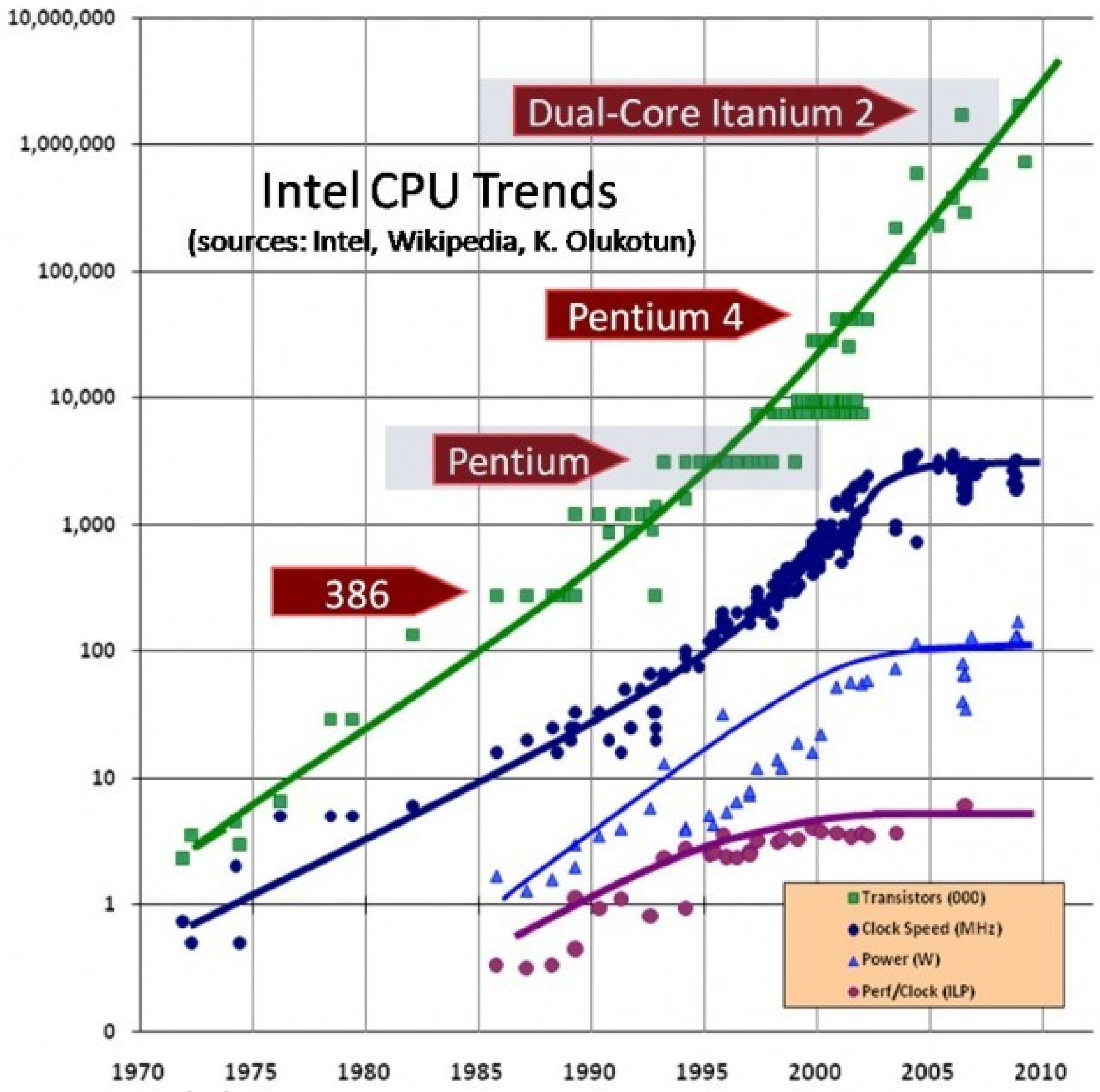
Parallel Paradigms and Techniques

Outline

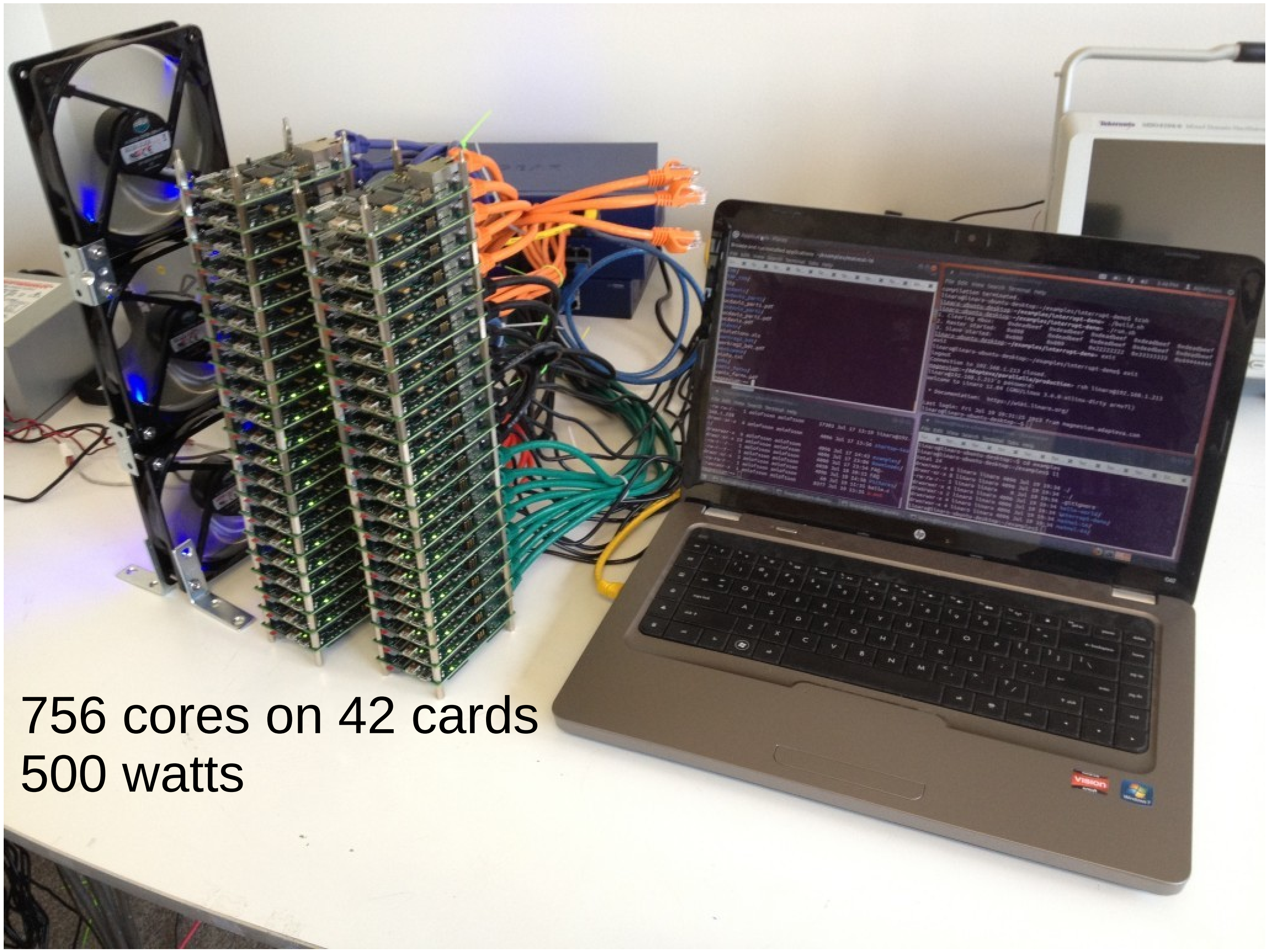
- Motivation
- MPI paradigm
- OpenMP paradigm
- MPI vs. High Level Languages
- Declarative Languages
- Map Reduce and Hadoop
- Shared Global Address Space Languages
- GPUs

Why Parallel?



H. Sutter, “The Free Lunch is Over”, 2005, 2009

The future of supercomputing?



756 cores on 42 cards
500 watts

Parallelism Everywhere

- In the processor:
  - Pipelining
  - Superscalar
- In chip
  - multiple cores
  - hyperthreading
- On the graphics card
  - Pipelining
  - Many cores

MPI Overview

- Defacto standard on all large machines.
- Minimal infrastructure (shared-nothing)
- Run (usually) the same code on many cores
- Unique ID is assigned to each core
- Information is exchanged via library calls
  - Point-to-point
  - Collective

MPI Simple Example

```
#include "mpi.h"
#include <stdio.h>
int main(argc,argv)
int argc;
char *argv[]; {
int  numtasks, rank, len, rc;
char hostname[MPI_MAX_PROCESSOR_NAME];
rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
  printf ("Error starting MPI program. Terminating.\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  }
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on %s\n",
numtasks,rank,hostname);
/*******  do some work *******/
MPI_Finalize();
}
```

MPI and deadlocks

```
#include <stdio.h>
#include "mpi.h"
#define MSGLEN 2048         /* length of message in elements */
#define TAG_A 100
#define TAG_B 200
main( int argc, char **argv ) {
  float message1 [MSGLEN], message2 [MSGLEN];
  int rank, i; MPI_Status status; /* status of communication */
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  for ( i=0; i<MSGLEN; i++ )  {
    message1[i] = 100; message2[i] = -100; }
  if ( rank == 0 )  {
    MPI_Send(message1,MSGLEN,MPI_FLOAT, 1, TAG_A, MPI_COMM_WORLD );
    MPI_Recv(message2,MSGLEN,MPI_FLOAT, 1, TAG_B, MPI_COMM_WORLD,
&status );
  } else if ( rank == 1)  {
    MPI_Send(message1,MSGLEN,MPI_FLOAT,0,TAG_B, MPI_COMM_WORLD );
    MPI_Recv(message2,MSGLEN,MPI_FLOAT,0,TAG_A, MPI_COMM_WORLD,
&status );
  }
  MPI_Finalize();
}
```

Deadlock solutions

- Immediate calls

```
MPI_Isend ( message1, MSGLEN, MPI_FLOAT, dest, send_tag,
             MPI_COMM_WORLD, &request);
MPI_Recv ( message2, MSGLEN, MPI_FLOAT, source, recv_tag,
            MPI_COMM_WORLD, &status );
MPI_Wait ( &request, &status);
```

- Ordered send/receive
- Buffered sends

MPI: collectives

- MPI_Barrier()
- MPI_Bcast()
  - read_params.f90

    ```
    call MPI_BCAST(infile,80,MPI_CHARACTER,0,MPI_COMM_WORLD,ierr)
    ```
- MPI_Scatter()
- MPI_Gather()
- MPI_Reduce()
- <u>Use these over point-to-point if possible</u>

Simple example from RAMSES

```
! Compute global quantities
comm_buffin(1)=mass_loc
comm_buffin(2)=ekin_loc
comm_buffin(3)=eint_loc
call MPI_ALLREDUCE(comm_buffin,comm_buffout,3,
     MPI_DOUBLE_PRECISION,MPI_SUM,&
     &MPI_COMM_WORLD,info)
call MPI_ALLREDUCE(dt_loc,dt_all,1,
     MPI_DOUBLE_PRECISION,MPI_MIN,&
     &MPI_COMM_WORLD,info)
mass_all=comm_buffout(1)
ekin_all=comm_buffout(2)
eint_all=comm_buffout(3)
```

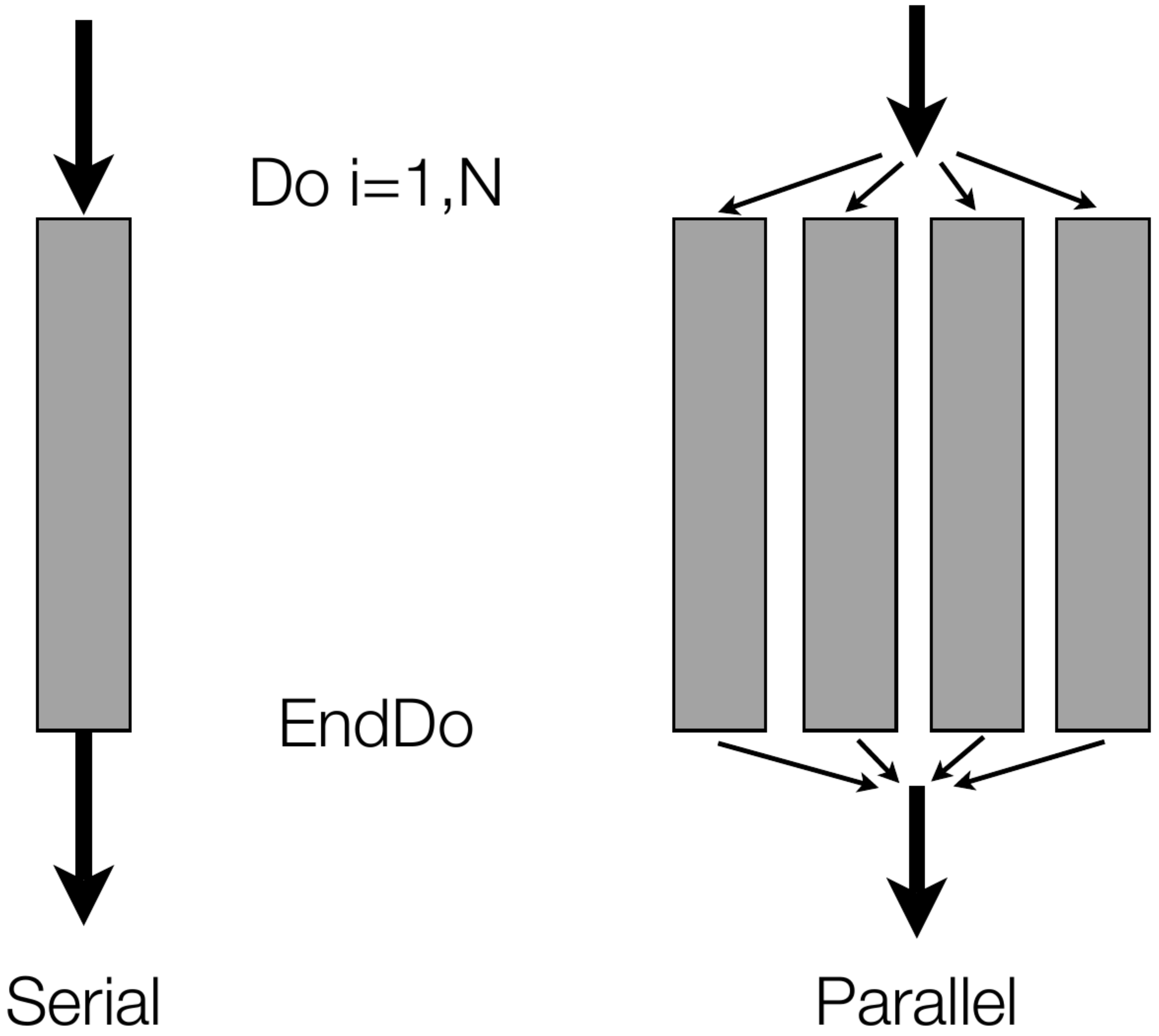
MPI summary

- Standard: you can expect the MPI implementation to be high quality on most supercomputers.
- Exquisite control over data movement and thread execution
  - No conflict on who owns what data: messages must be explicitly sent/received.
- Do you want this much control?

OpenMP Overview

- Easier (generally) to use than MPI
- Details handled by the compiler
- No apparent data movement
  - But machine could move data at a performance cost (NUMA)
- Suitable for single machine (which are almost all multi-core)
- Can be combined with MPI programs

Formation of threads: master thread spawns a team of threads



OpenMP Simple Example

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
  int th_id, nthreads;
#pragma omp parallel private(th_id)
  {
  th_id = omp_get_thread_num();
  printf("Hello World from thread %d\n", th_id);
#pragma omp barrier
  if ( th_id == 0 ) {
    nthreads = omp_get_num_threads();
    printf("There are %d threads\n",nthreads);
    }
  }
  return 0;
}
```

OpenMP: race conditions

```
int sum = 0, loc_sum = 0;
/*forks off the threads and starts the work-
sharing construct*/
#pragma omp parallel for private(w,loc_sum)
                              schedule(static,1)
{
  for(i = 0; i < N; i++) {
    w = i*i;
    loc_sum = loc_sum + w*a[i];
    }
#pragma omp critical
  sum = sum + loc_sum;
  }
printf("\n %d",sum);
```

OpenMP Summary

- Easy to partially parallelize existing code
- Requires shared memory machine and compiler support
- Care must be taken concerning threads read/writing the same memory

Latest Machines

- Blue Waters:
  - 760k cores in 25k nodes; 1.5PB; 3k GPUs
- Stampede
  - 50k+ cores in 3k+ nodes; 150k+ MIC cores
- Titan
  - 300k cores; 18.7k nodes, 18.7k GPUs
- “Path to Exascale”
  - Extremely wide parallelism
  - Heterogeneous

Latest Machines

- Do we need them?
  - Yes!   Planet formation to cosmology.
- Will MPI programs just scale?
  - So far, so good: astrophysicists are smart and industrious.
  - Mixed shared memory/messaging/accelerator gives new level of difficulty.
  - MPI/OpenMP/OpenAcc?

Advanced Parallel Programming

Is there life beyond MPI?

Parallel Programming in MPI

- Good performance
- Highly portable: de facto standard
- Poor match to some architectures
  - Active Messages, Shared Memory
- New machines are hybrid architectures
  - Multicore, Vector, RDMA, GPU, Xeon Phi
- Parallel Assembly?
  - Processors => registers?

Parallel Programming in High Level Languages

- Abstraction allows easy expression of new algorithms
- Low level architecture is hidden (or abstracted)
- Integrated debugging/performance tools
- Sometimes a poor mapping of algorithm onto the language
- Steep learning curve

Parallel Programming Hierarchy

- Decomposition of computation into parallel components
  - Parallelizing compiler, Chapel
- Mapping of components to processors
  - Charm++
- Scheduling of components
  - OpenMP, HPF
- Expressing the above in data movement and thread execution
  - MPI

Language Requirements

- General Purpose
- Expressive for application domain
  - Including matching representations: *(a + i) vs a[i]
- High Level
- Efficiency/obvious cost model
- Modularity and Reusability
  - Context independent libraries
  - Similar to/interoperable with existing languages

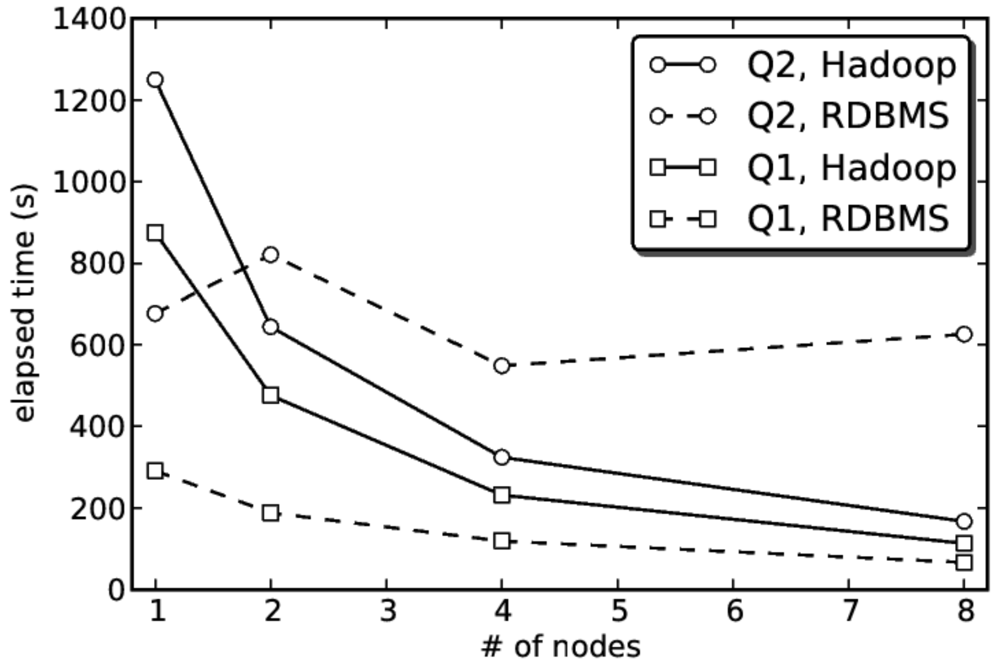
Declarative Languages

- SQL example:

  SELECT SUM(L_Bol) FROM stars WHERE tform > 12.0
- Performance through abstraction
- Limited expressivity, otherwise
  - Complicated
  - Slow (UDF)
- Optimizer is critical

Map Reduce & Hadoop

- Map: function produces (key, value) pairs
- Reduce: collects Map output
- Pig: SQL-like query language
- Effective data reduction framework
- Suitability for HPC?



Array Languages: e.g., CAF, UPC

- Arrays distributed across images
- Each processor can access data on other processors via co-array syntax

```
call sync_all(/up, down/)

new_A(1:ncol) = new_A(1:ncol)
+A(1:ncol)[up] + A(1:ncol)[down]

call sync_all(/up, down/)
```

- Easy expression of array model
- Cost transparent

Charm++

- Parallel library and run-time system
- User decomposes problem into parallel components
- Run-time maps components onto processors; schedules execution
- Not a language: more work for the user.

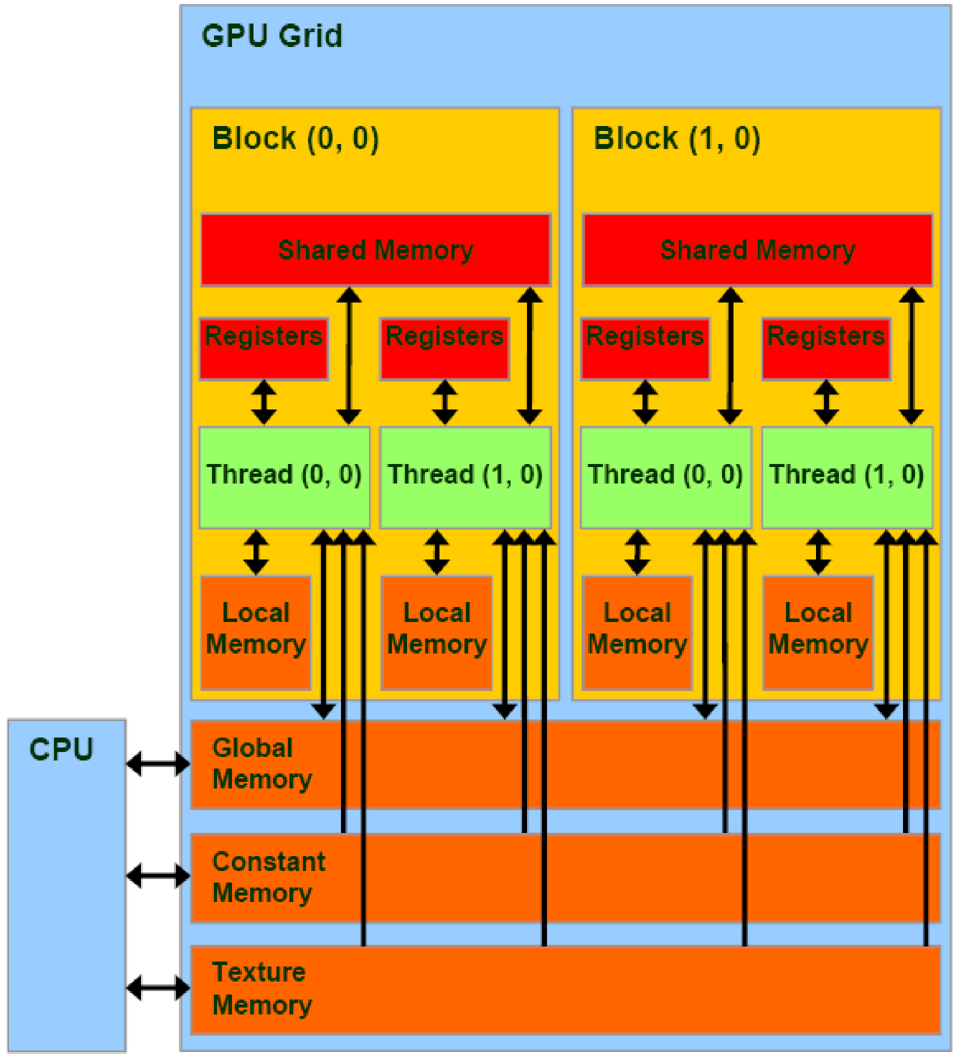
GPUs and Parallelism

- GPU: the ultimate SIMD machine
  - 100,000+ vertices
    - One operation: coordinate transform
  - 1,000,000+ pixels
    - One operation: shade to RGBA color
- Independent input data
  - No data hazards
- No control hazards
- Regular memory access

CUDA Programming model

- Compute device
  - Coprocessor to the CPU
  - Has own RAM
- Data-parallel code is run in *kernels* on many threads
- Threads:
  - Lightweight, little creation overhead
  - Need 1000s for full efficiency
- Lots of work, limited data transfer for good performance

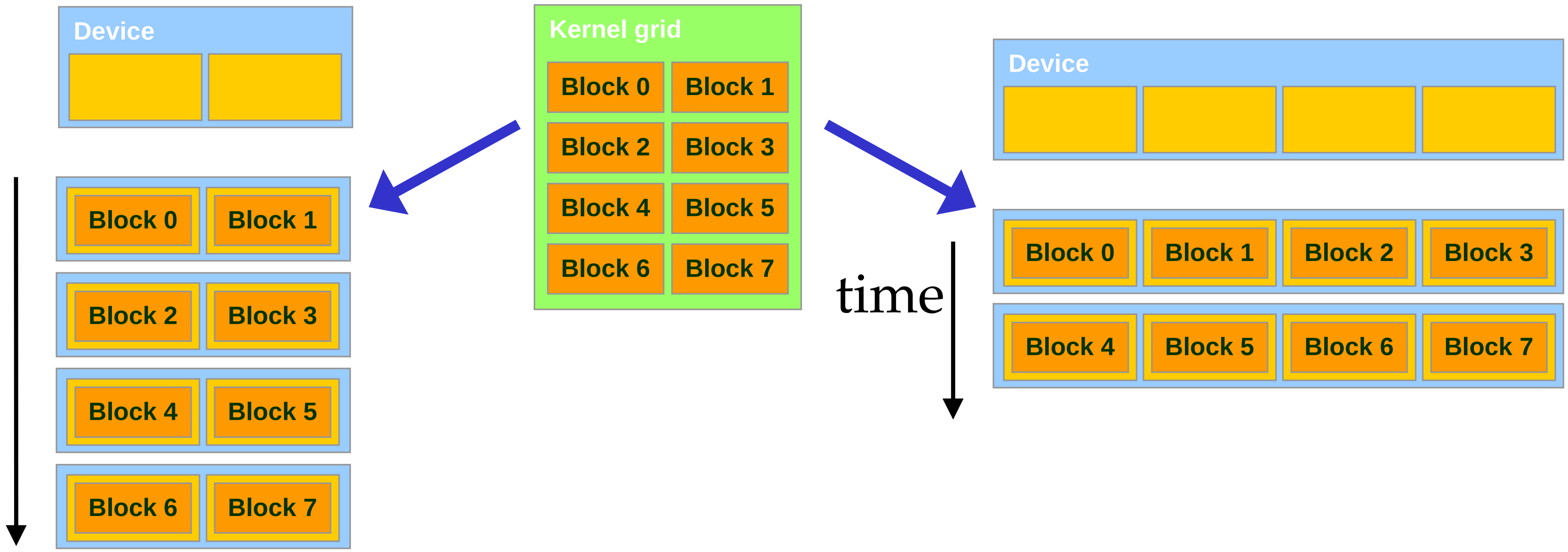
CUDA memory model



Global memory allocated by host

Model to Device Mapping

- Hardware assigns blocks to available processors
  - But concurrency can be limited by hardware resources.



CUDA Host-Device Data Transfer

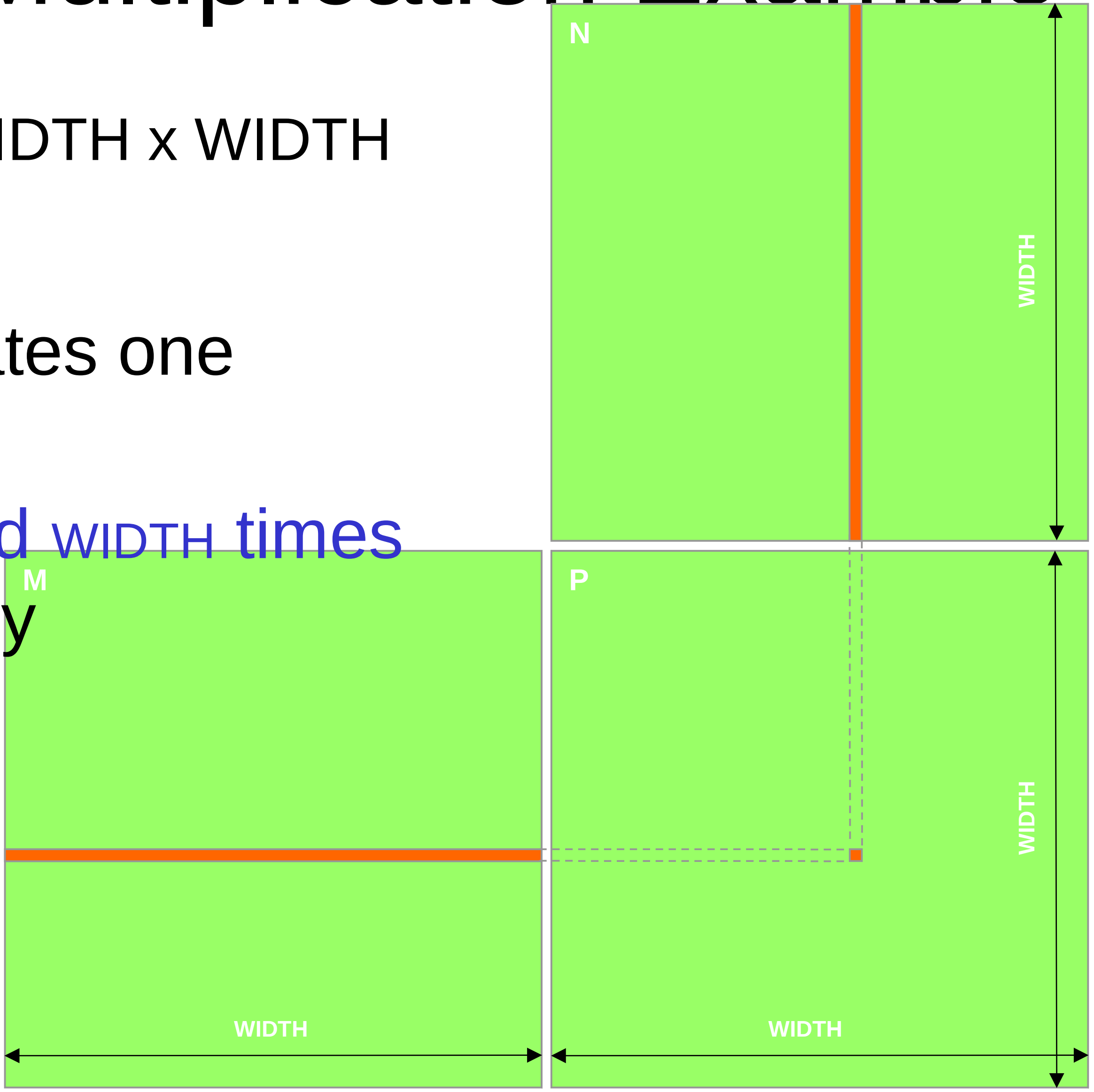
- Example:
  - Transfer a float array
  - M is in host memory and Md is in device memory
    - Md allocated with CudaMalloc()
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```

Programming Model: Square Matrix Multiplication Example

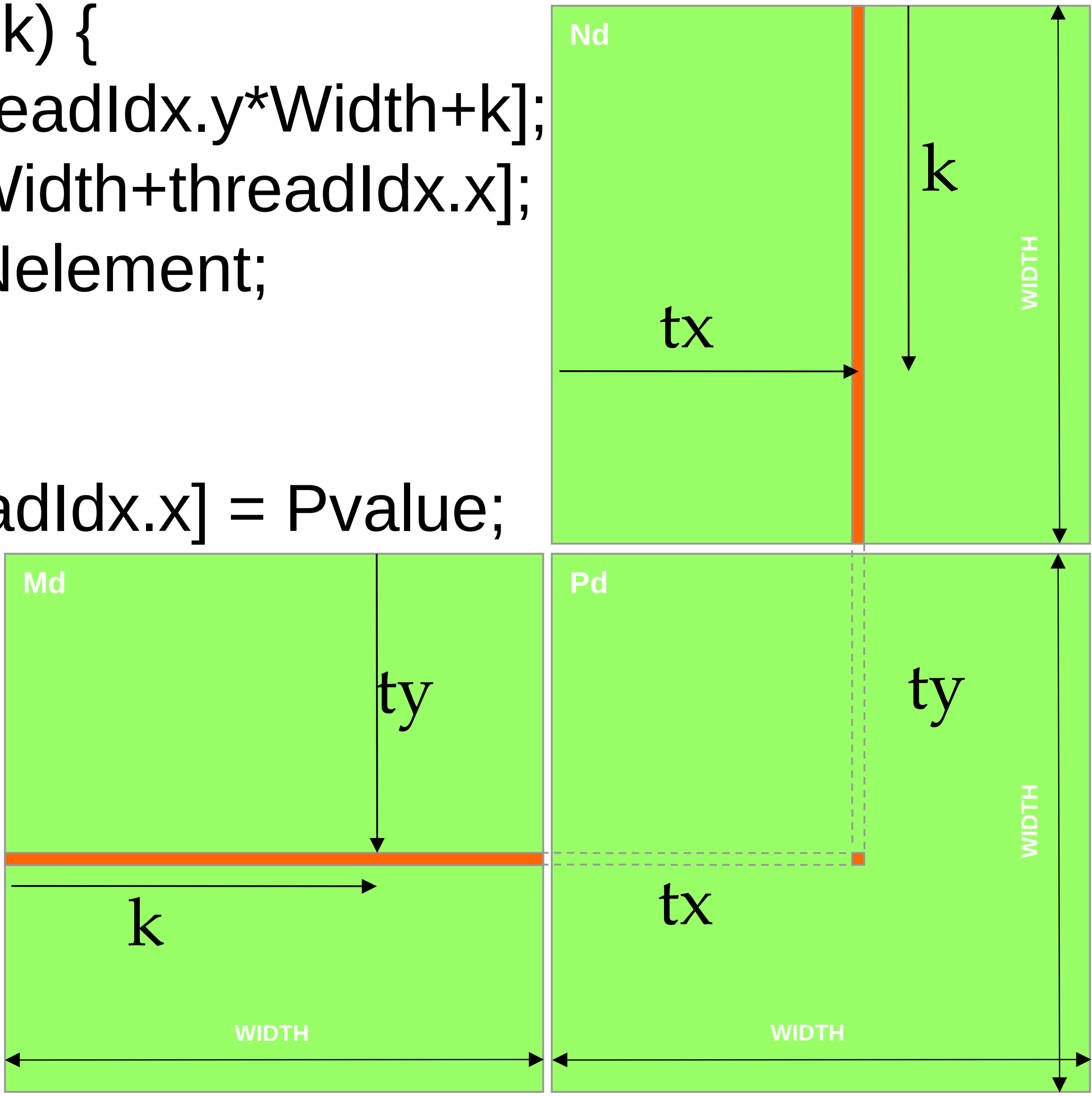
- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory



Kernel Function

```
    Pvalue = 0;
    for (int k = 0; k < Width; ++k) {
        float Melement = Md[threadIdx.y*Width+k];
        float Nelement = Nd[k*Width+threadIdx.x];
        Pvalue += Melement * Nelement;
    }

   Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```



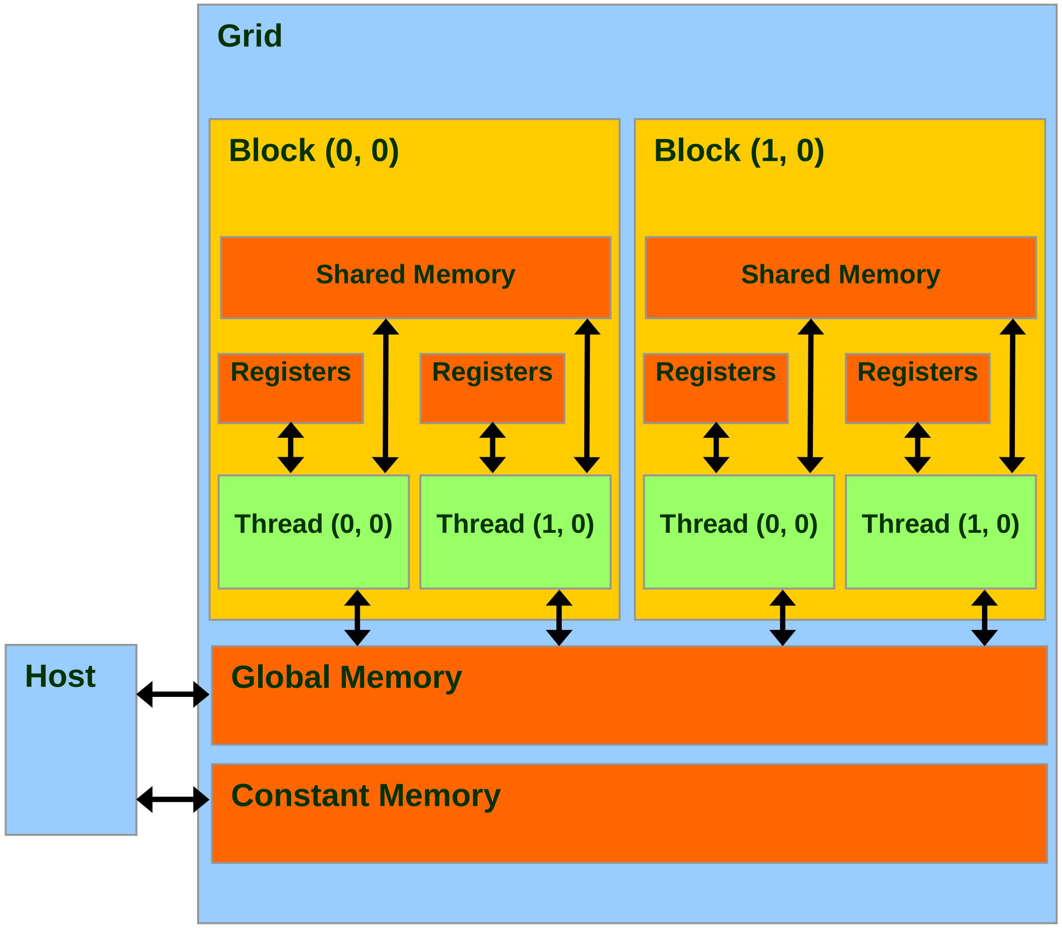
Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
  dim3 dimGrid(1, 1);
  dim3 dimBlock(Width, Width);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

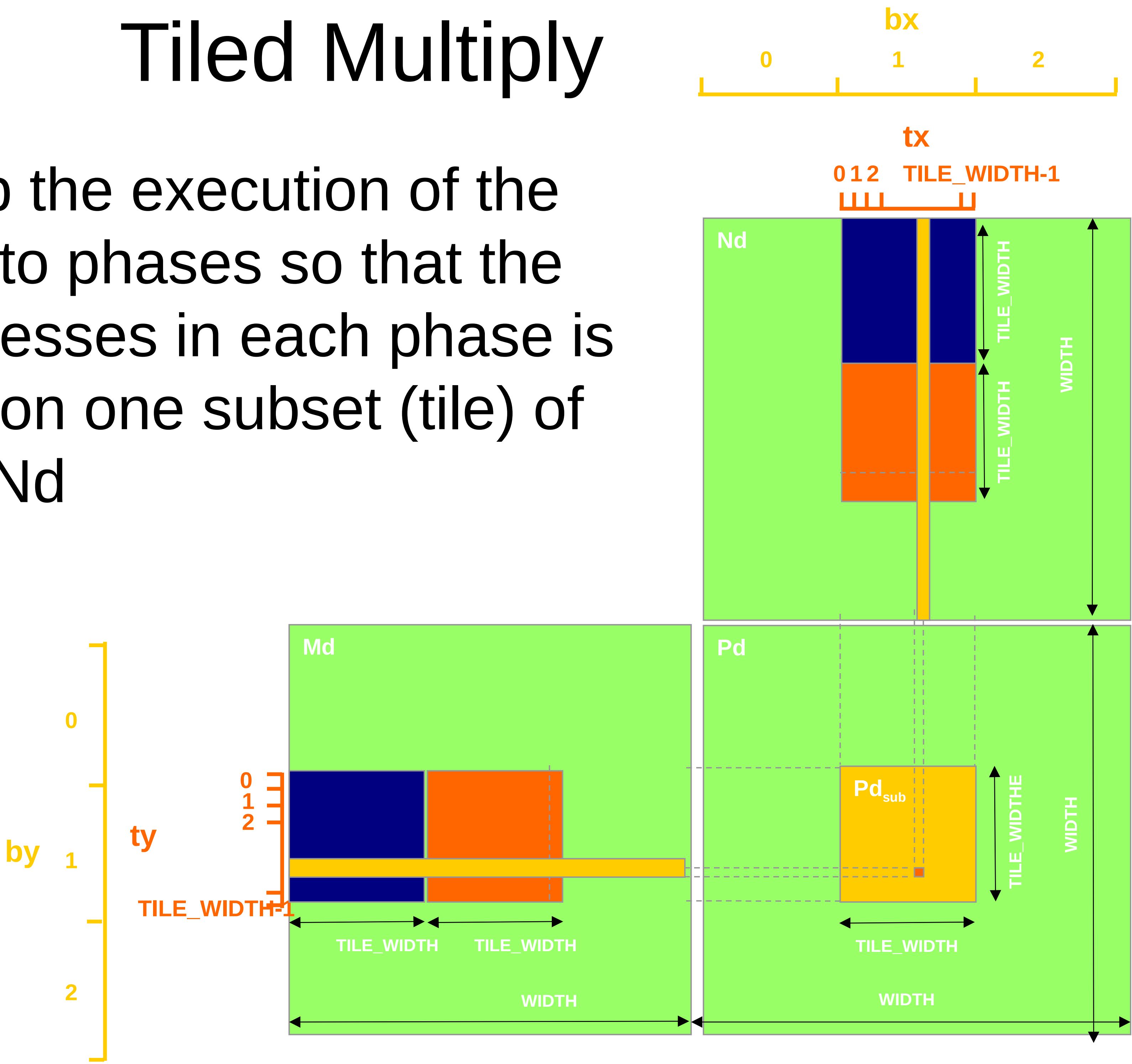
Implementation of CUDA Memories

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread local memory
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**



Tiled Multiply

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd



Tiled Matrix Multiplication Kernel

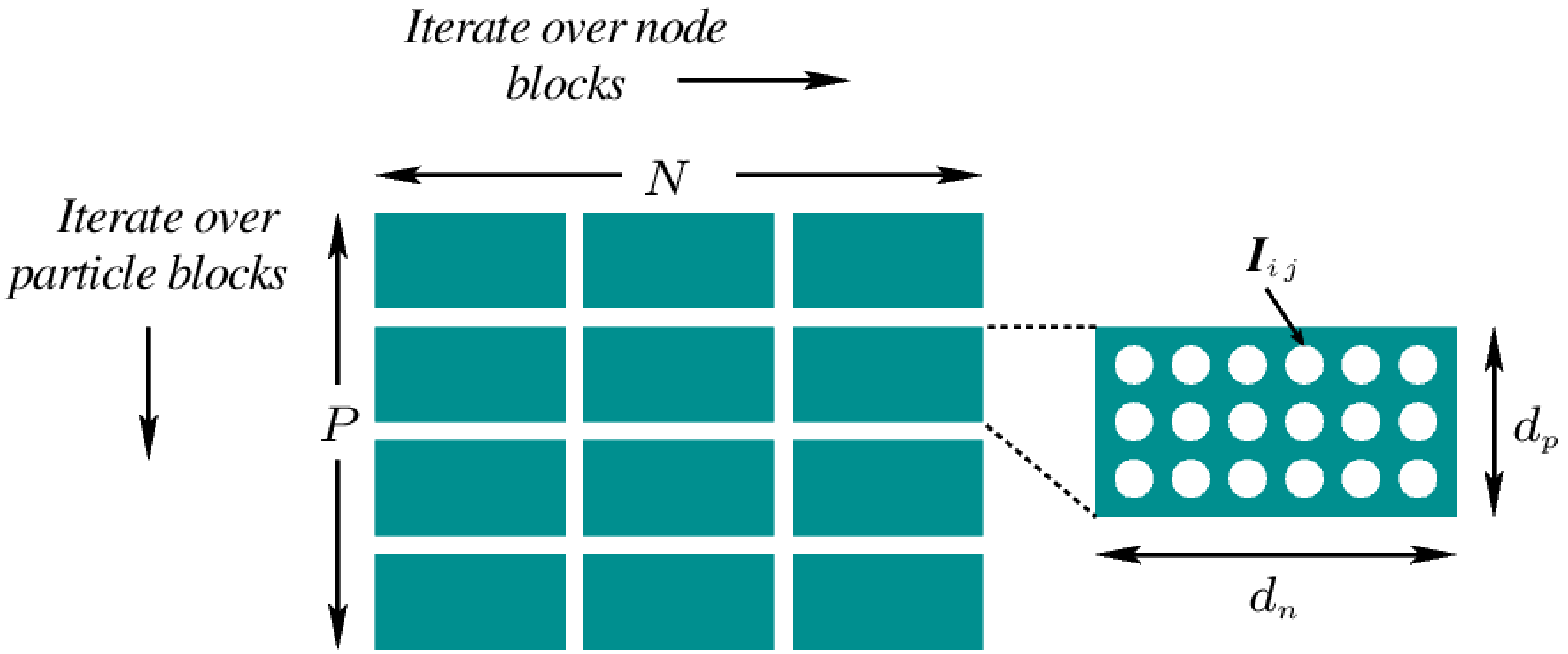
```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  __shared __float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared __float Nds[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;
// Identify the row and column of the Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Collaborative loading of Md and Nd tiles into shared memory
      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
      Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
__syncthreads();
    for (int k = 0; k < TILE_WIDTH; ++k)
      Pvalue += Mds[ty][k] * Nds[k][tx];
    __syncthreads();
    }
 Pd[Row*Width + Col] = Pvalue;
}
```

Organizing the Force for the GPU

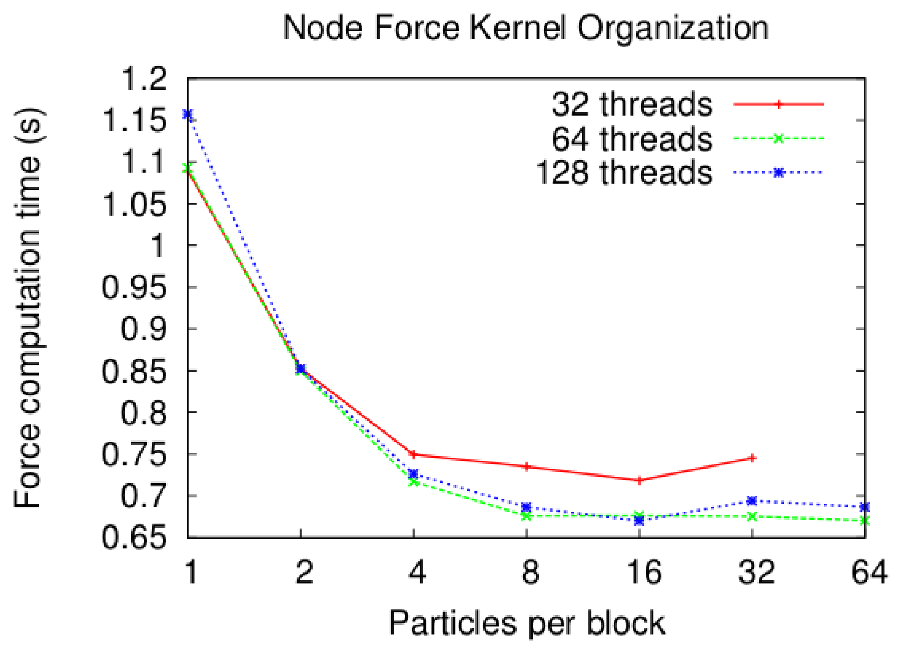
- For each bucket, treewalk produces list of nodes and particles to be interacted
- Interaction on p_i from n_j is I_{ij}
- **I** is P by N matrix where
  - P = particles in bucket
  - N = number of interacting nodes (1000s)
- Organize into blocks of size T < N x P

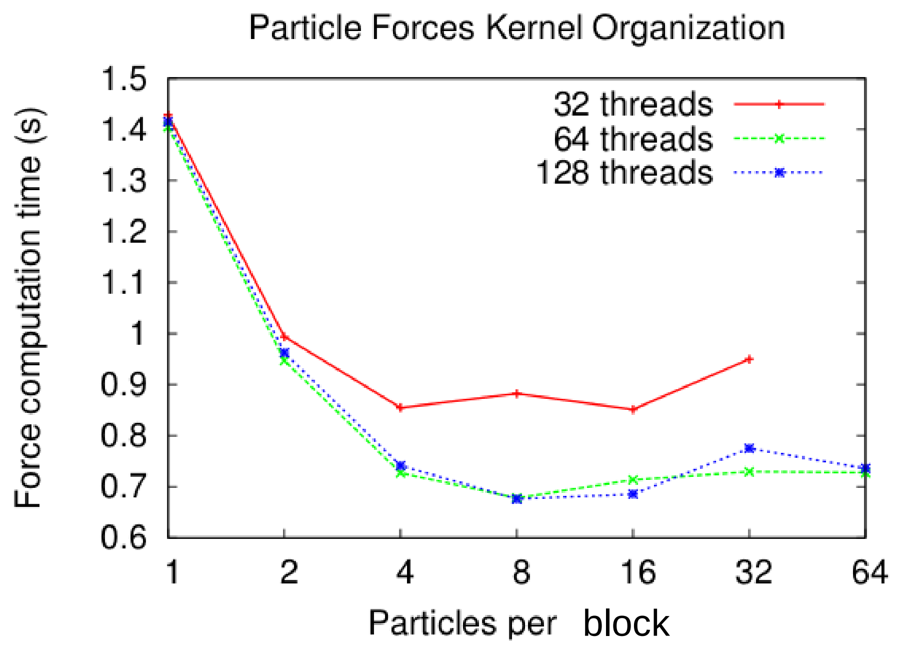
Force Kernel Optimization

More particles->fewer loads
More particles->larger shared memory use
Fewer executing blocks



Kernel Optimization Results





Optimum at 128 threads, 16 particles, 8 nodes/block

GPU summary

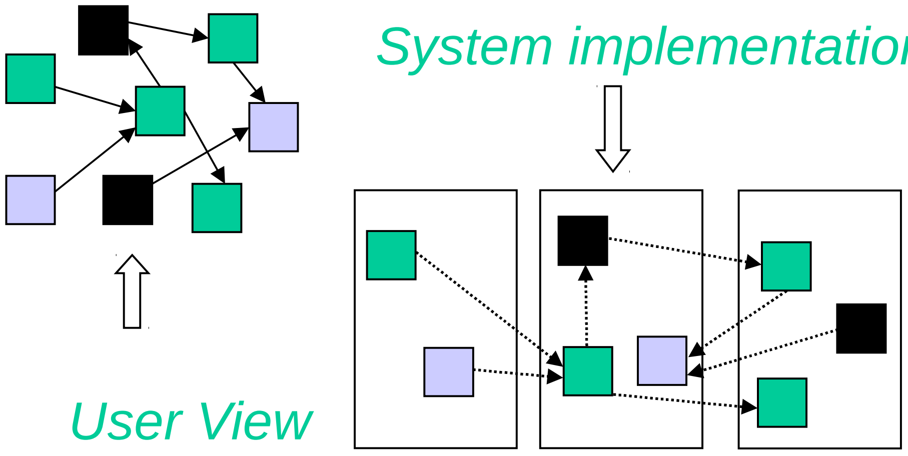
- GPUs enable parallelism at a very fine level (similar to OpenMP)
- CUDA/OpenCL enables portable (device independent) codes
- Good performance requires tuning to the device

Charm++: Migratable Objects

Programmer: **[Over] decomposition into virtual processors**

Runtime: **Assigns VPs to processors**

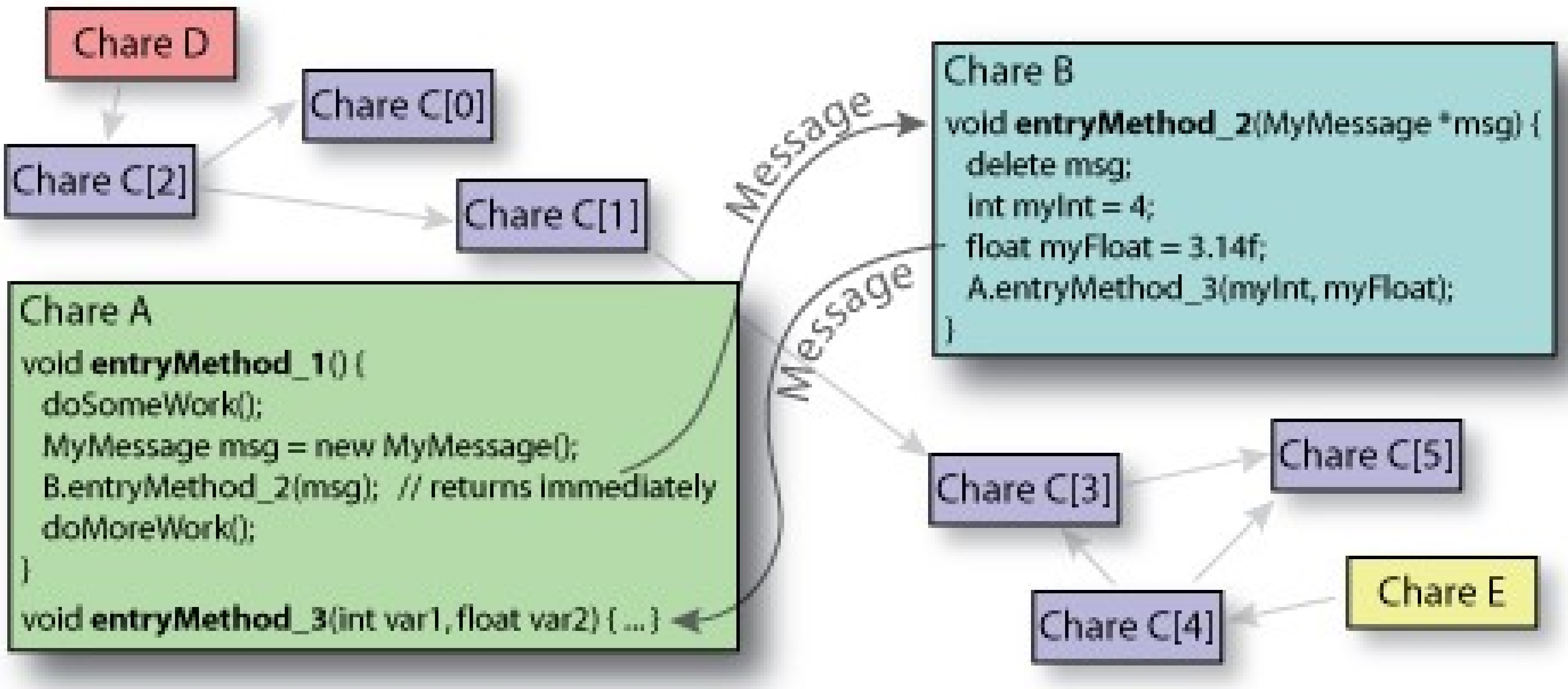
Enables *adaptive runtime strategies*



Benefits

- Software engineering
  - Number of virtual processors can be independently controlled
  - Separate VPs for different modules
- Message driven execution
  - Adaptive overlap of communication
- Dynamic mapping
  - Heterogeneous clusters
    - Vacate, adjust to speed, share
  - Automatic checkpointing
  - Change set of processors used
  - Automatic dynamic load balancing
  - Communication optimization

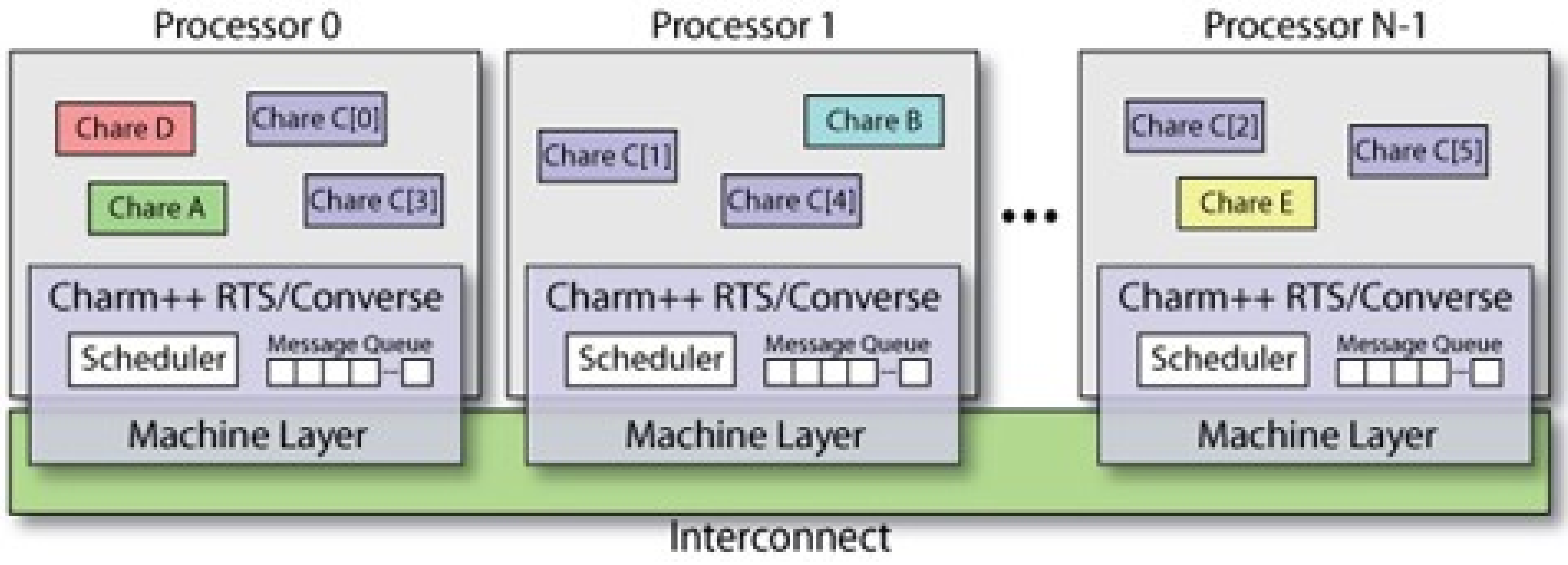
User view



```
void entryMethod_1() {
  doSomeWork();
  MyMessage msg = new MyMessage();
  B.entryMethod_2(msg);   // returns immediately
  doMoreWork();
}
void entryMethod_3(int var1, float var2) { ... }
```

```
void entryMethod_2(MyMessage *msg) {
  delete msg;
  int myInt = 4;
  float myFloat = 3.14f;
  A.entryMethod_3(myInt, myFloat);
}
```

System View



Processor 0
Processor 1
Processor N-1
Chare D
Chare C[0]
Chare A
Chare C[3]
Chare C[1]
Chare B
Chare C[4]
Chare C[2]
Chare C[5]
Chare E
Charm++ RTS/Converse
Scheduler
Message Queue
Machine Layer
Interconnect

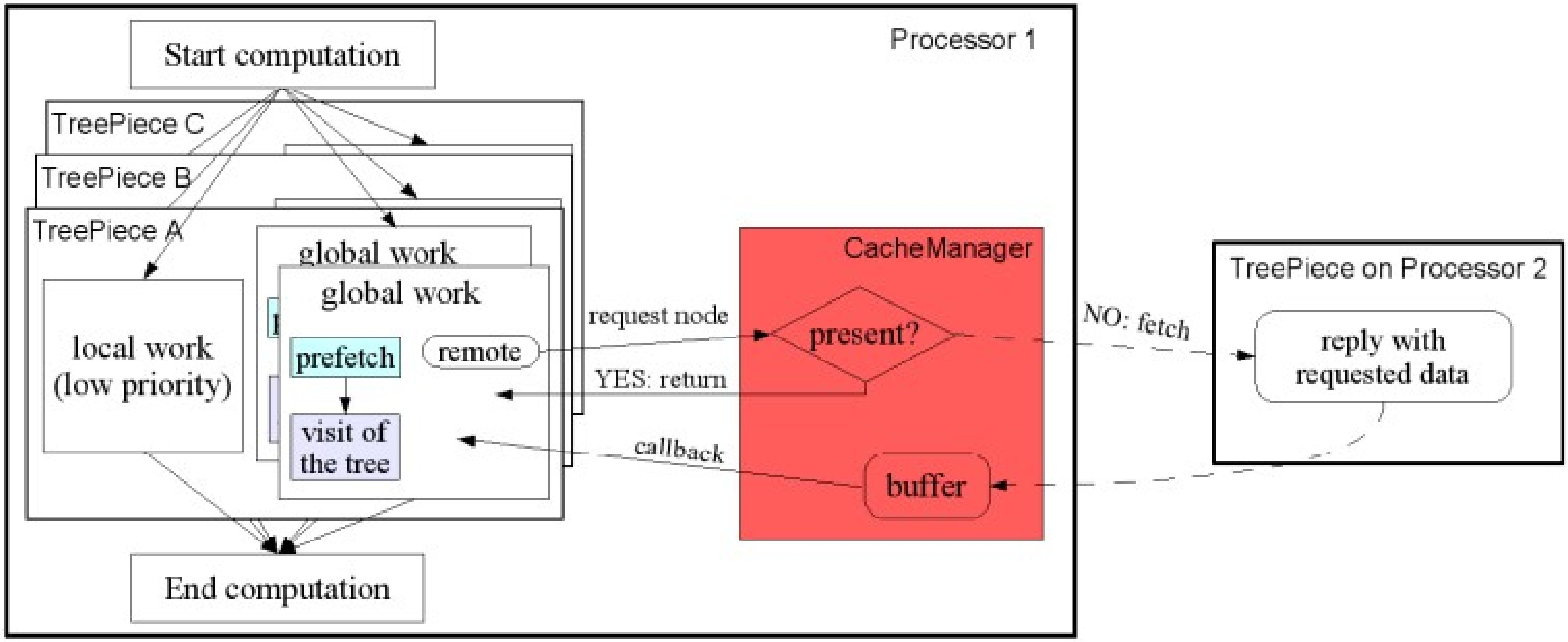
Gravity Implementations

- Standard Tree-code
- “Send”: distribute particles to tree nodes as the walk proceeds.
  - Naturally expressed in Charm++
  - Extremely communication intensive
- “Cache”: request treenodes from off processor as they are needed.
  - More complicated programming
  - “Cache” is now part of the language

ChaNGa Features

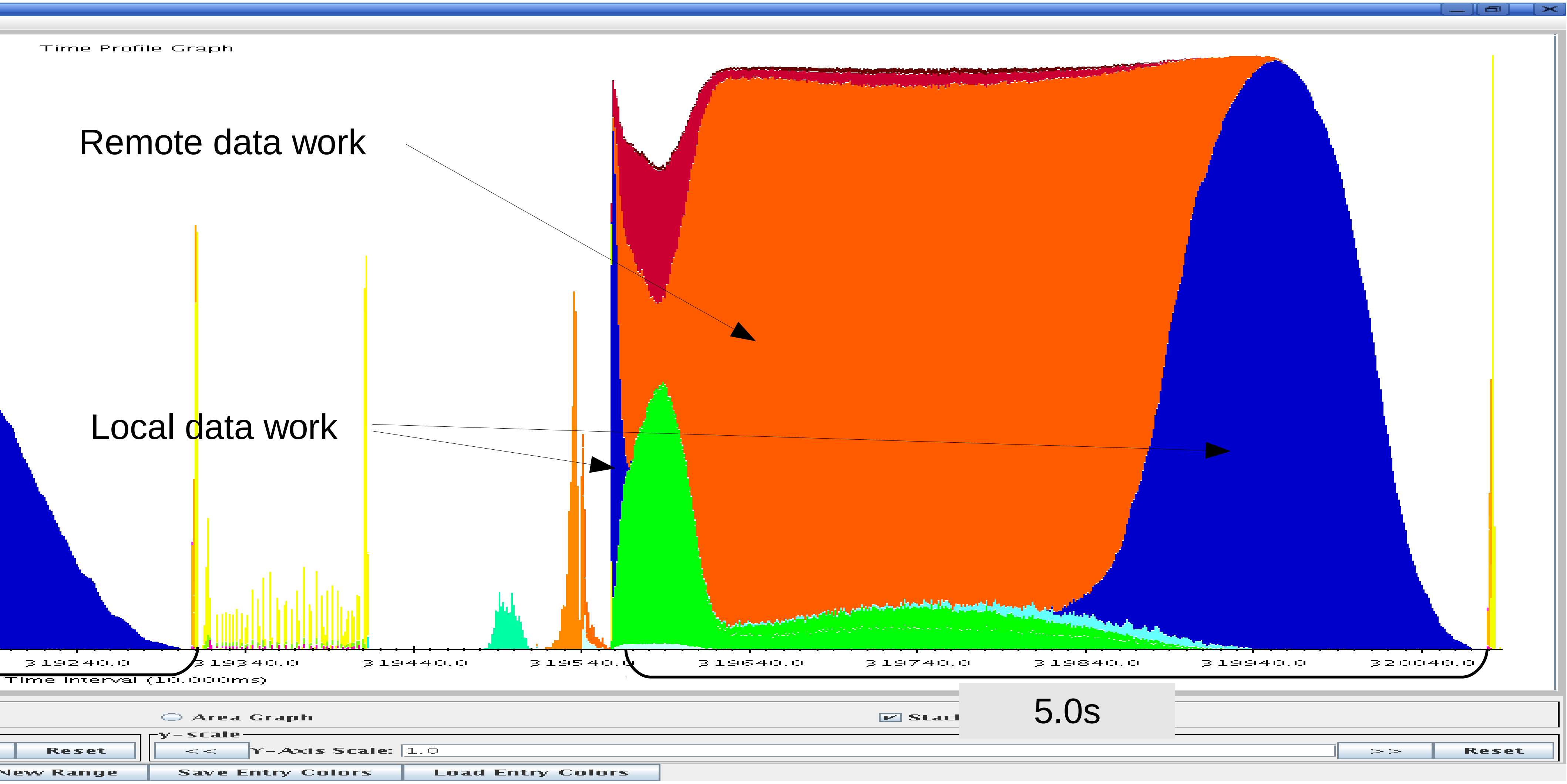
- Tree-based gravity solver
- High order multipole expansion
- Periodic boundaries (if needed)
- SPH: (Gasoline compatible)
- Individual multiple timesteps
- Dynamic load balancing with choice of strategies
- Checkpointing (via migration to disk)
- Visualization

Overall structure



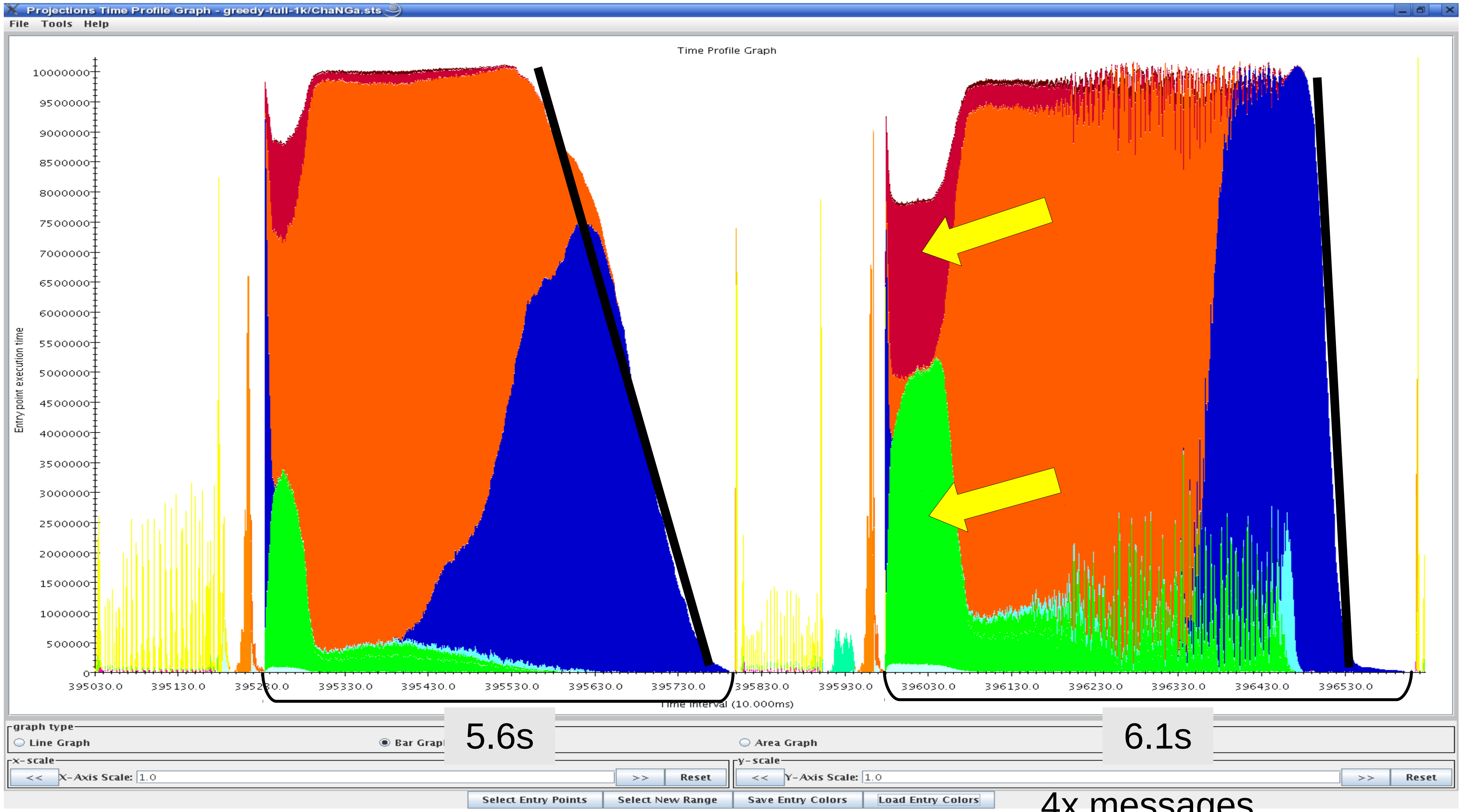
Remote/local latency hiding

Clustered data on 1,024 BlueGene/L processors



Load balancing with GreedyLB

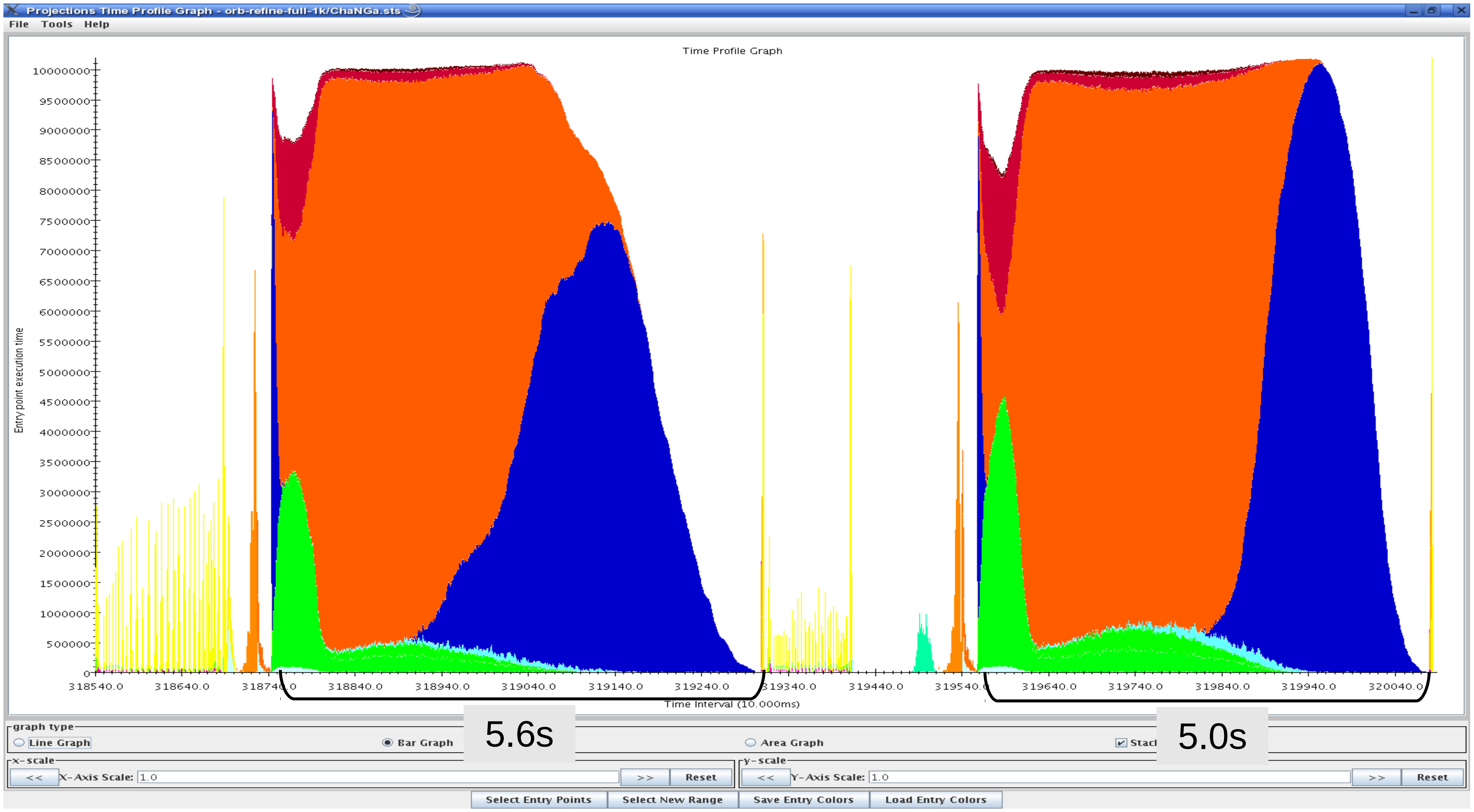
Zoom In 5M on 1,024 BlueGene/L processors



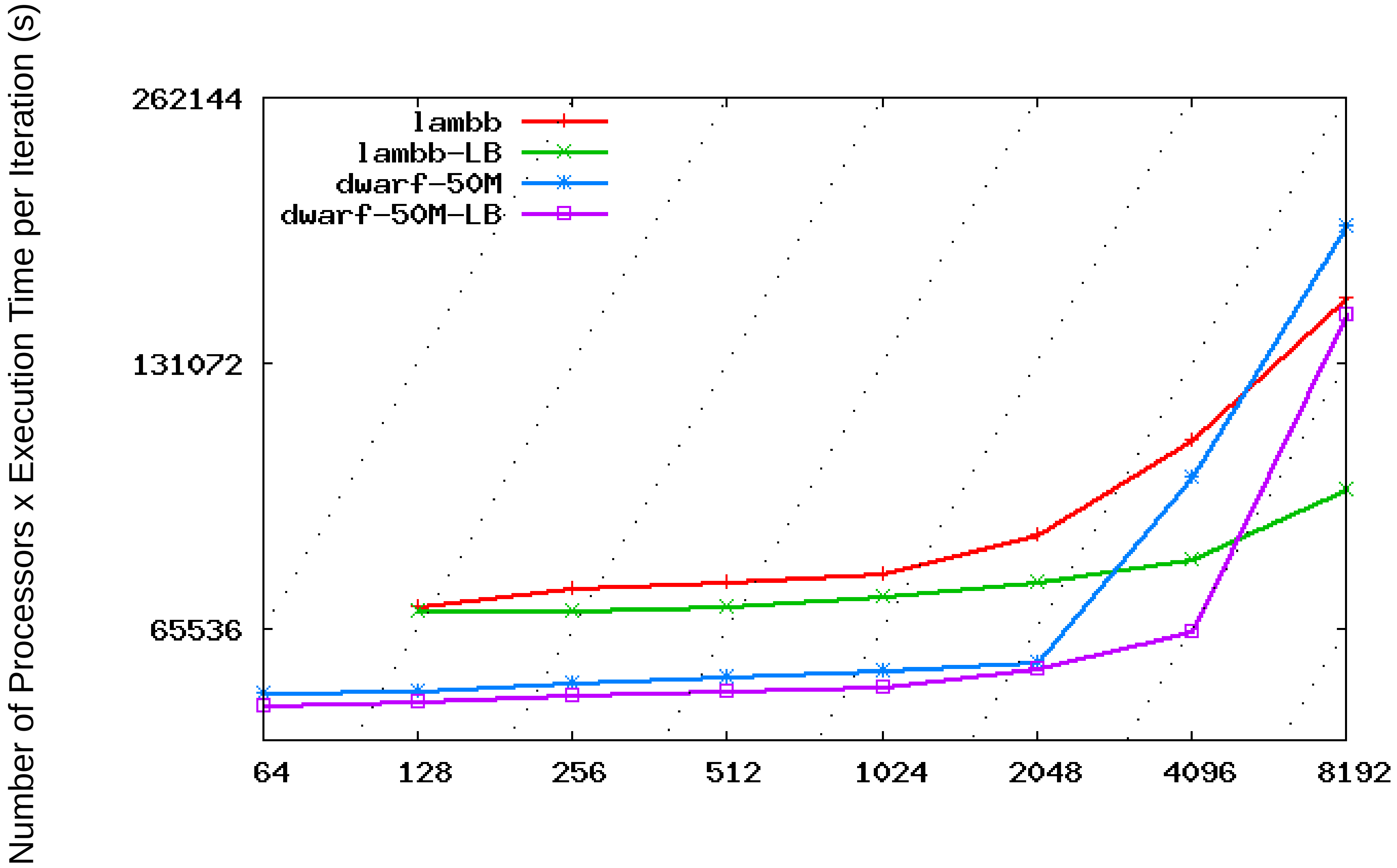
4x messages

Load balancing with OrbRefineLB

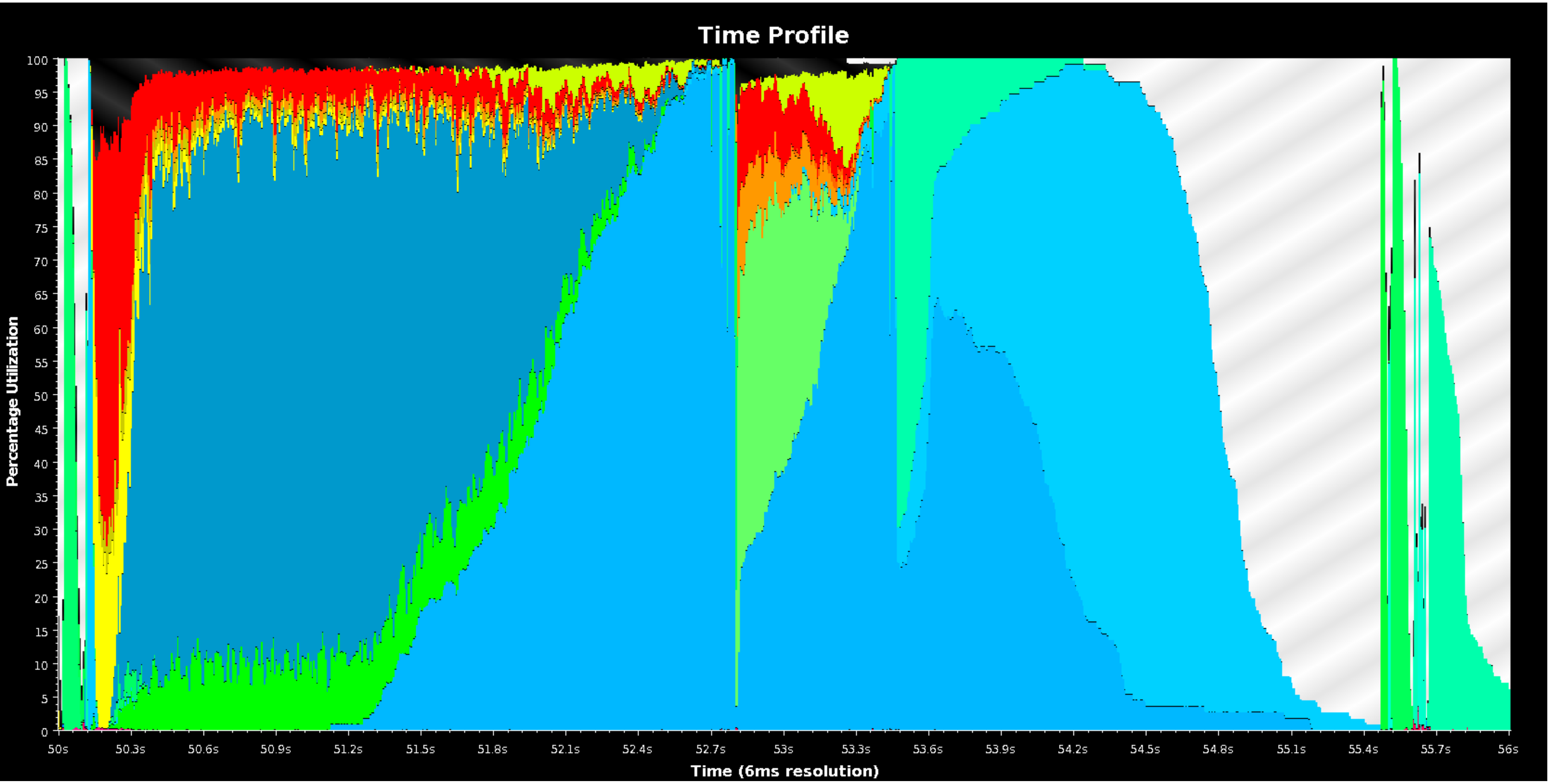
Zoom in 5M on 1,024 BlueGene/L processors



Scaling with load balancing



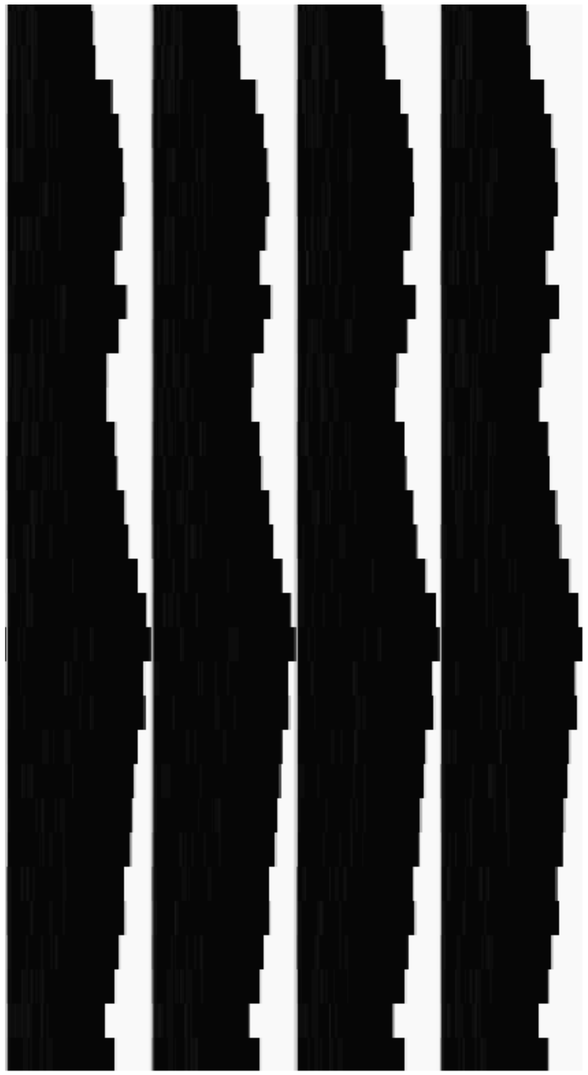
Overlap of Phases



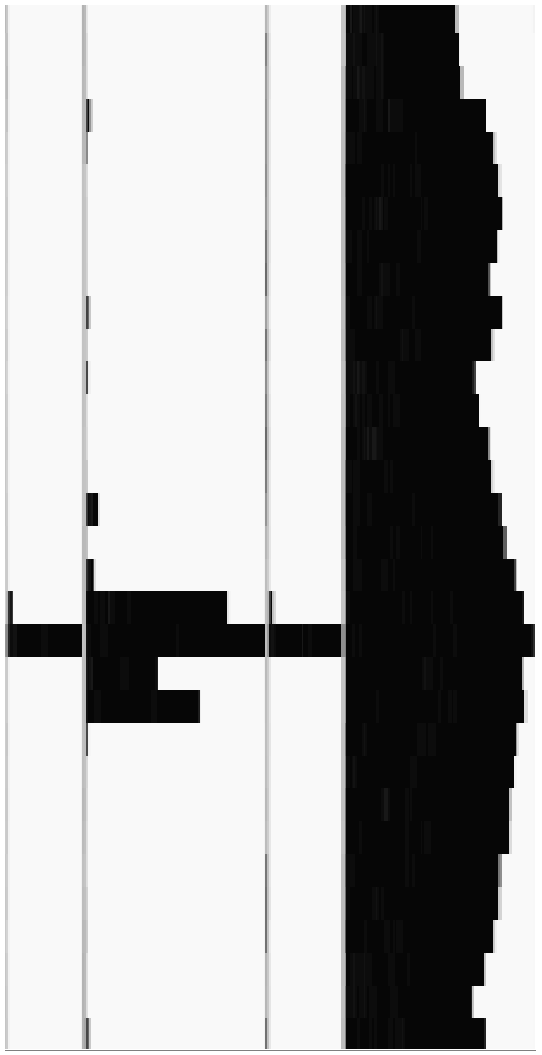
Cosmo Loadbalancer

- Use Charm++ measurement based load balancer
- Modification: provide LB database with information about timestepping.
  - “Large timestep”: balance based on previous Large step
  - “Small step” balance based on previous small step

Results on 3 rung example



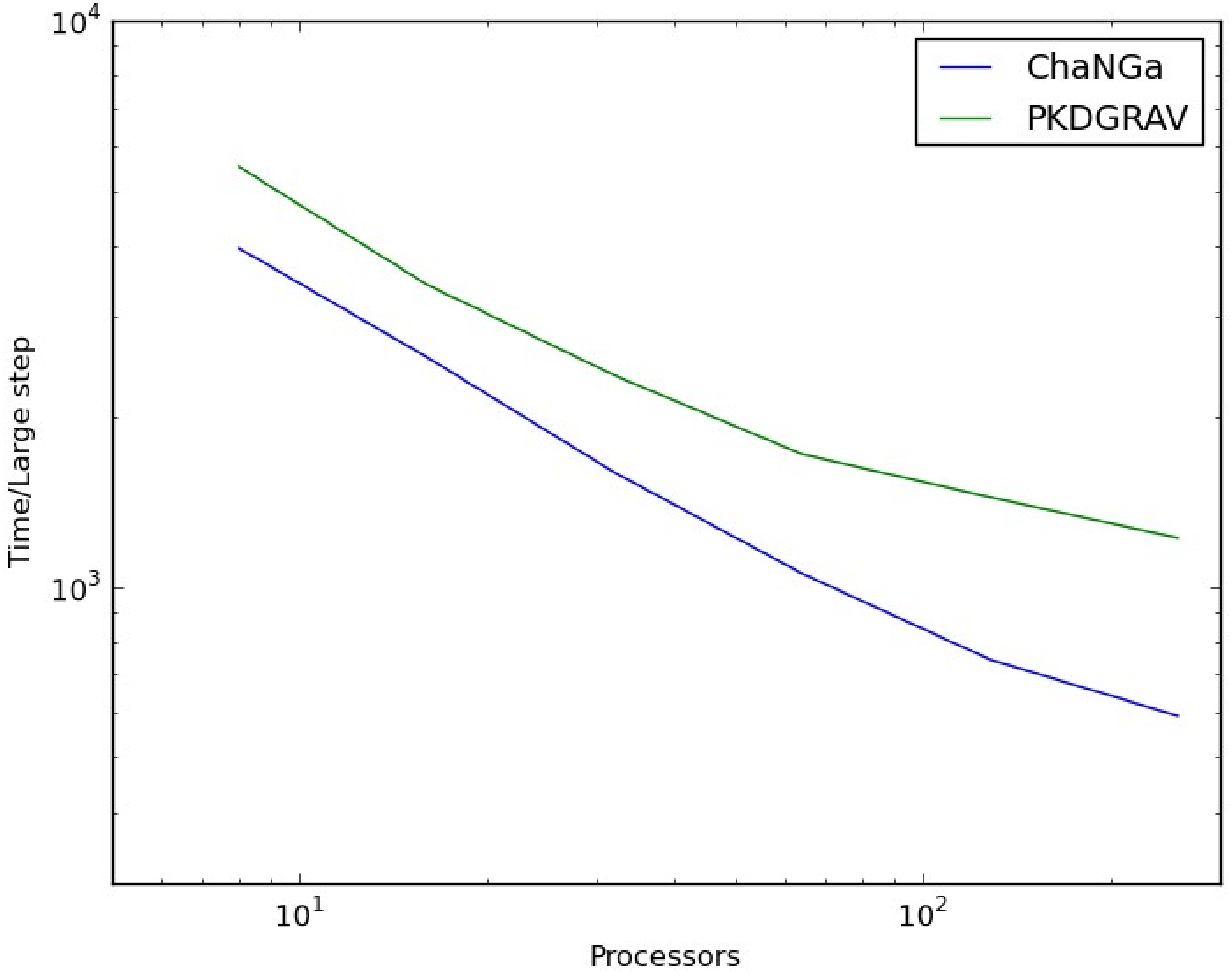
613s



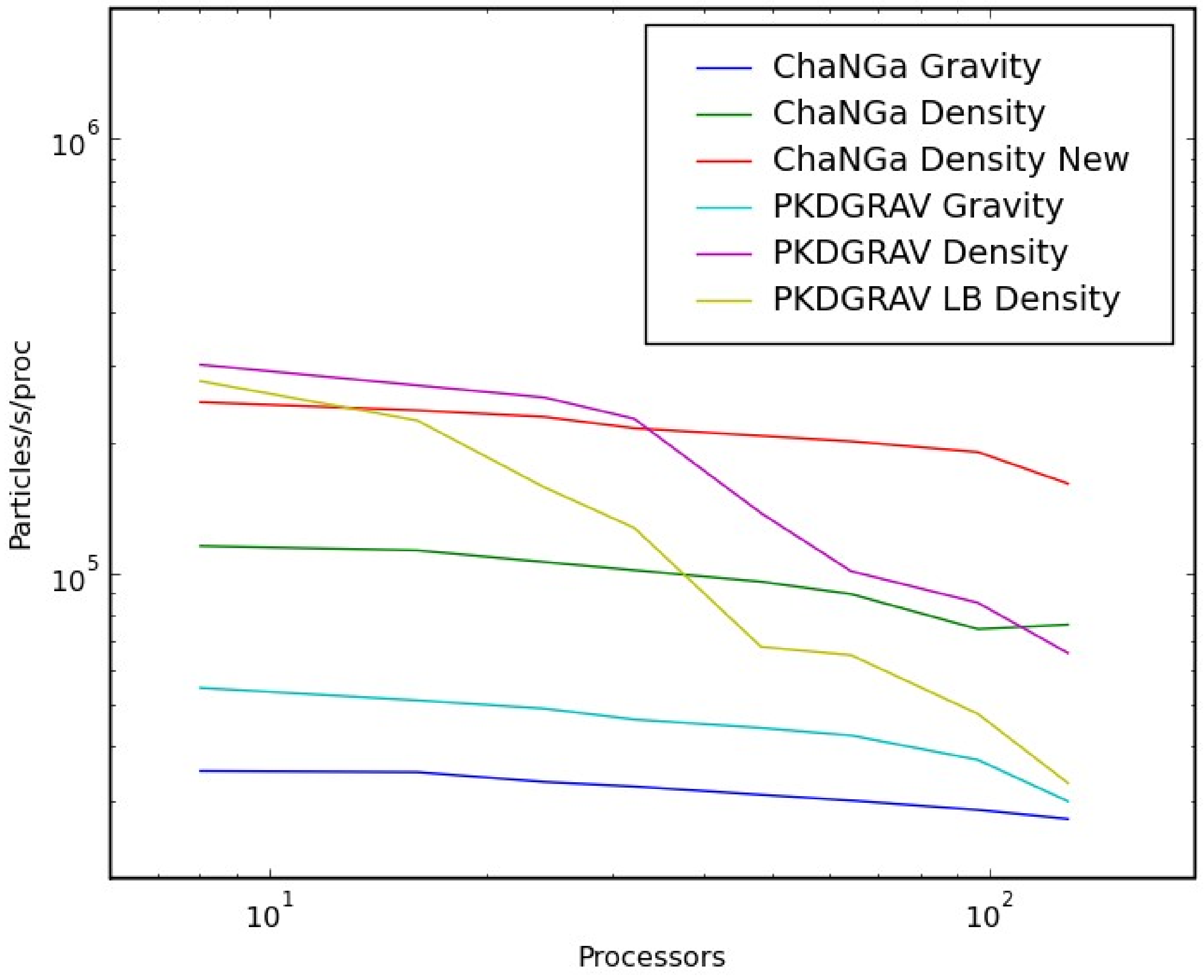
429s

228s

Multistep Scaling



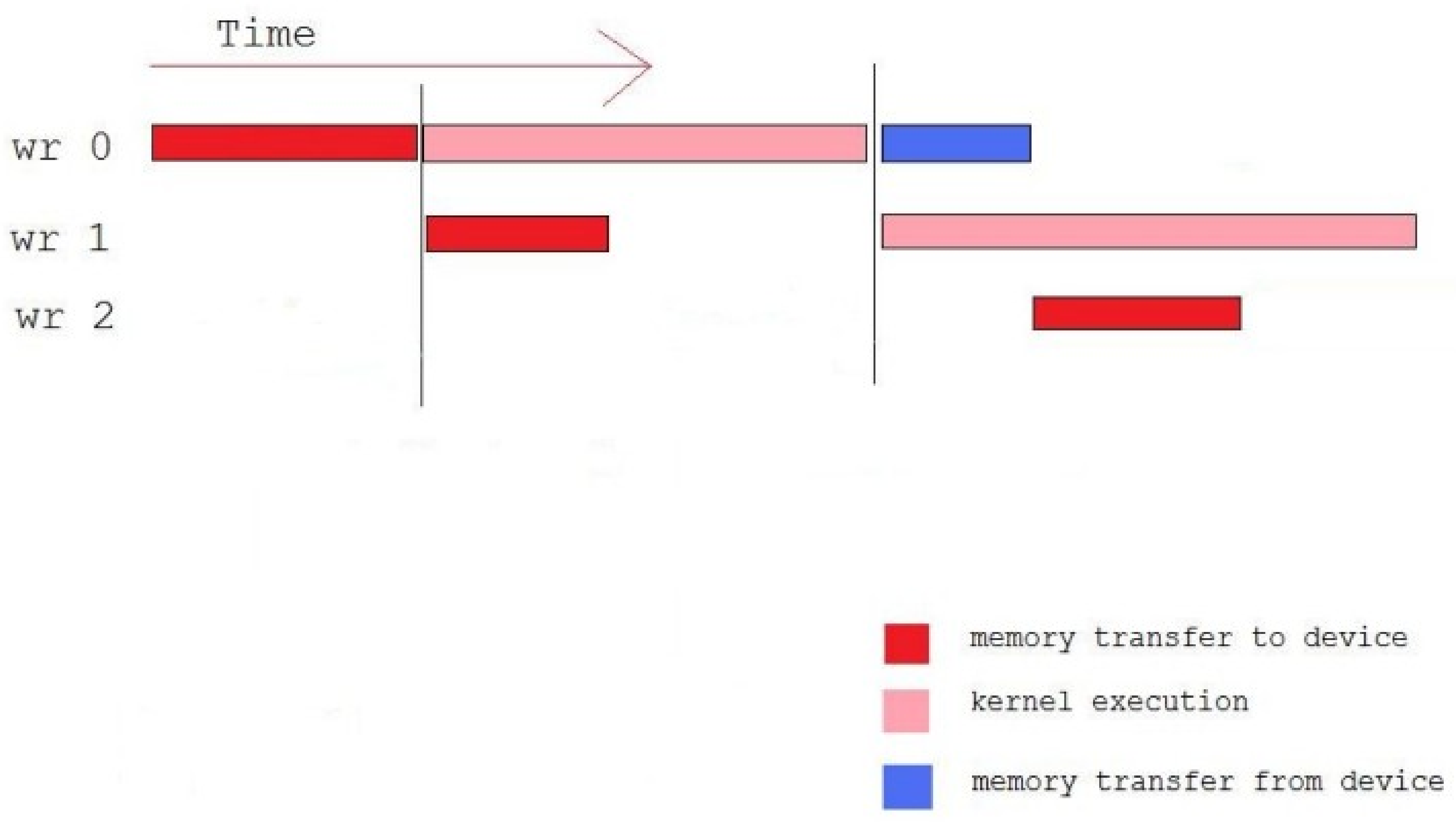
SPH Scaling


ChaNGa Gravity
ChaNGa Density
ChaNGa Density New
PKDGRAV Gravity
PKDGRAV Density
PKDGRAV LB Density
10^6
10^5
Particles/s/proc
10^1
10^2
Processors

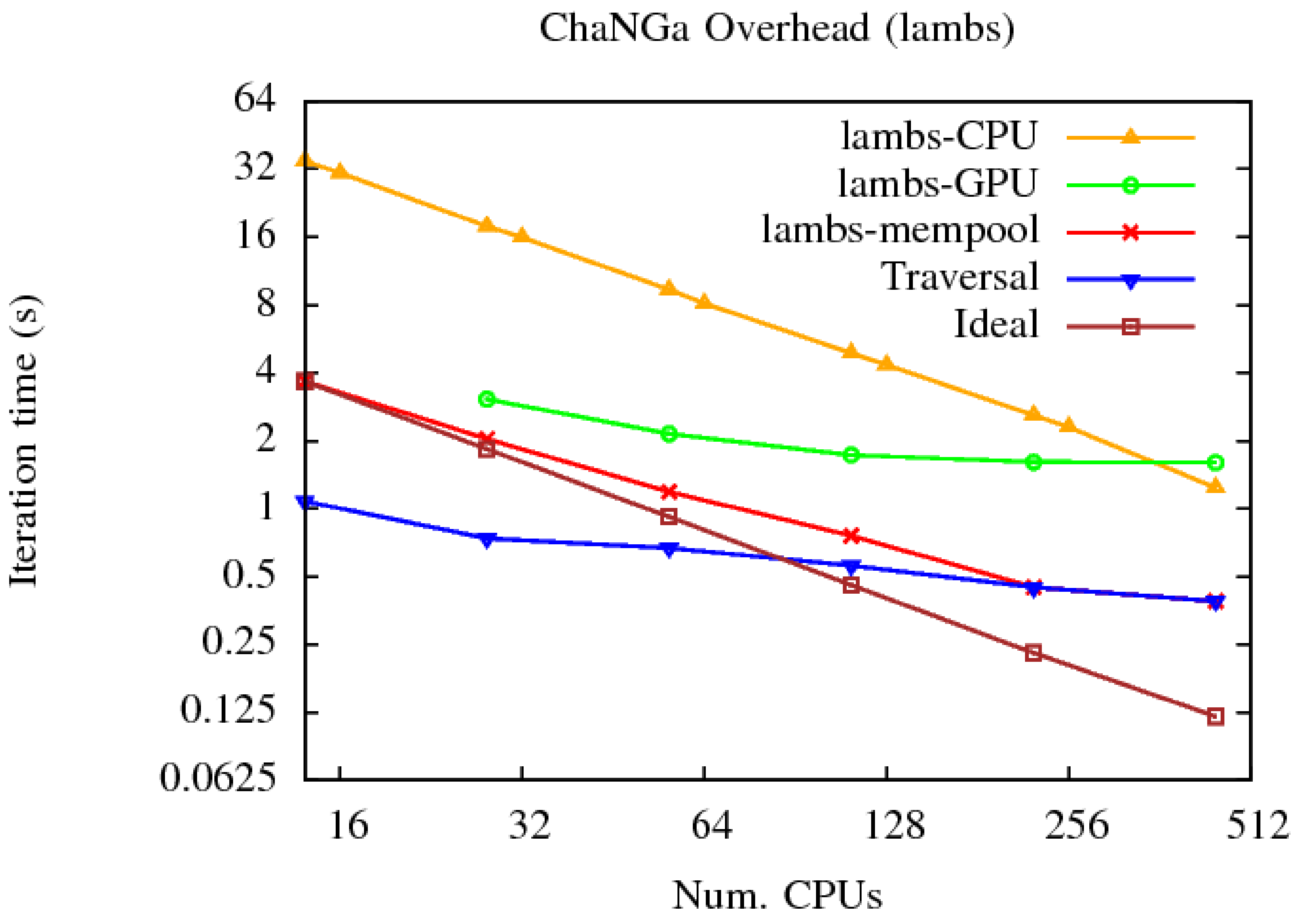
ChaNGa on GPU clusters

- Immense computational power
- Feeding the monster is a problem
- Charm++ GPU Manager
  - User submits work requests with callback
  - System transfers memory, executes, returns via callback
  - GPU operates asynchronously
  - Pipelined execution

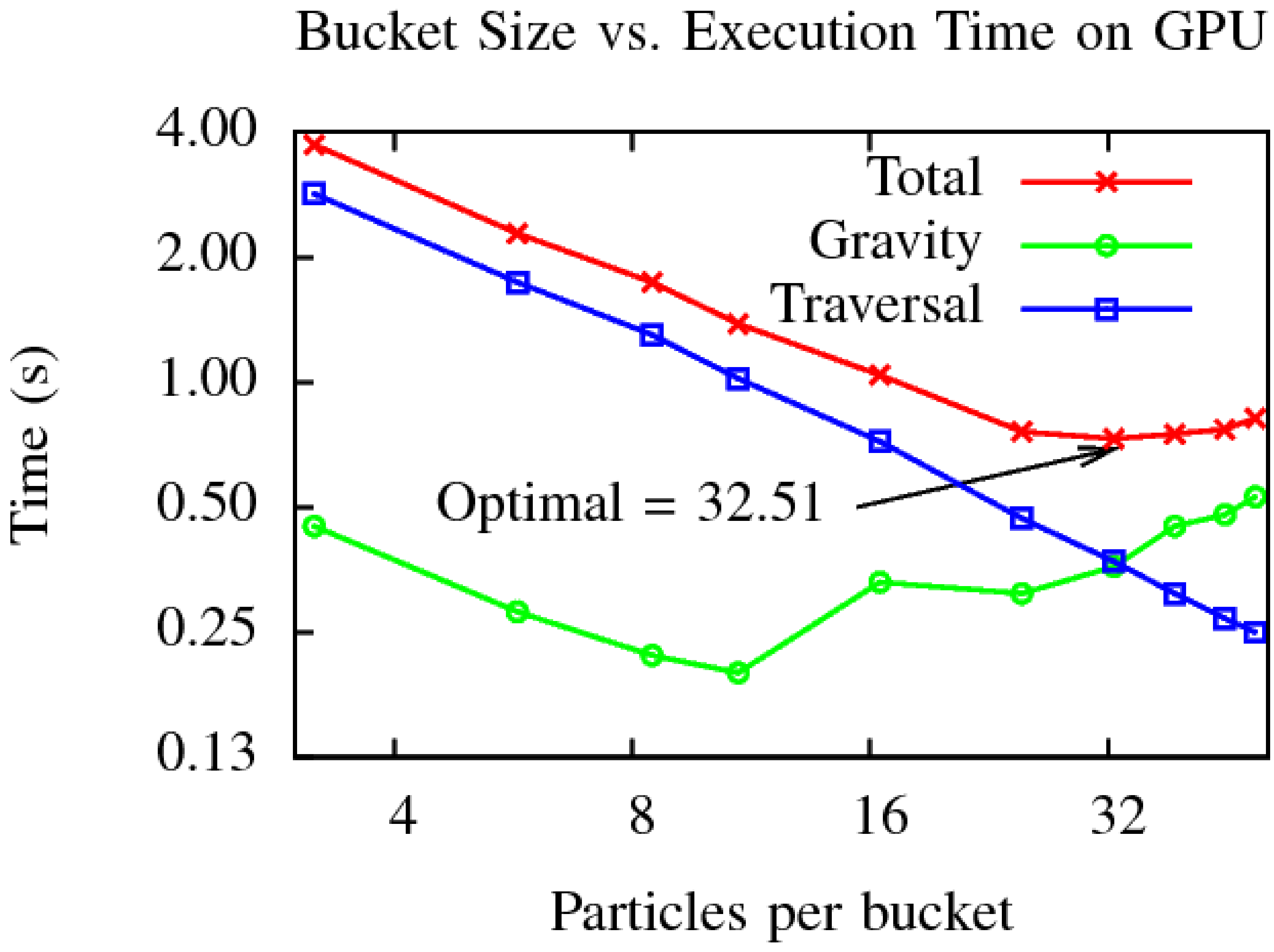
Execution of Work Requests



GPU Scaling

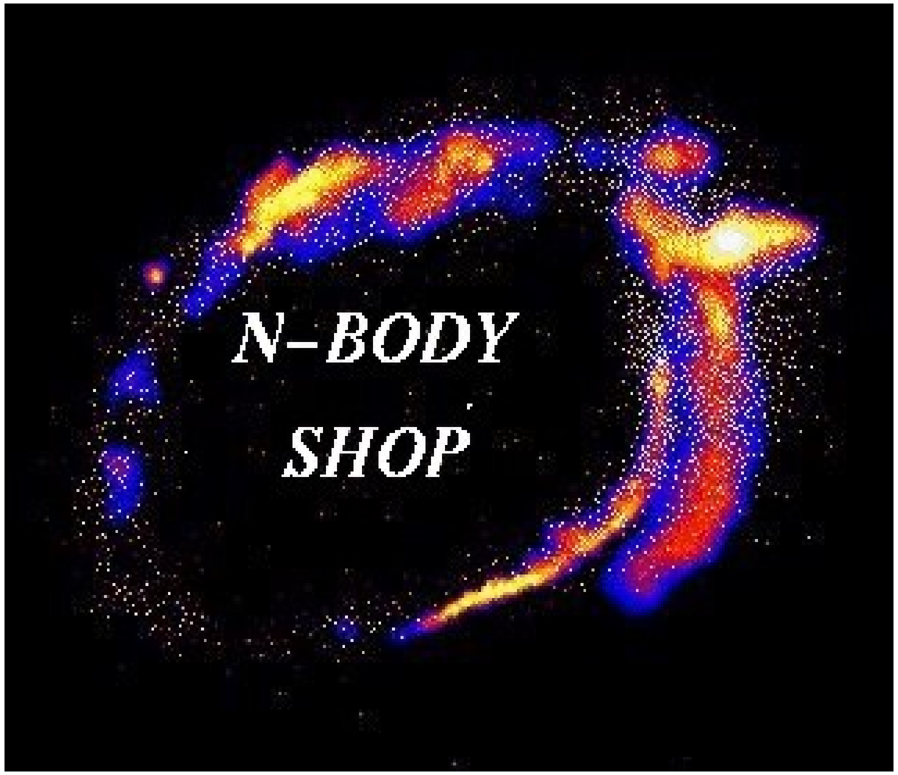

ChaNGa Overhead (lambs)
Iteration time (s)
Num. CPUs
64
32
16
8
4
2
1
0.5
0.25
0.125
0.0625
16
32
64
128
256
512
lambs-CPU
lambs-GPU
lambs-mempool
Traversal
Ideal

GPU optimization



Summary

- Successfully created highly scalable code in HLL
  - Computation/communication overlap
  - Object migration for LB and Checkpoints
  - Method prioritization
  - GPU Manager framework
- HLL not a silver bullet
  - Communication needs to be considered
  - “Productivity” unclear
    - Real Programmers write Fortran in any language



PARALLEL PROGRAMMING LAB
DEPT OF COMPUTER SCIENCE, UNIVERSITY OF ILLINOIS

Laxmikant Kale
Filippo Gioachin
Pritish Jetley
Celso Mendes
Amit Sharma
Lukasz Wesolowski
Edgar Solomonik

Thomas Quinn
Graeme Lufkin
Joachim Stadel
James Wadsley

Availability

- Charm++: http://charm.cs.uiuc.edu
- ChaNGa download: http://software.astro.washington.edu/nchilada/
- Release information: http://hpcc.astro.washington.edu/tools/changa.html
- Mailing list: changa-users@u.washington.edu