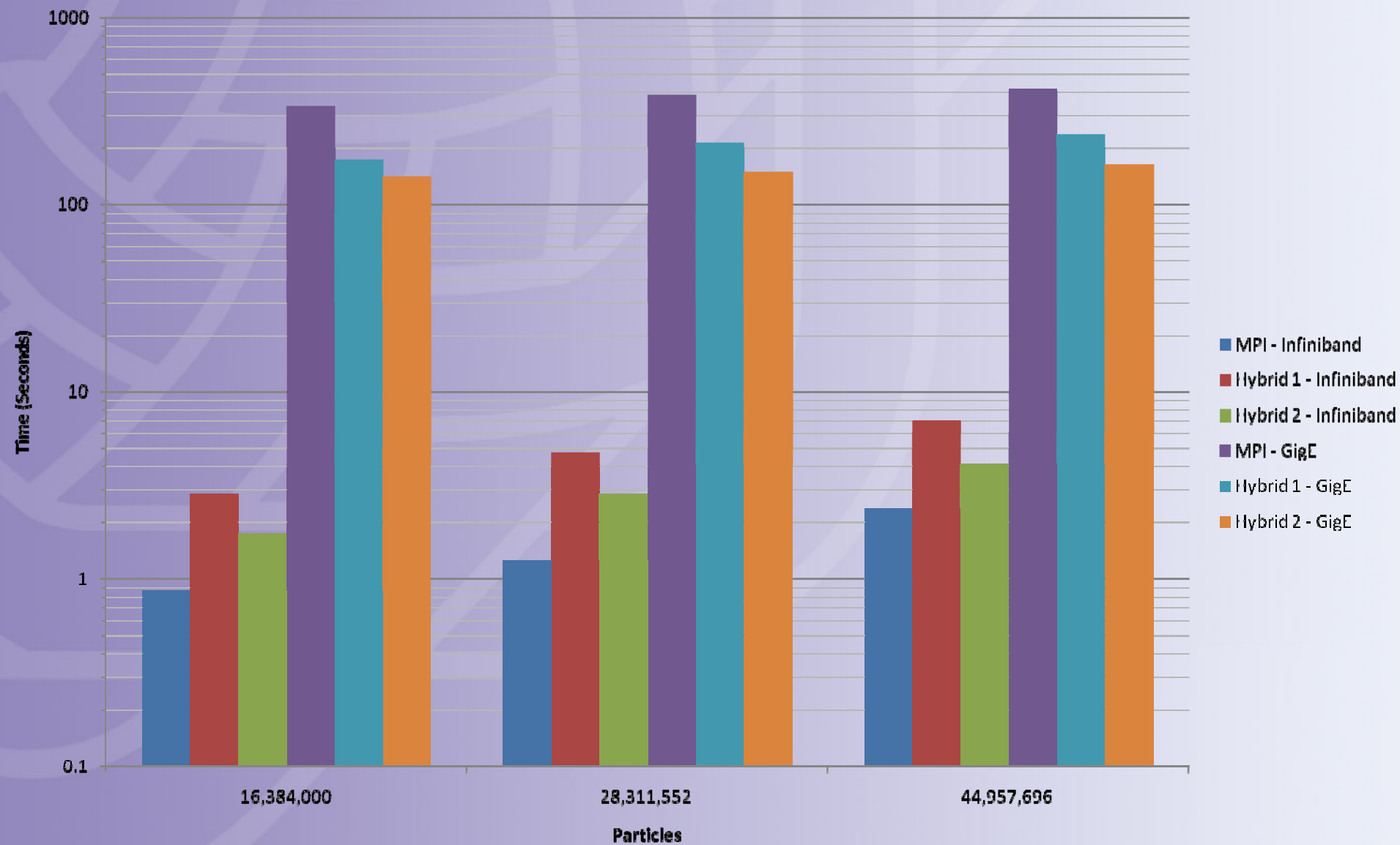
Merlin - 512 Cores



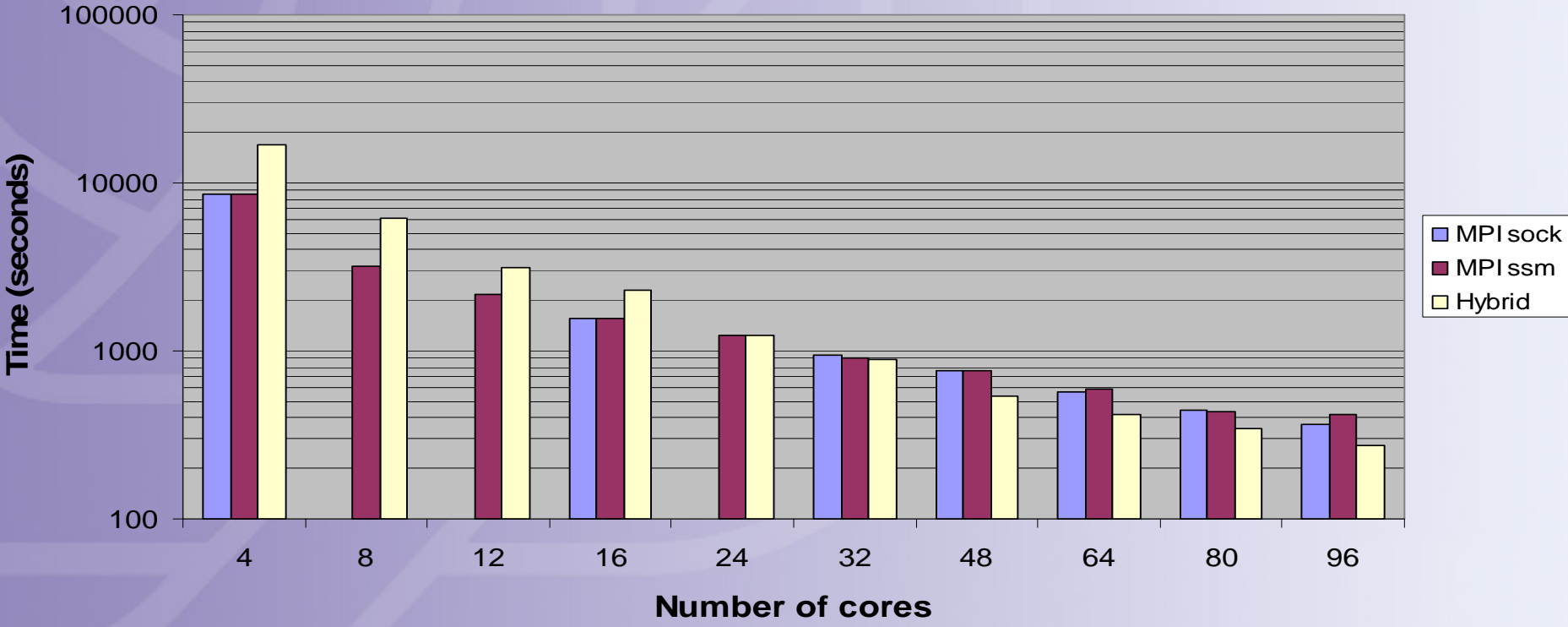
Merlin - 128 Cores



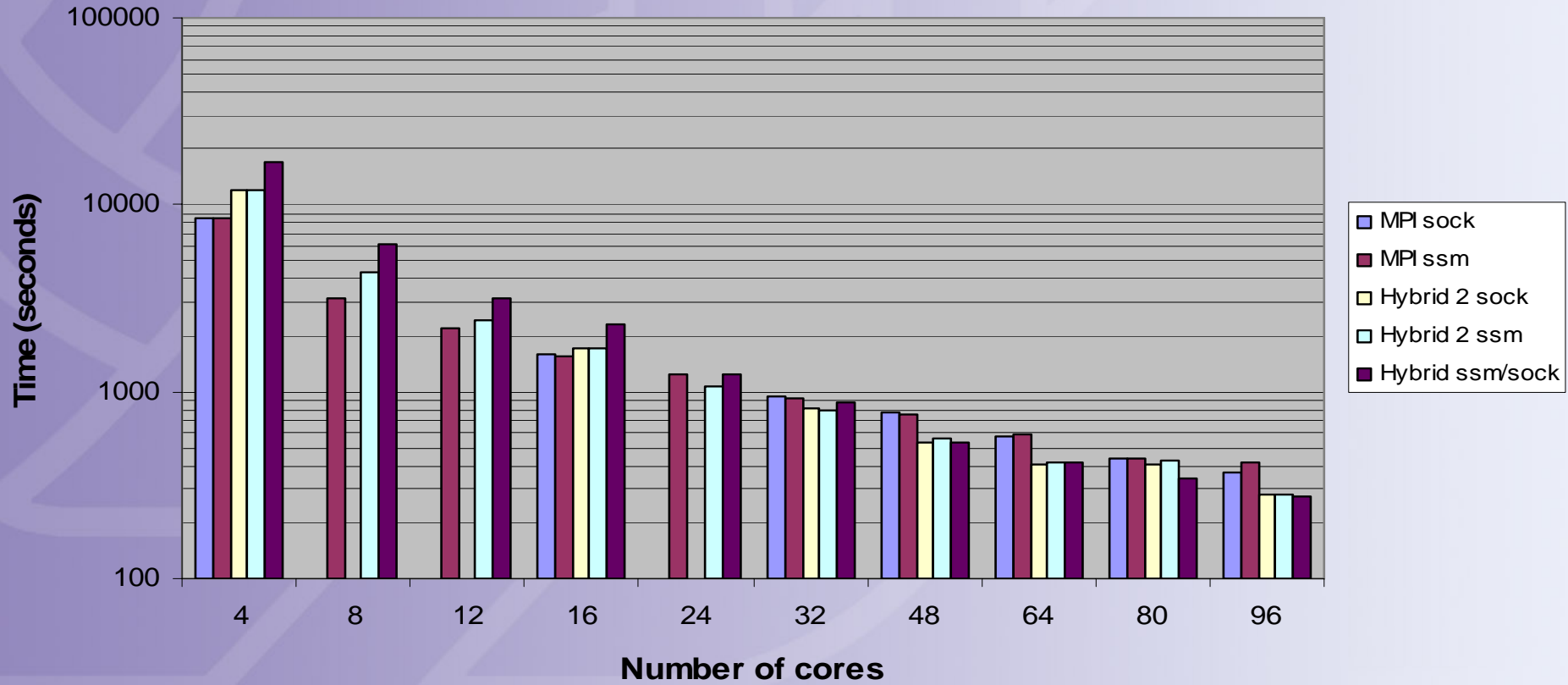
Merlin - Movout Routine - 512 Cores



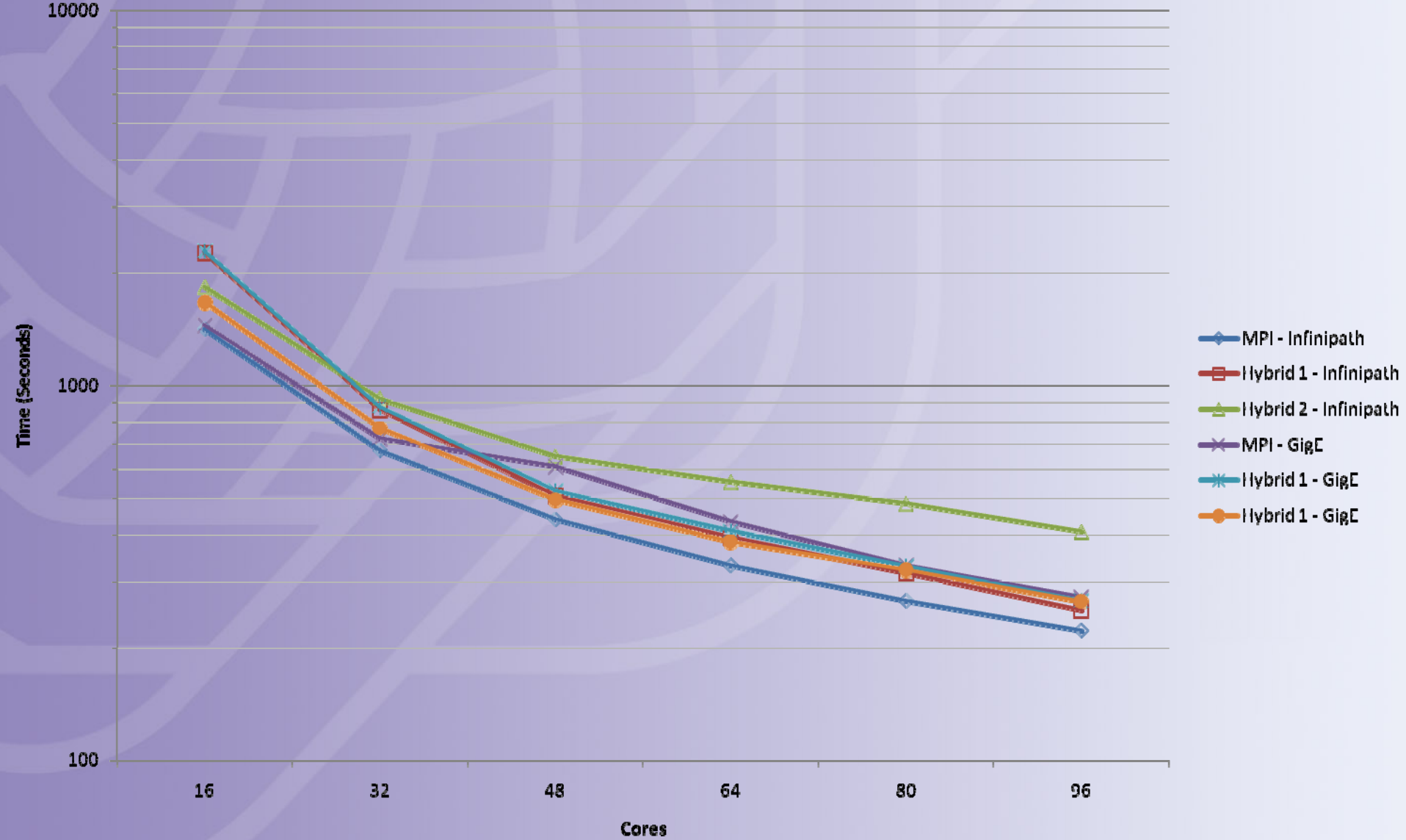
CSEEM64T (GigE) - MPI sock vs ssm vs Hybrid - 44,957,696 particles



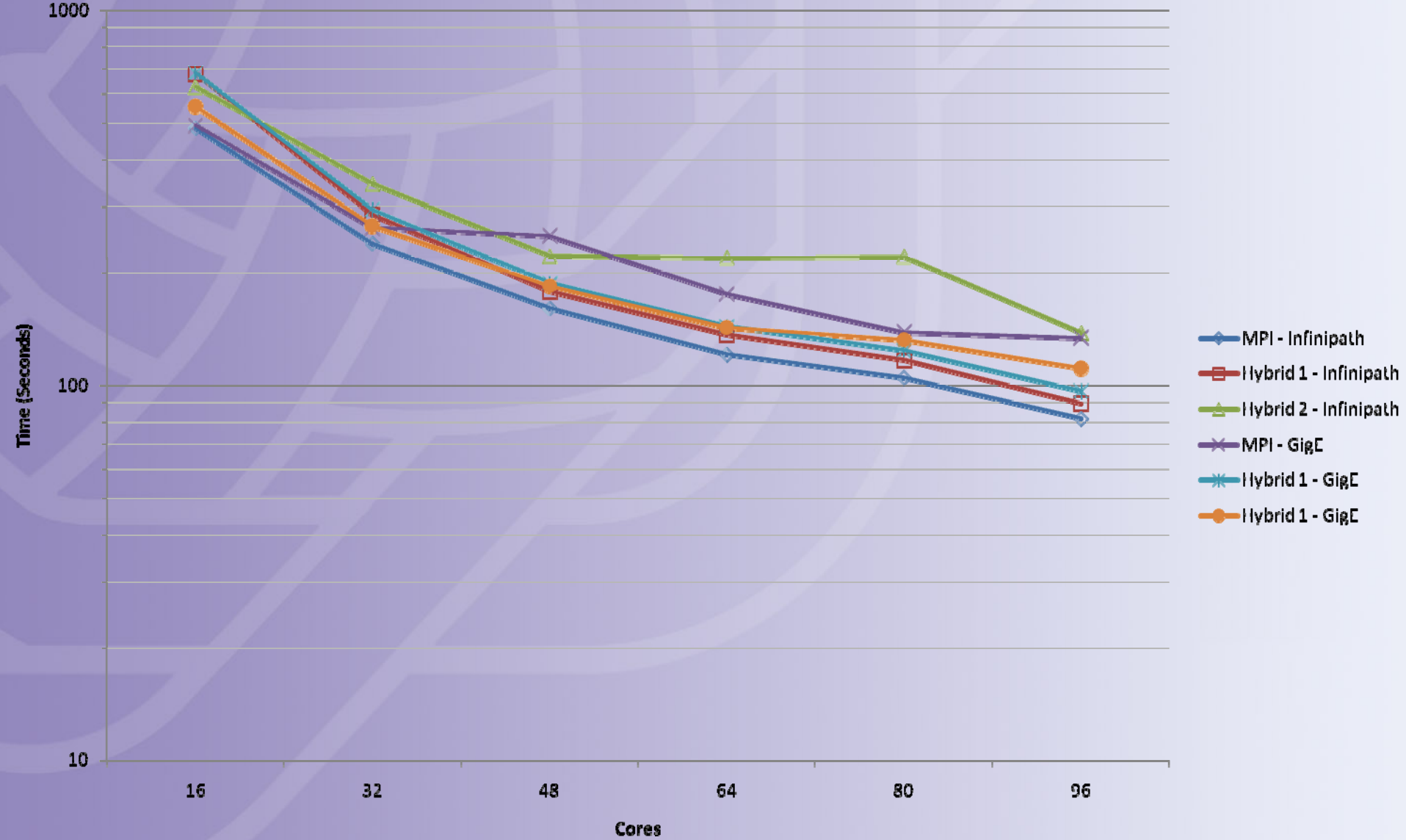
CSEEM64T (GigE) - sock vs ssm - 44,957,696 particles



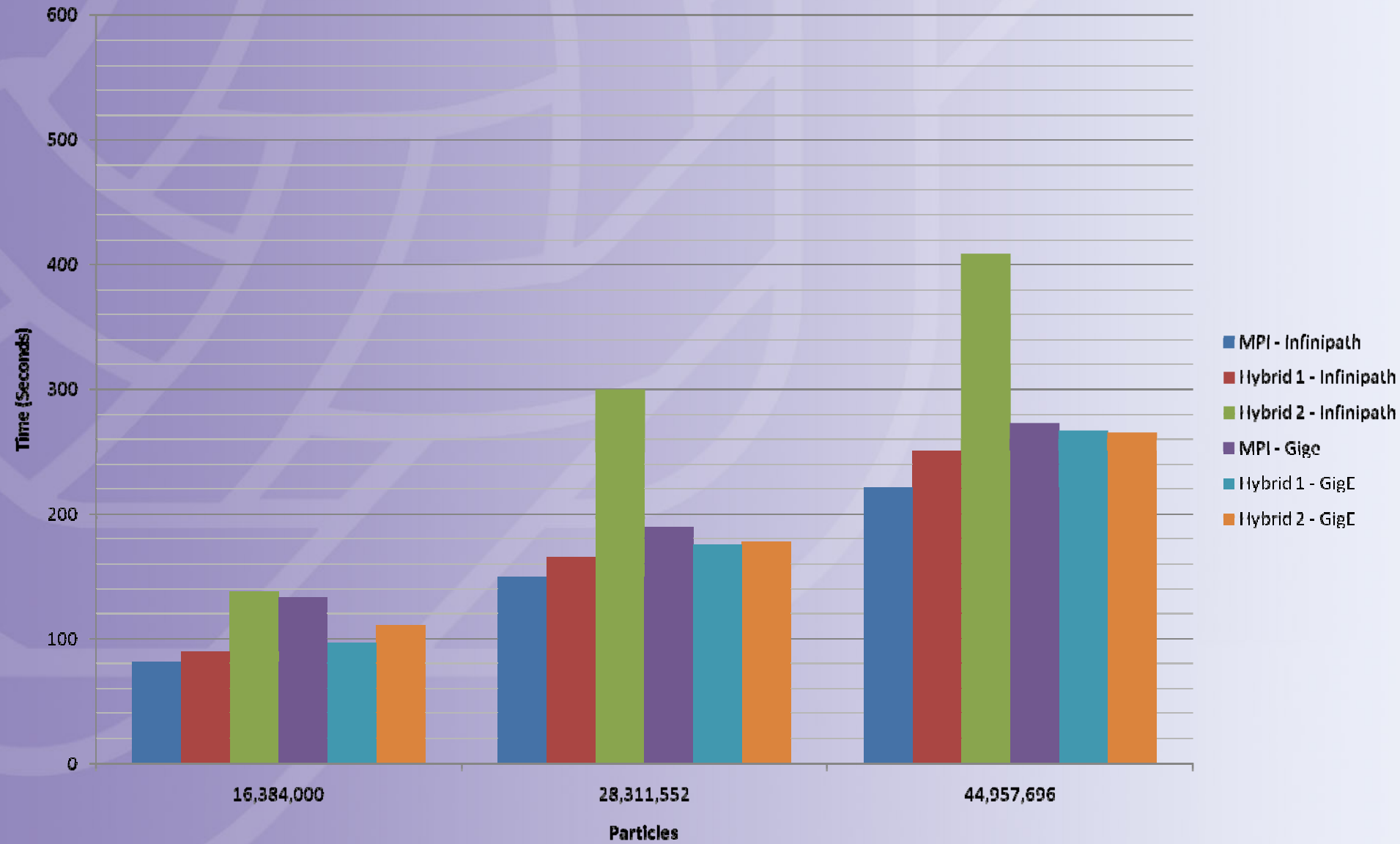
CSEEM64T - 44,957,696 Particles



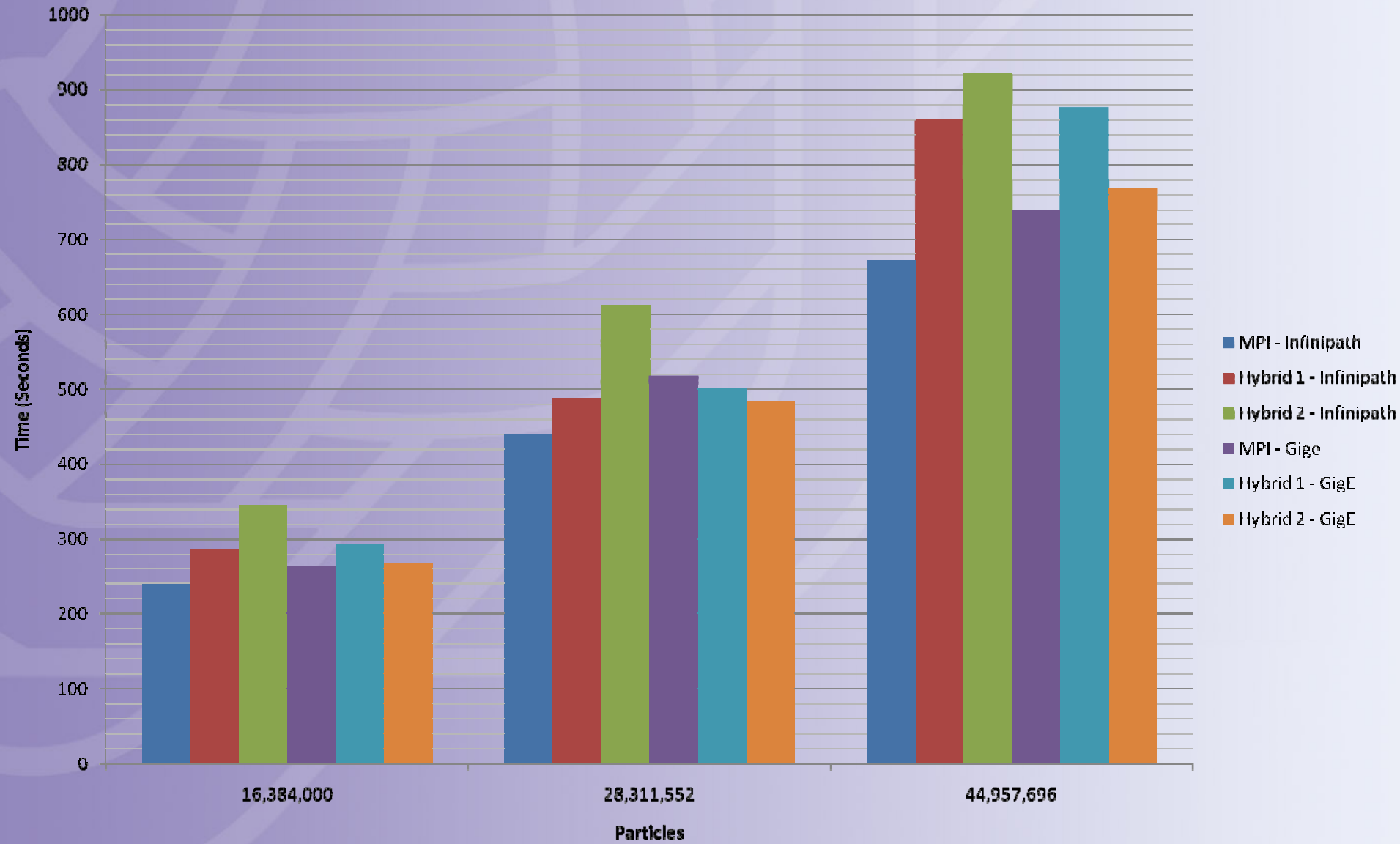
CSEEM64T - 16,384,000 Particles



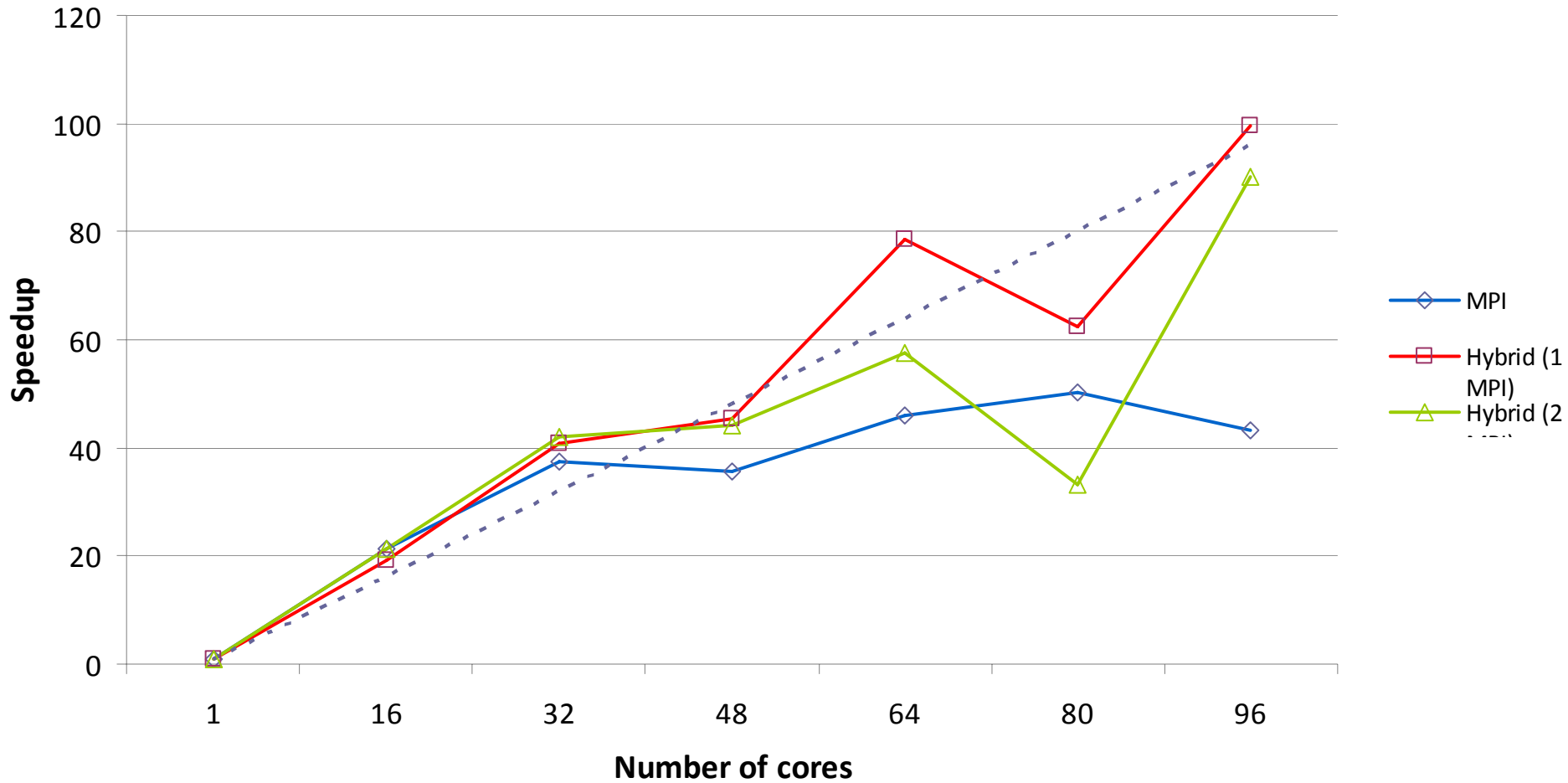
CSEEM64T - 96 Cores



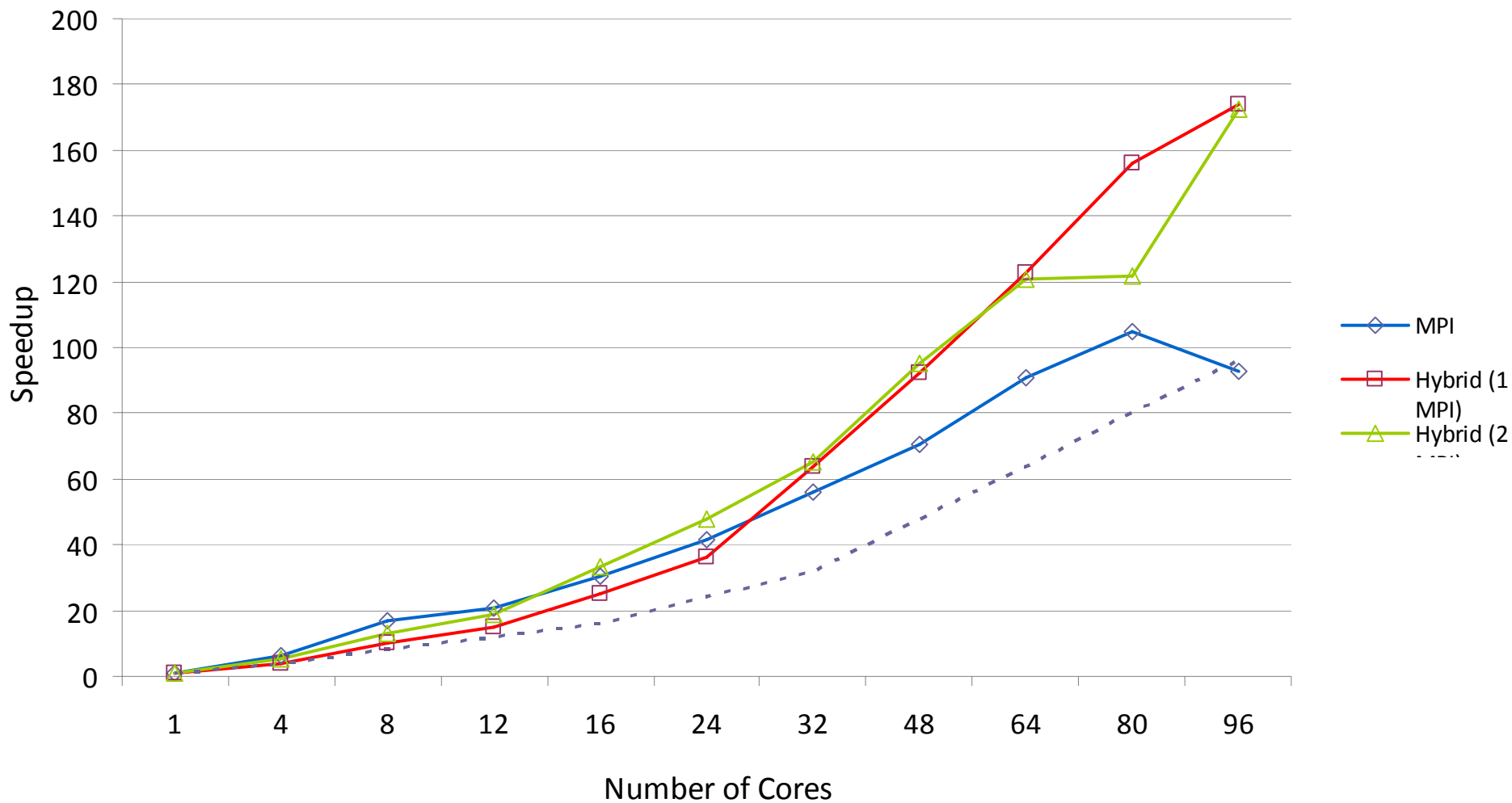
CSEEM64T - 32 Cores



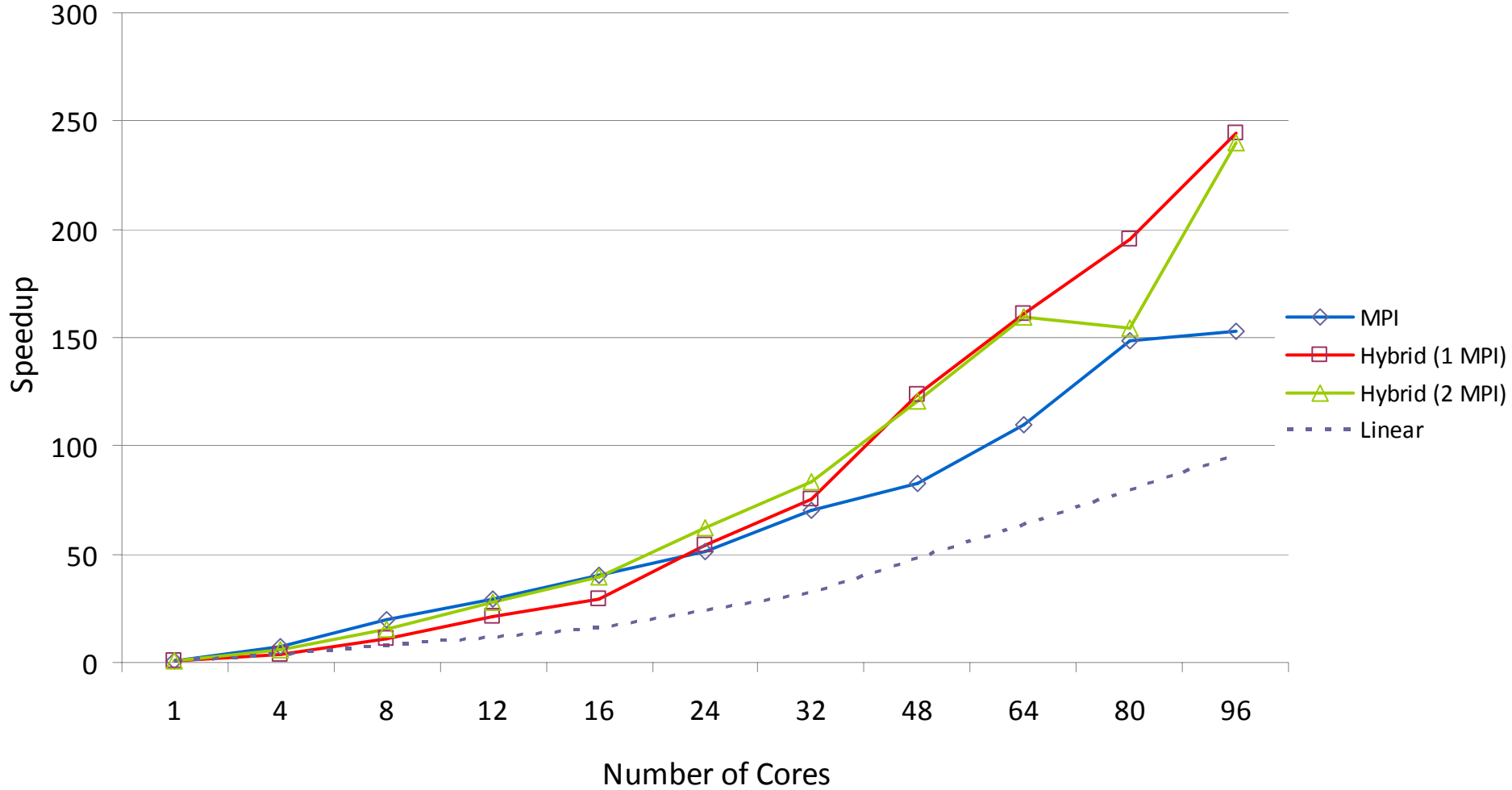
CSEEM64T (GigE) - Speedup - 8,388,608 particles



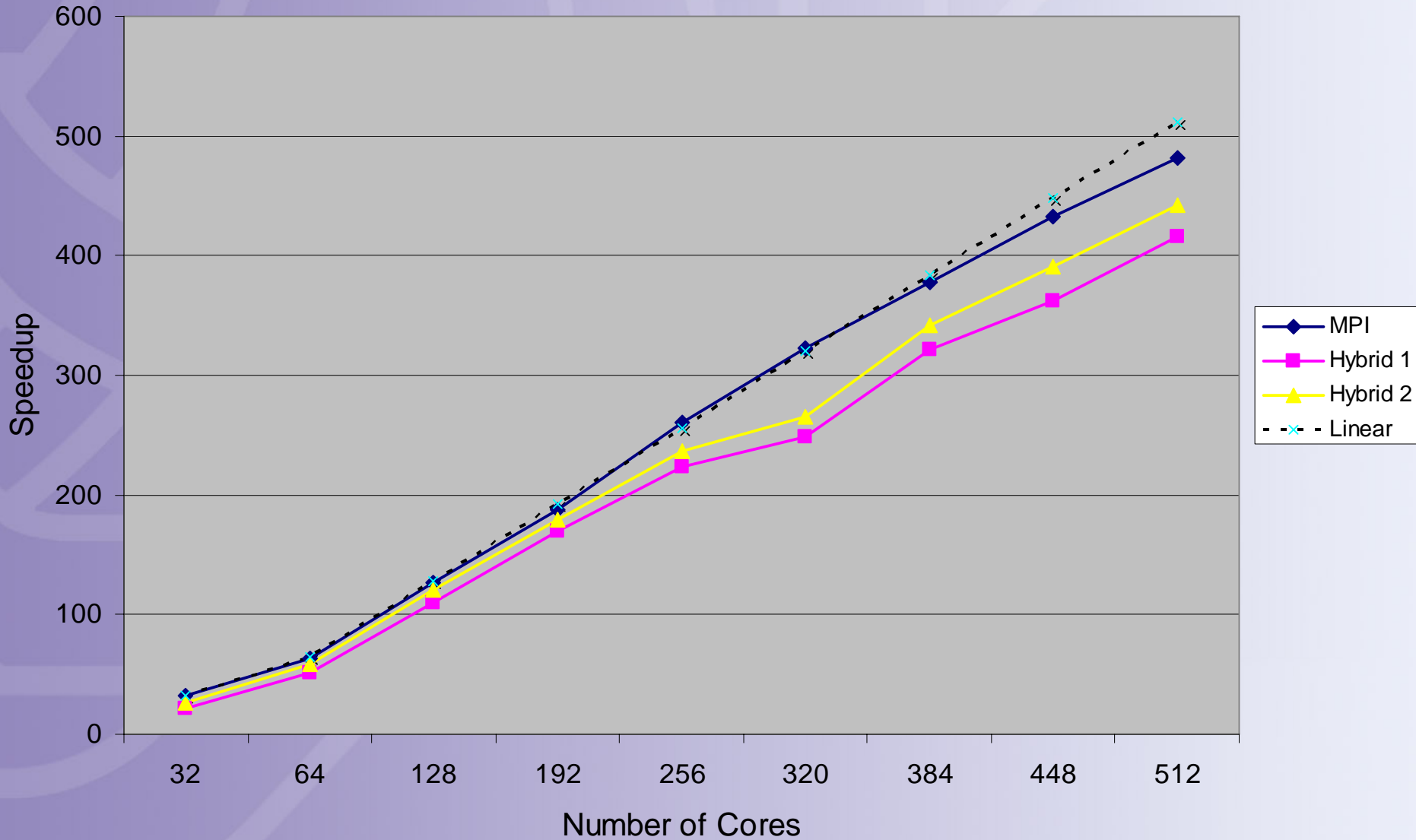
CSEEM64T (GigE) - Speedup - 28,311,552 Particles



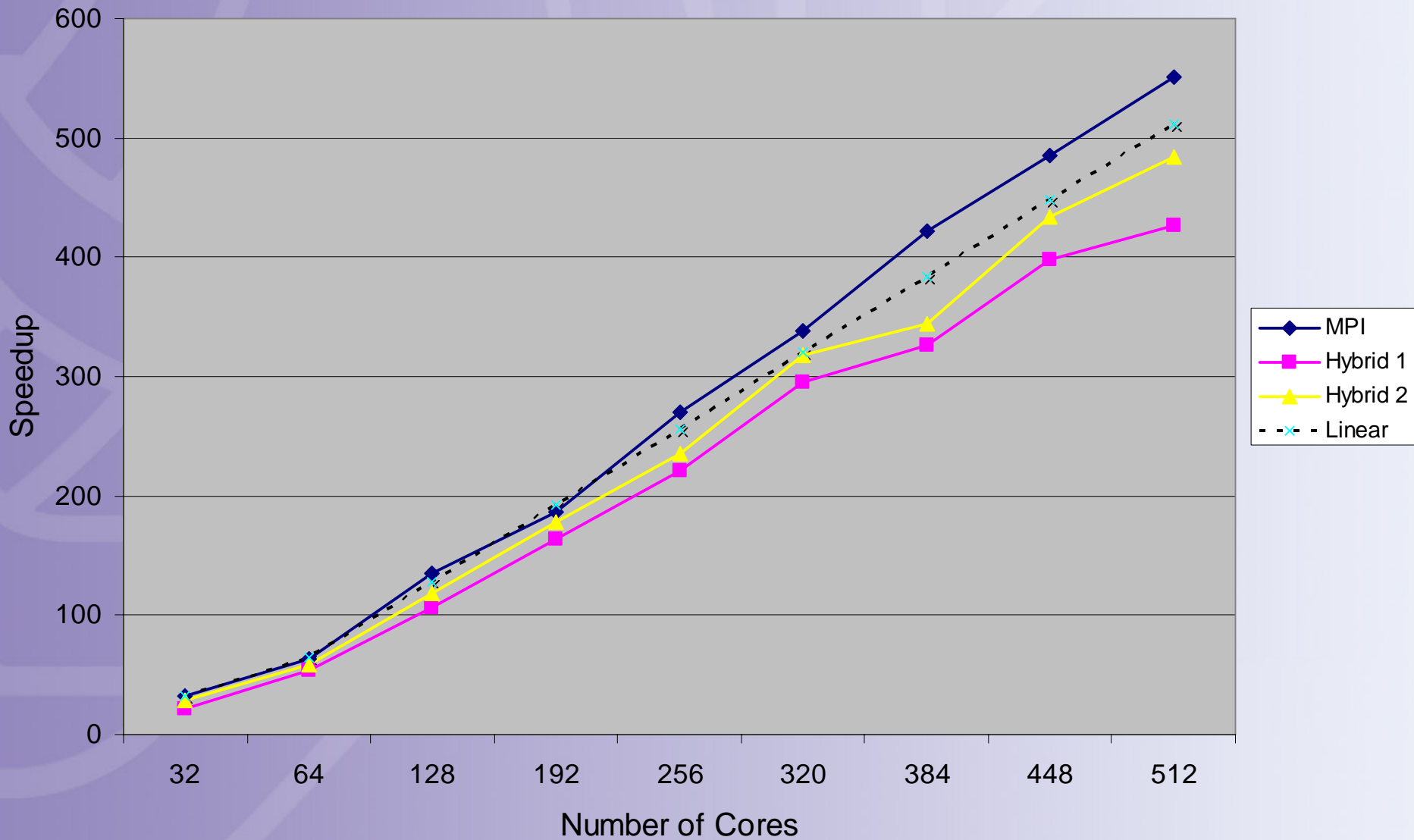
CSEEM64T (GigE) - Speedup - 44,957,696 Particles



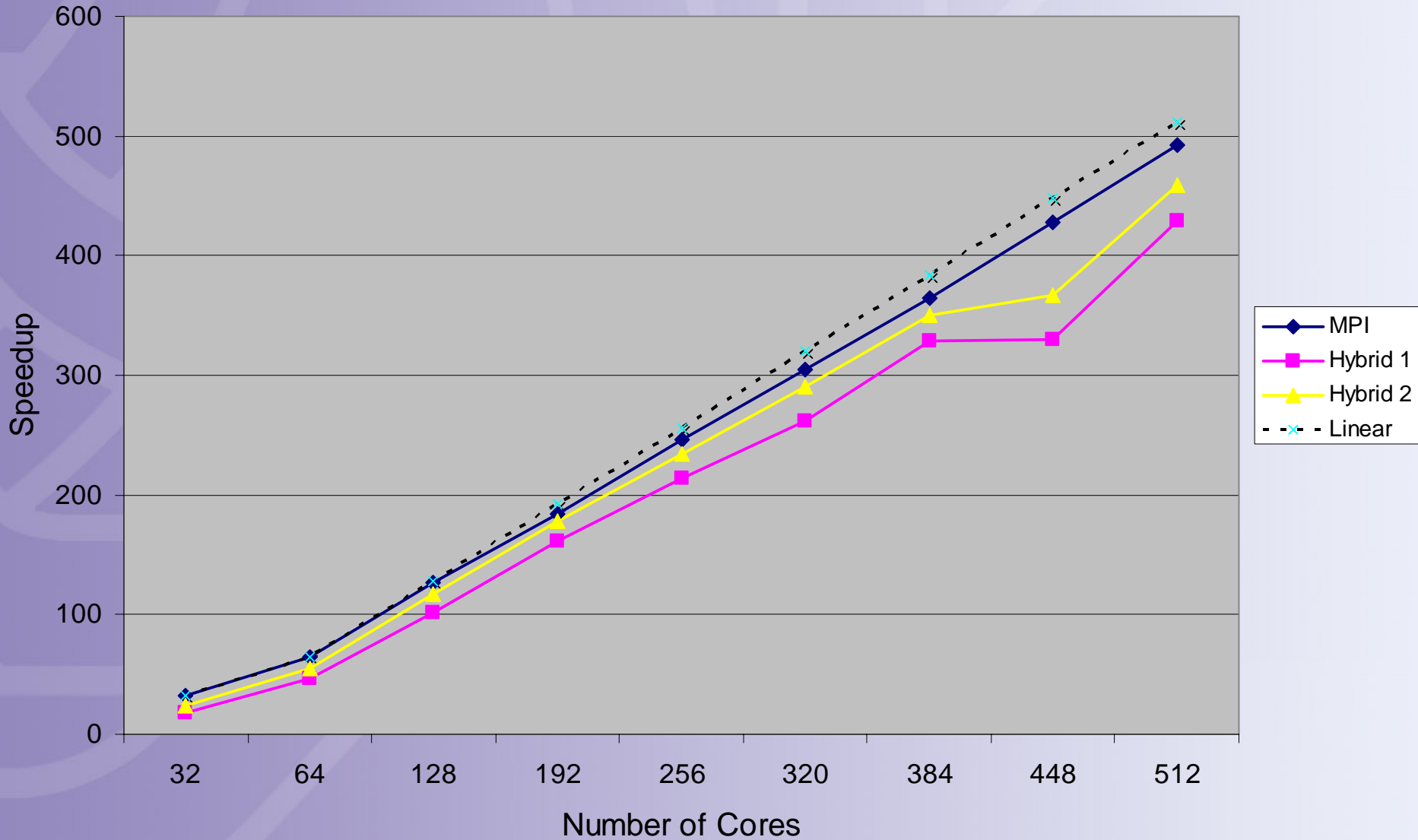
Merlin Infiniband - Speedup- 16,384,000 particles



Merlin Infiniband - Speedup- 28,311,552 particles



Merlin Infiniband - Speedup- 44,957,696 particles



Hybrid Programming - CPMD

- Developed at IBM Zurich from the original Car-Parrinello Code in 1993(www.cpmd.org); Hutter & Curioni
 - Different strategies are followed in parallel implementations of plane-wave / pseudo-potential codes. Parallelization of CPMD was done on different levels. **The central parallelization is based on a distributed-memory coarse-grain algorithm** that is a compromise between load balancing, memory distribution and parallel efficiency.
 - In addition to the basic scheme, **a fine-grain shared-memory parallelization** was implemented. Parallelization on the loop level is achieved by using OpenMP compiler directives and multi-threaded libraries (BLAS and FFT) if available.
 - The two parallelization methods are independent and can be mixed. This yields good performance on distributed computers with shared memory nodes and several thousands of CPUs..

Future Work

- Complete set of timings runs for CSEEM64T with InfiniPath
- Extend timing results to more nodes.
- More analysis of hybrid MD code – would like to develop a better understanding of performance.
- Apply similar techniques to Molpro – a Quantum Chemistry code with sections that may benefit from hybrid parallelism.

Conclusion

- Research is focused on improving performance of real world scientific codes.
 - Hybrid shared memory and message passing programming.
- Previous work has been done on this, but not with modern multi-core clustered hardware.
- Initial results show there may be a benefit to using hybrid programming in certain cases.
- Need to extend initial results to other computational chemistry codes.

Acknowledgments

- It is a pleasure to acknowledge the following:
 - Prof Peter Knowles
 - Funding from the EPSRC: EP/C007832/1



Any Questions?