Future Programming Models

Stefano Markidis

2015 PDC SUMMER SCHOOL

Programming Models

- Historic Perspective on Programming Models
- The challenge for Programming Models at Exascale:
  - Synchronization, Delays and Imbalance
- Incremental vs Disruptive Programming Models for exascale:
  - Future of MPI (incremental)
  - The PGAS Approach (disruptive)
  - Task-based Approaches (disruptive)

Ongoing 3rd Software Crisis !

60s-70s

First Crisis

Assembler too hardware specific and too low level.

Need for abstraction and portability still retaining Assembler performance

SOLUTION Fortran, C

Ongoing 3rd Software Crisis !

60s-70s	70s-80s	> 2005
First Crisis	Second Crisis	
Assembler too hardware specific and too low level. Need for abstraction and portability still retaining Assembler performance	C and Fortran not convenient for building and maintaining code with million lines of code developed by several developers.	
SOLUTION	Fortran, C	C++, Java, Python

Ongoing 3rd Software Crisis !

	First Crisis	Second Crisis	Third Crisis
	Assembler too hardware specific and too low level. Need for abstraction and portability still retaining Assembler performance	C and Fortran not convenient for building and maintaining code with million lines of code developed by several developers.	What is the best programming approach for exploiting effectively parallelism In the processor, cluster, accelerators ?
SOLUTION	Fortran, C	C++, Java, Python	? MPI + X, PGAS, Tasks

What is it next in Programming Models?

- HPC programming models are tailored to a given (expensive) hardware/architecture. Think HPC systems as the Ferrari of the computers. Need skilled pilot, not for everybody!
- How is going to be the next programming system → How is going to be the next hardware ?

Now: Top500 #1 – Thianhe-2

- 3,120,000 cores.
- Using Intel Xeon Phi **accelerators**
- Theoretical Peak Performance: 54.9 PFlops.
- Maximum performance obtained with highly optimized compute-intense code (Linpack) is 33 PFlops → **Efficiency is 60%**!
- 17 MW

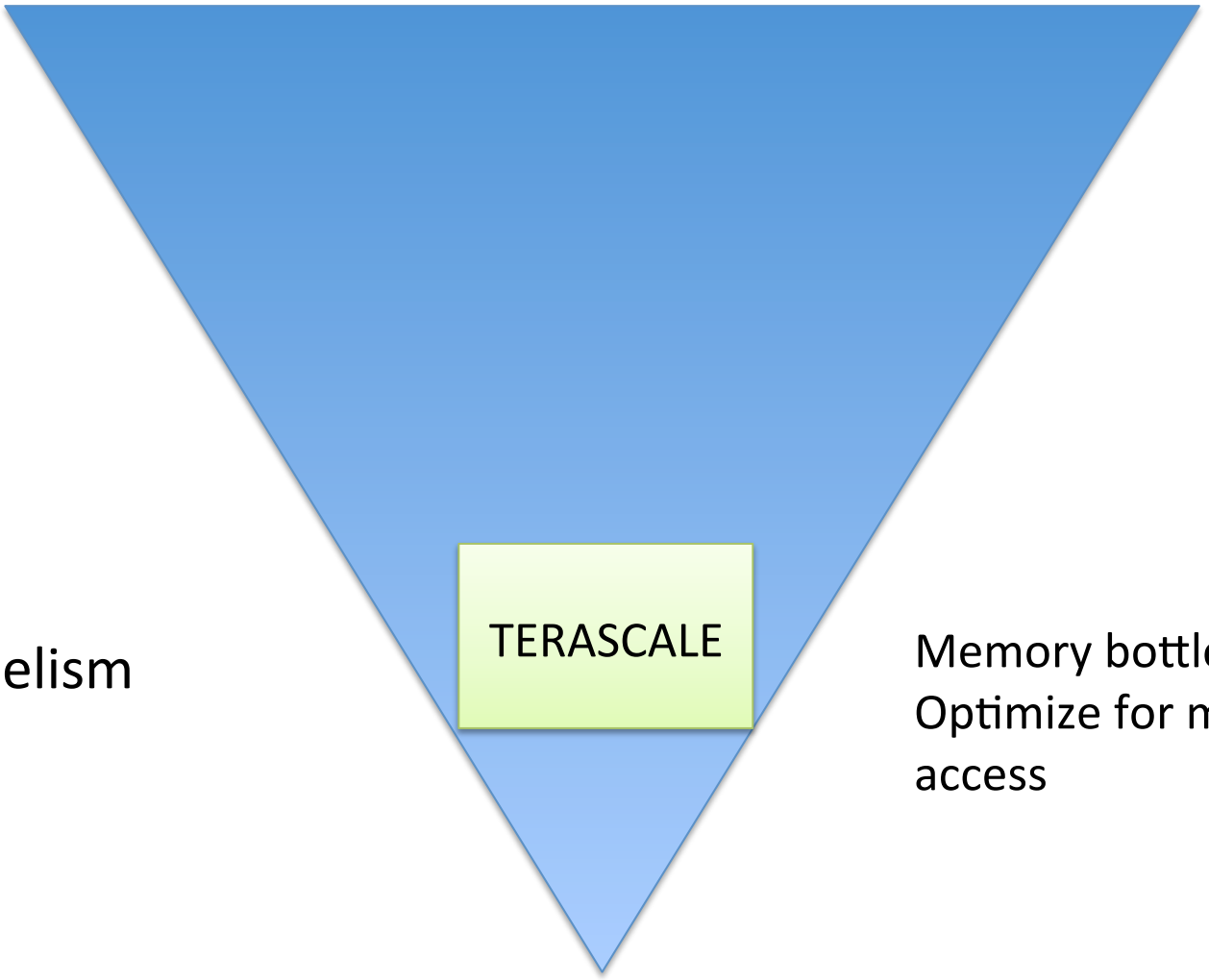
Next: Aurora at ANL (Coral DOE Initiative)

- > 180 PFlops (Peak Performance) in 2018.
- Intel and Cray
- Processor: Intel Xeon Phi Gen 3 (**accelerator**)
- 13 MW

Next next: Exaflop Machine

- U.S President Obama wants to bring back the glory days of 2012. China's Tianhe-2 became the fastest supercomputer in the world in 2013
- On July 29th President Obama issued an Executive Order to build the first exascale supercomputer by **2024**.
- U.S blocked exports of Intel Chips to Thianhe-2. China decided to go for Thianhe-2A upgrade with new chip based on DSP (accelerator).
- **Race to Exascale!**

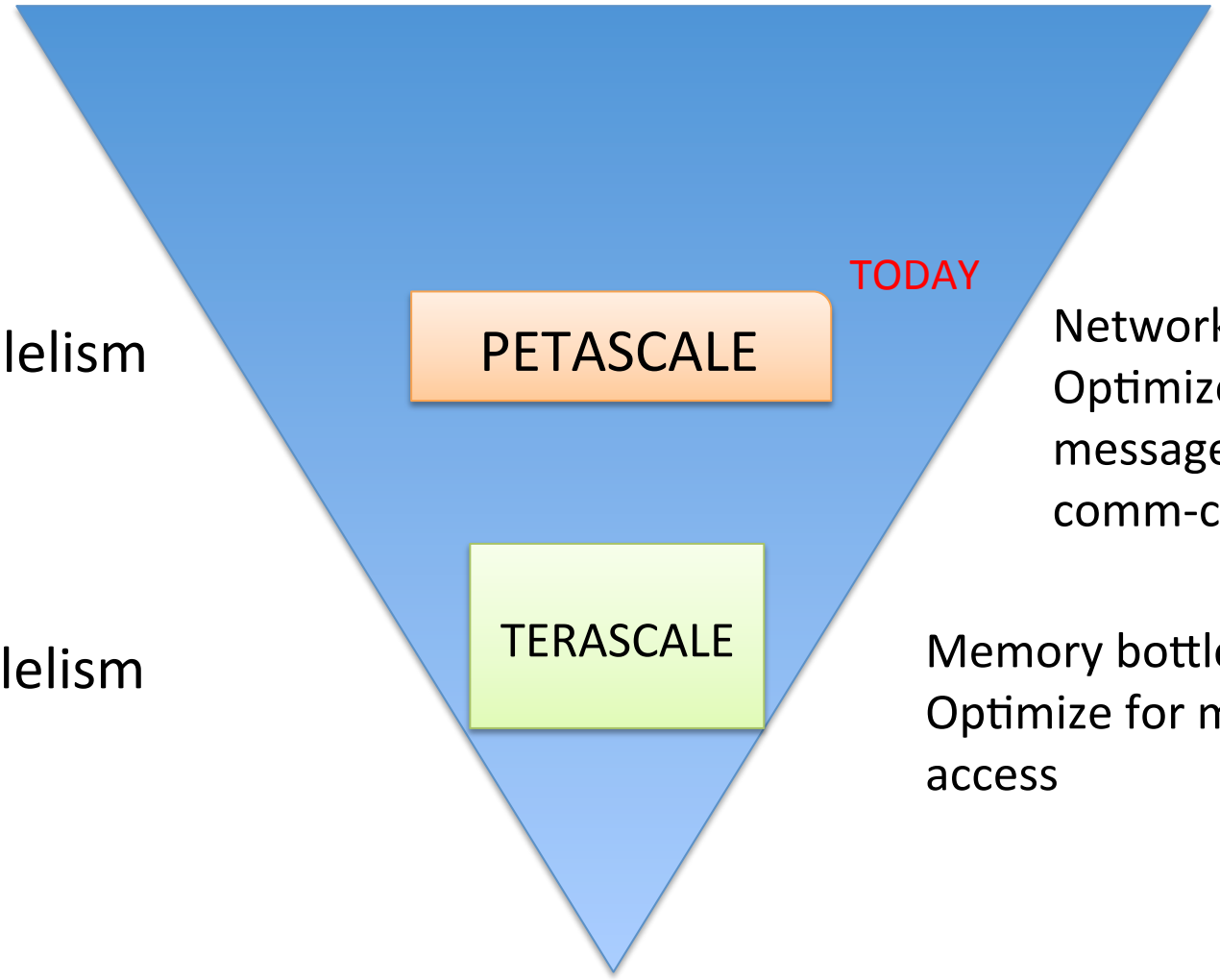
Going to Exascale: Software Challenges



10^2 parallelism

Memory bottleneck:
Optimize for memory access

Going to Exascale: Software Challenges



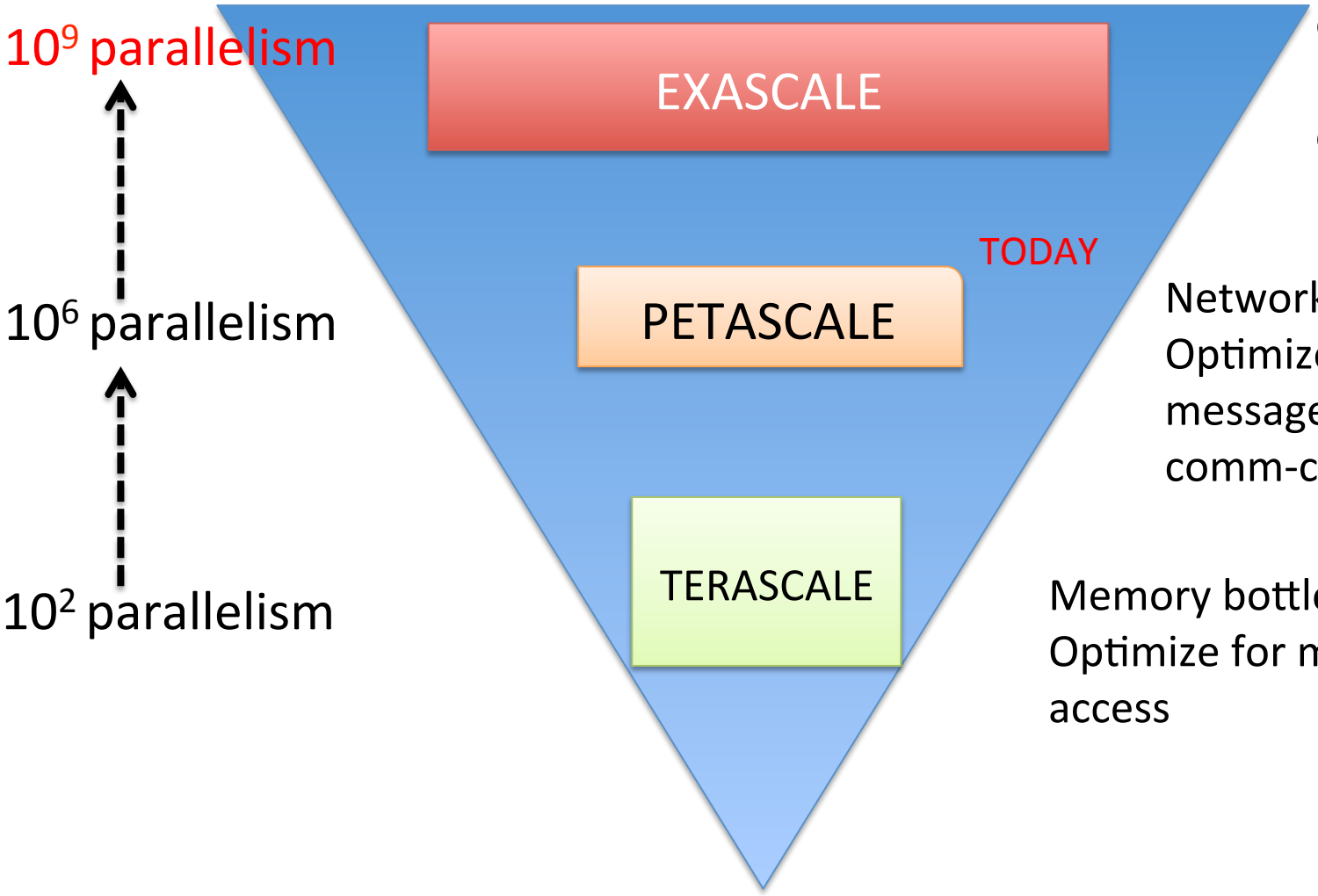
10^6 parallelism

10^2 parallelism

Network bottleneck:
Optimize for few large messages and overlap comm-comp.

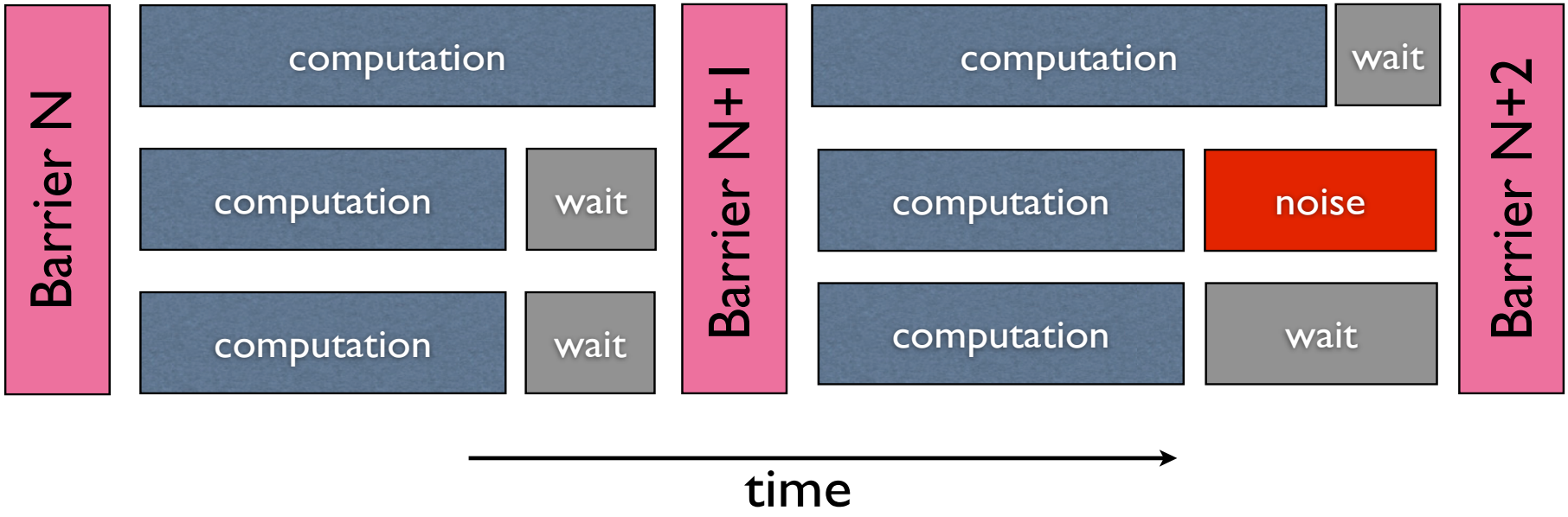
Memory bottleneck:
Optimize for memory access

Going to Exascale: Software Challenges

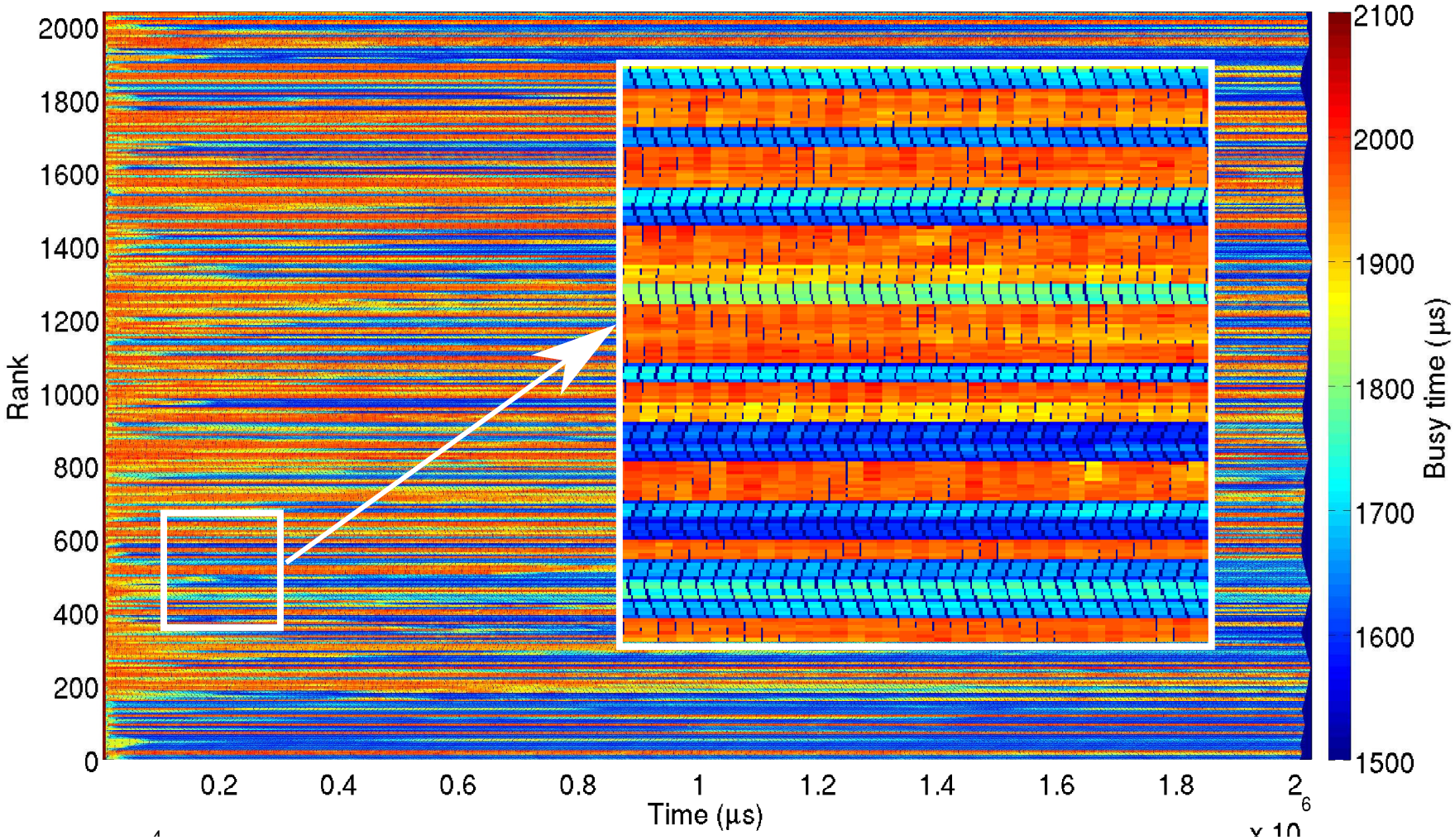


Common HPC application

- Iterative computation phase + synchronization (fast processes waiting for slow phase).



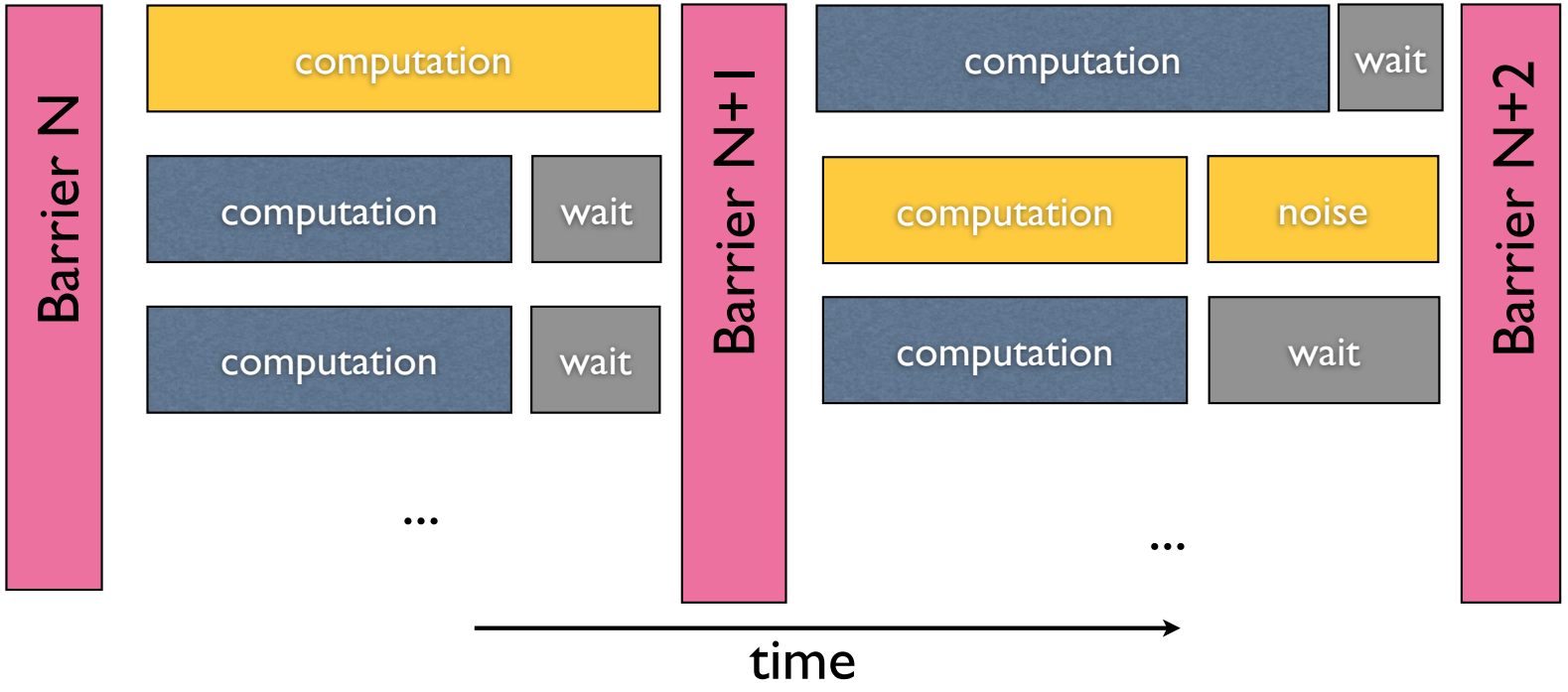
Always Some Delay !



Different Computation Time per process on 2,046 processes doing the same thing!

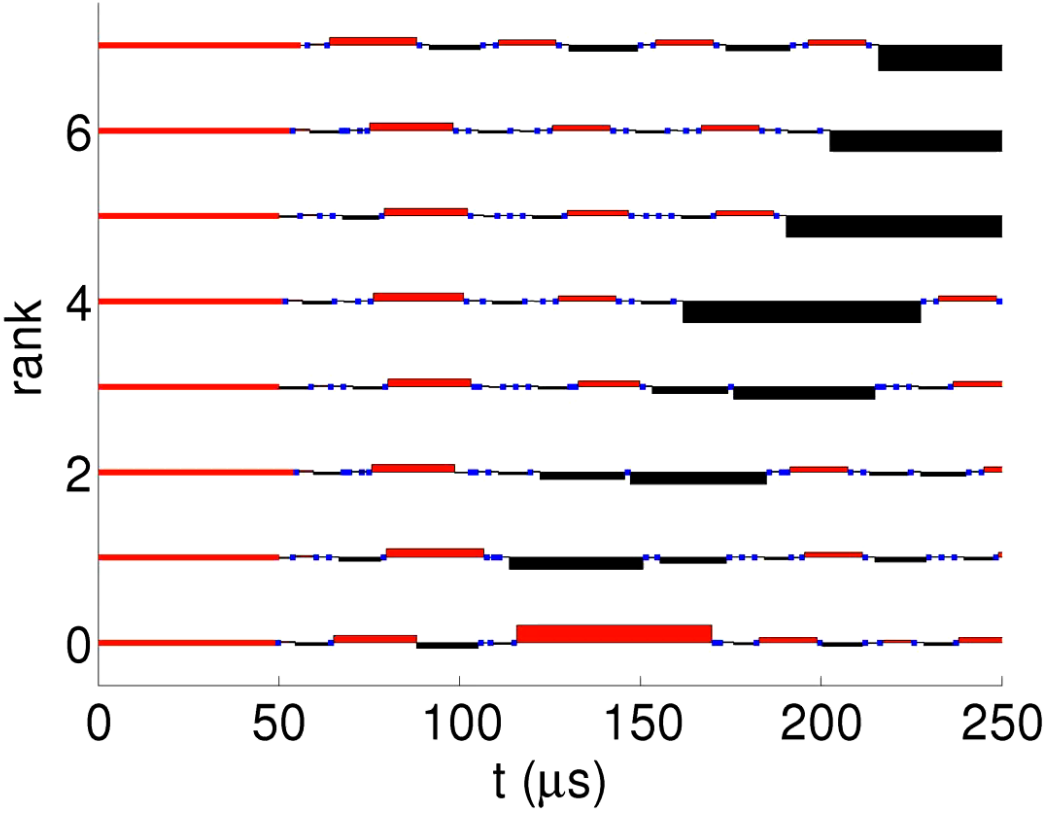
Delay Impacts Global Synchronization

Each computation is as fast as the slowest process.



Local Synchronization

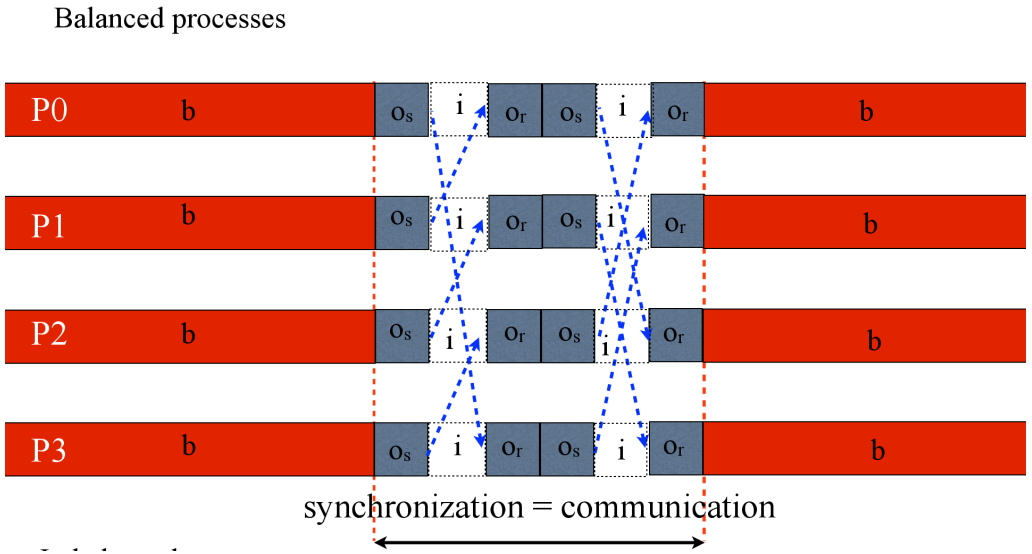
Idle propagates to other processes via neighbor communication.



Markidis, Stefano, et al. "Idle waves in high-performance computing." Physical Review E 91.1 (2015)

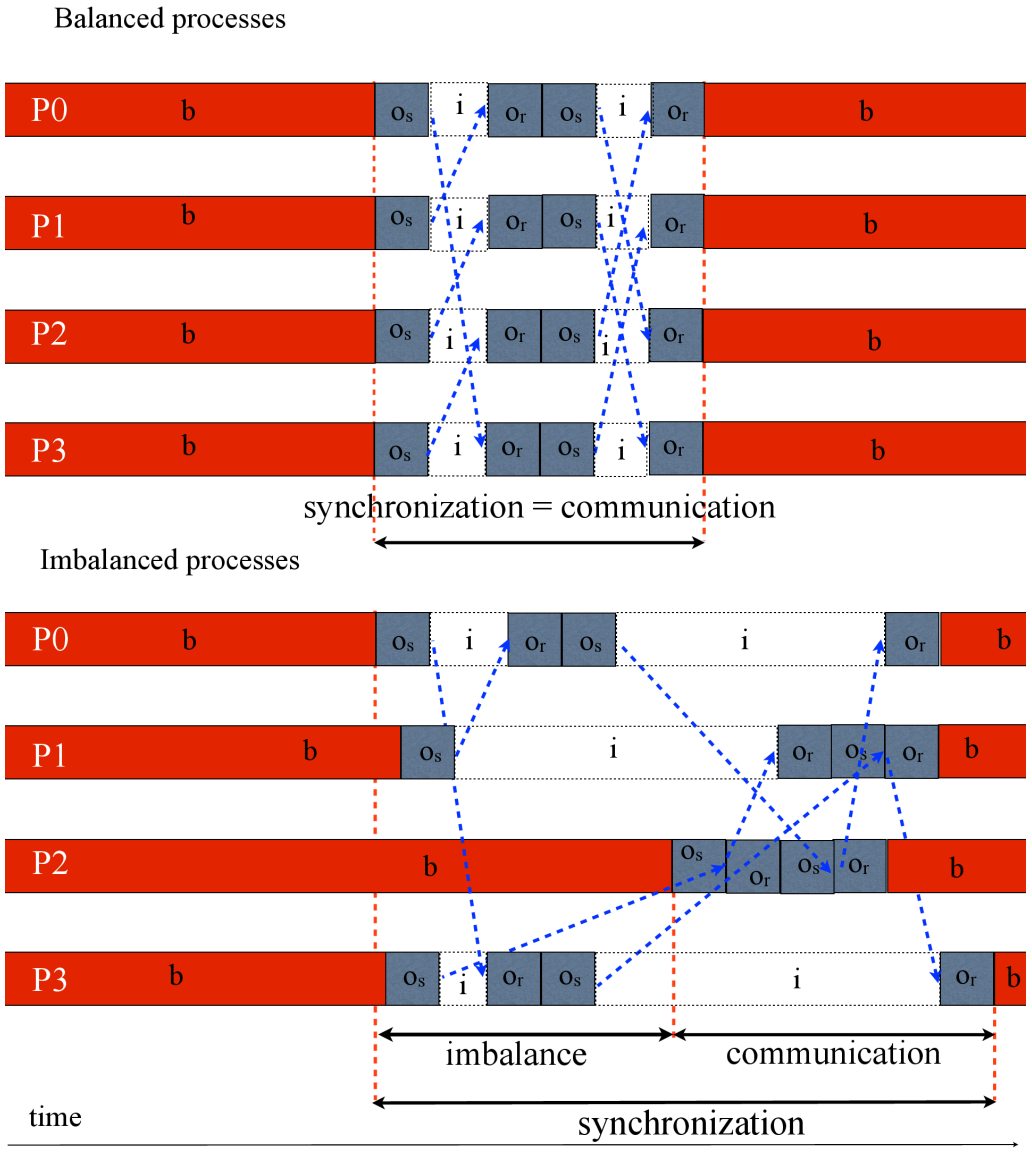
Idle Propagating in Collectives

Collectives are a series of point-to-point communication (look at implementation!).



Idle Propagating in Collectives

REALITY



Ivy Bo Peng et al., "The Cost of Synchronizing Imbalanced Processes in Message Passing Systems", CLUSTER 2015 Proceedings.

Programming Models for Exascale

- Programming models that can adapt in time, spawning new processes and react to imbalance:
  - **MPI** Dynamic Process Management (MPI_Comm_spawn)
  - Smart scheduler that spawns **tasks** depending on the need

Programming Models for Exascale

- Programming models that can adapt in time, spawning new processes and react to imbalance:
  - **MPI** Dynamic Process Management (MPI_Comm_spawn)
  - Smart scheduler that spawns **tasks** depending on the need
- Use (collective) algorithms that can hide process imbalance (make slowest process to communicate last or overlap communication with communication):
  - Non-blocking collective in **MPI**3.0

Programming Models for Exascale

- Programming models that can adapt in time, spawning new processes and react to imbalance:
  - **MPI** Dynamic Process Management (MPI_Comm_spawn)
  - Smart scheduler that spawns **tasks** depending on the need
- Use (collective) algorithms that can hide process imbalance (make slowest process to communicate last or overlap communication with communication):
  - Non-blocking collective in **MPI**3.0
- Make application developer think about synchronization in the same way MPI makes you think about data locality.
  - **PGAS** approach (implicit comm and explicit synch)
  - **Task-based** approaches force you think about dependencies

Going to Exascale: Incremental vs Disruptive

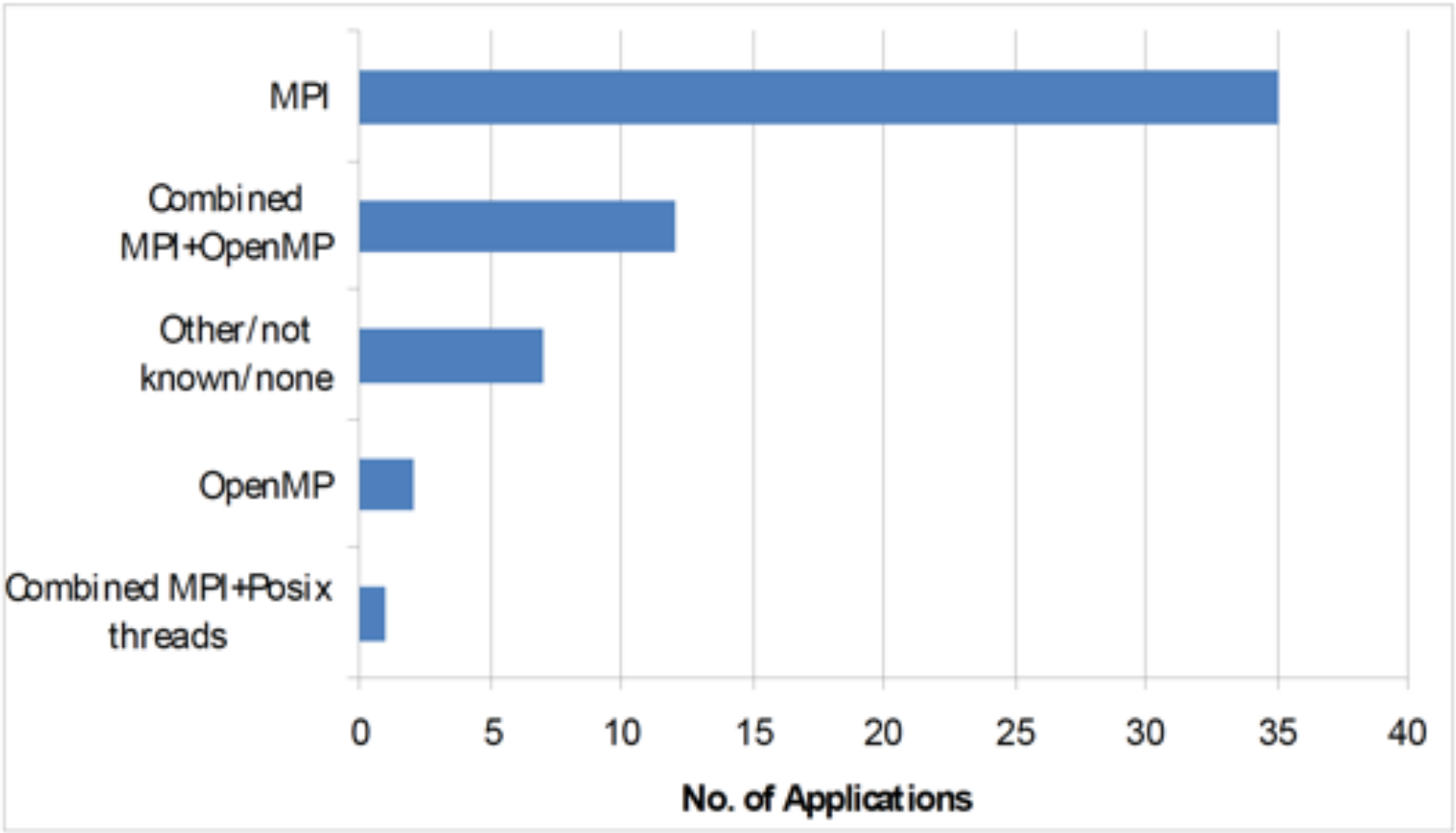
- Incremental
  - improve traditional approaches, i.e. **MPI**. Focus on scalability in execution time (think about collective algorithms) and memory consumption (think about zero-copy algorithms).
  - Add to existing code new programming systems, MPI + PGAS. Interoperability becomes central issue.

Going to Exascale: Incremental vs Disruptive

- Incremental
  - improve traditional approaches, i.e. **MPI**. Focus on scalability in execution time (think about collective algorithms) and memory consumption (think about zero-copy algorithms).
  - Add to existing code new programming systems, MPI + PGAS. Interoperability becomes central issue.
- Disruptive → 1) rethink your algorithms 2) rewrite large part of your code from scratch:
  - **PGAS programming model**
  - **Task-Based Approaches**

Ubiquity of MPI

Almost all the codes running on supercomputer use MPI. OpenMP is used in combination with MPI.



From a survey of 57 applications running on PRACE machines

New features in MPI-3

- New version MPI3 in Sept 2012 (not very active, MPI-2 in 1996):
- Major improvements:
  - Collectives (Non-blocking, Neighborhood)
  - One-sided Communication

Why we care about collectives so much ?

- Top systems have tens of thousands of nodes and order 1 million cores:
- – Tianhe-2: 16,000 nodes
  – Sequoia (LLNL) 98,304 nodes, >1M cores
- Just getting all of these nodes to agree takes time **O(10us) or about 10,000 cycles of time!**

Non-blocking Collective (New Feature in MPI3.0)

Start a broadcast of 100 ints from process 0 to every process in the group, perform some computation on independent data (array2)

```
MPI_Comm comm;
int array1[100], array2[100];
int root=0;
MPI_Request req;
...
MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req)
compute(array2, 100);
```

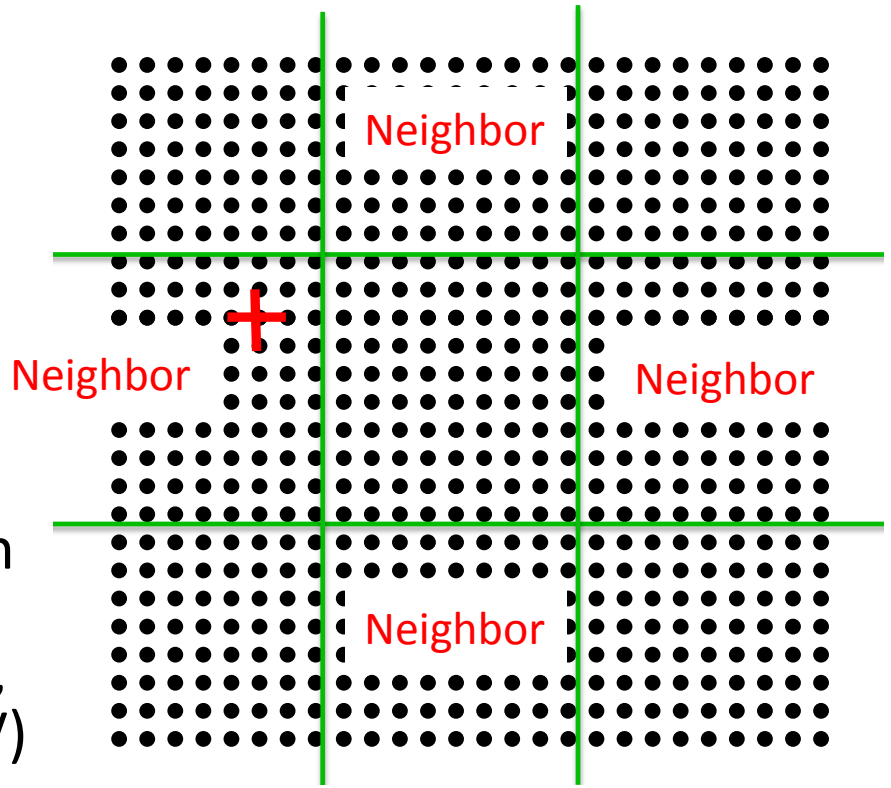
Non-blocking Collective (New Feature in MPI3.0)

Start a broadcast of 100 ints from process 0 to every process in the group, perform some computation on independent data (array2), and then complete the outstanding broadcast operation.

```
MPI_Comm comm;
int array1[100], array2[100];
int root=0;
MPI_Request req;
...
MPI_Ibcast(array1, 100, MPI_INT, root, comm, &req)
compute(array2, 100);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Neigborhood Collective

- Collective operations that are defined only process neighborhoods. Process neighborhoods are attached to the communicator on which the collectives are called.
- 2 different neighbor collectives:
  - Gather (both blocking and non blocking) (MPI_NEIGHBOR_ALLGATHER, MPI_NEIGHBOR_ALLGATHERV)
  - **All-to-all** (MPI_NEIGHBOR_ALLTOALL)



Neighborhood Communication

```
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
INTEGER ierr
sndbuf(1:100,1) = u(  1,1:100)
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100,  1)
sndbuf(1:100,4) = u(1:100,100)



  DO i=1,num_neigh
    CALL MPI_IRECV(rcvbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i-1),&
                   ierr)
    CALL MPI_ISEND(sndbuf(1,i),100,MPI_REAL,neigh_rank(i),...,rq(2*i  ),&
                   ierr)
  END DO
  CALL MPI_WAITALL (2*num_neigh, rq, statuses, ierr)

u(  0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
u(1:100,  0) = rcvbuf(1:100,3)
u(1:100,101) = rcvbuf(1:100,4)
END
```

Use point to point comm

Neighborhood Communication

```
SUBROUTINE exchange (u, comm_cart, neigh_rank, num_neigh)
REAL u(0:101,0:101)
INTEGER comm_cart, num_neigh, neigh_rank(num_neigh)
REAL sndbuf(100,num_neigh), rcvbuf(100,num_neigh)
INTEGER ierr
sndbuf(1:100,1) = u(  1,1:100)
sndbuf(1:100,2) = u(100,1:100)
sndbuf(1:100,3) = u(1:100,  1)
sndbuf(1:100,4) = u(1:100,100)
CALL MPI_NEIGHBOR_ALLTOALL (sndbuf, 100, MPI_REAL, rcvbuf, 100, MPI_REAL, &
                            comm_cart, ierr)
```

Just 1 Collective !

```
u(  0,1:100) = rcvbuf(1:100,1)
u(101,1:100) = rcvbuf(1:100,2)
u(1:100,  0) = rcvbuf(1:100,3)
u(1:100,101) = rcvbuf(1:100,4)
END
```

Why we care about One-Sided ?

- Many machines (like Beskow) have hardware support on the network card for one-sided communication (= faster than point-to-point communication).
- Programming style that make you think about synchronization!

New MPI One-sided Interface

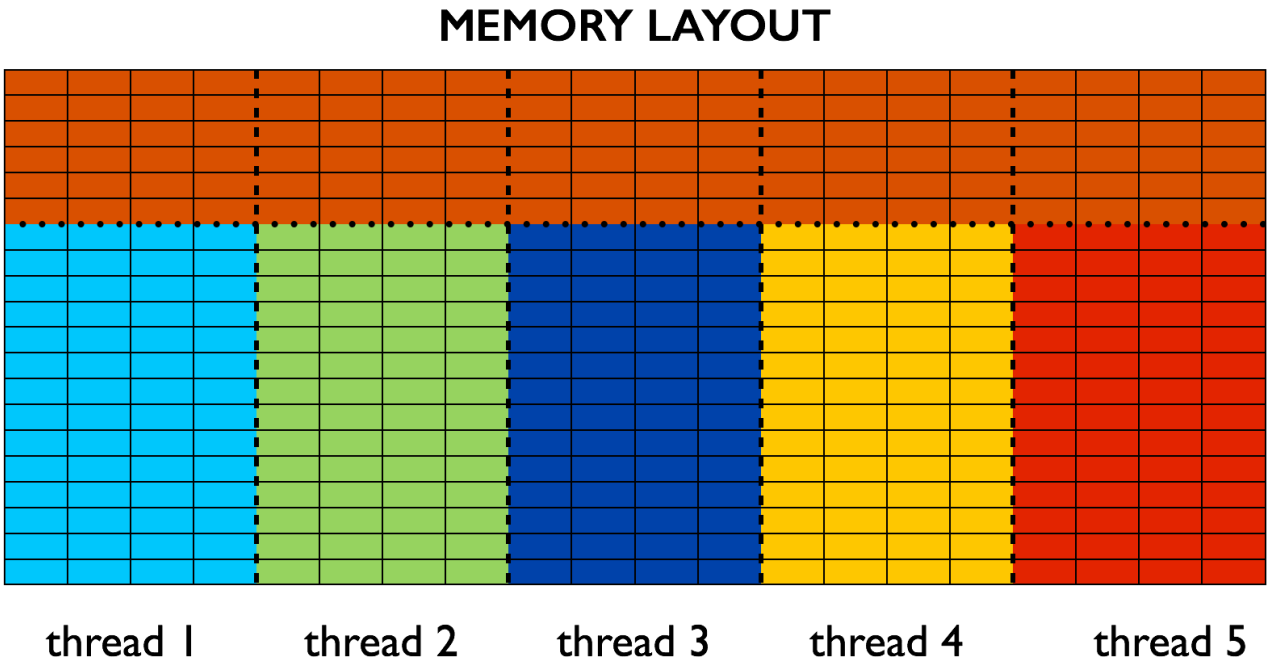
- MPI_Win_create_dynamic
  - Window without memory attached
  - MPI_Win_attach to attach memory to a window
- MPI_Win_allocate_shared
  - Windows with shared memory
  - Allows direct loads/store accesses by remote processes
- MPI_Rput, MPI_Rget, MPI_Raccumulate
  - Local completion by using MPI_Wait on request objects
- MPI_Get_accumulate, MPI_Fetch_and_op
  - Accumulate into target memory, return old data to origin
- MPI_Compare_and_swap
  - Atomic compare and swap
- MPI_Win_lock_all
  - Faster way to lock all members of win
- MPI_Win_flush / MPI_Win_flush_all
  - Flush all RMA ops to target / window
- MPI_Win_flush_local / MPI_Win_flush_local_all
  - Locally complete RMA ops to target / window
- MPI_Win_sync
  - Synchronize public and private copies of window
- Overlapping accesses were "erroneous" in MPI-2
  - They are "undefined" in MPI-3

New Proposals at MPI Forum

- MPI Endpoints: possibility to create extra ranks on a MPI process to be attached to a thread (so that each thread has a rank!). This would help interoperability of MPI + other programming systems.
- Persistent Collectives: re-use argument list in a collective that is repeated many times.

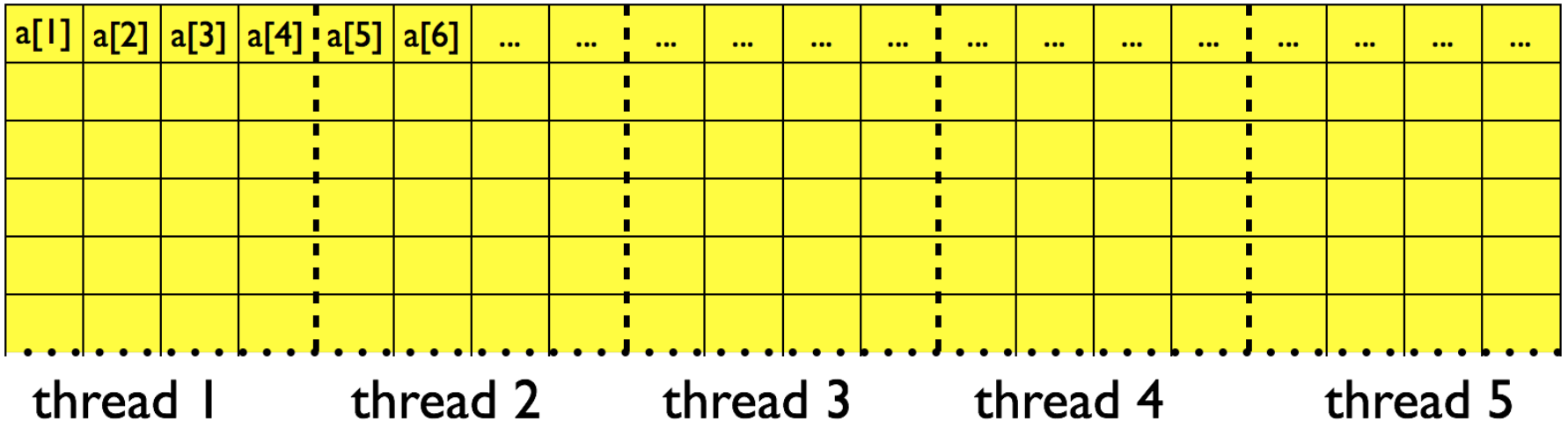
Partitioned Global Address Space (PGAS)

Programming Model for distributed systems (like MPI) that looks like OpenMP. UPC, Coarray Fortran, Chapel are PGAS languages; GPI is a PGAS library.



- Global located on one thread but accessible by all the other threads
- Local located on one thread but not accessible by all the other threads. Very fast.

Example: Shared Array



- Implicit communication is an access to global memory that is located on a different thread
- i.e. communication (handled by the language) occurs when we want to calculate:
- a[1] = a[4] + a[5];
- the value a[5] is transferred to thread 1 by implicit communication and summed to a[4] and set in thread 1 as a[1]

UPC (Unified Parallel C)

UPC is a PGAS model extending C language.

It operates in Single Program, Multiple Data (like MPI) fashion: multiple processes execute the same program, but the execution paths can differ.

To allow threads to access both local and remote memory, UPC provides at the program level the concept of two memory spaces: private and shared. Objects declared in private memory space use regular C declarations, e.g.

- int x; // private variable

and are only accessible by a single thread. Objects declared in shared memory space using the "shared" identifier, e.g.

- shared int y; // shared variable

are accessible by all threads.

Hello World in UPC

- Any legal C program is also a legal UPC program.
- If you compile and run it as UPC with P threads, it will run P copies of the program.
- Number of threads specified at compile-time or run-time; available as program variable THREADS
- MYTHREAD specifies thread index (0..THREADS-1)

```
#include <upc.h>  /* needed for UPC extensions */
#include <stdio.h>

main() {
  printf("Thread %d of %d: hello UPC world\n",
         MYTHREAD, THREADS);
}
```

Calculate PI in UPC

```
main(int argc, char **argv) {
  int i, hits, trials = 0;
  double pi;

  if (argc != 2)trials = 1000000;
  else trials = atoi(argv[1]);

  srand(MYTHREAD*17);

  for (i=0; i < trials; i++) hits += hit();
  pi = 4.0*hits/trials;
  printf("PI estimated to %f.", pi);
}
```

Each thread gets its own copy of these variables

Each thread can us input arguments

Initialize random in math library

Each thread calls “hit” separately

- Required includes:

```
#include <stdio.h>
#include <math.h>
#include <upc.h>
```

- Function to throw dart and calculate where it hits:

```
int hit(){
  int const rand_max = 0xFFFFFF;
  double x = ((double) rand()) / RAND_MAX;
  double y = ((double) rand()) / RAND_MAX;
  if ((x*x + y*y) <= 1.0) {
       return(1);
  } else {
       return(0);
  }
}
```

Co-Array Fortran

Fortran Co-Arrays are an example of a PGAS model and he is part of the Fortran90 Standard.

Co-array Fortran programs follow a SPMD model. Each replication is referred to as an image

The execution path may differ from image to image: each image has a unique identifier that can be used in control statements (codimension). A new, co-dimension syntax is used in addition to the standard array dimension syntax. For example,

- real :: x(10)[*] → declares a co-array which has a dimension of size 10 on each image.

How to use UPC/Fortran on Beskow

- UPC and CAF are present on the Beskow supercomputer (All Cray machines support them)
- Easy to compile: add a flag!
- Instructions on the PDC website

```
module swap PrgEnv-pgi PrgEnv-cray
```

To compile your UPC code:

```
cc -h upc -o UPCexec UPCcode.c
```

Note that "-h upc" flag for the cc compiler.

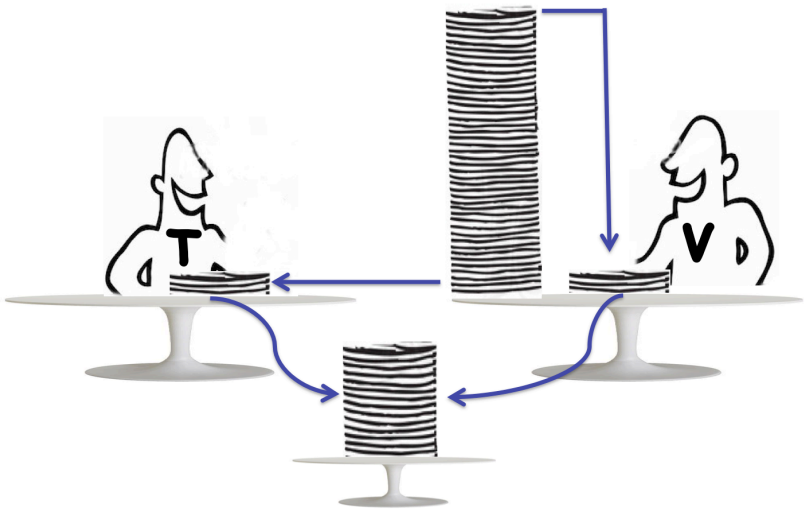
To run the UPC code, use the aprun command. This batch script is an example for running the UPC code on 4 cores of 1 node:

```
#PBS -l mppwidth=4
#PBS -l mppnppn=4
#PBS -l walltime=00:10:00
#PBS -j eo
#PBS -V

cd $PBS_O_WORKDIR
aprun -n 4 ./UPCexec > output_upc
```

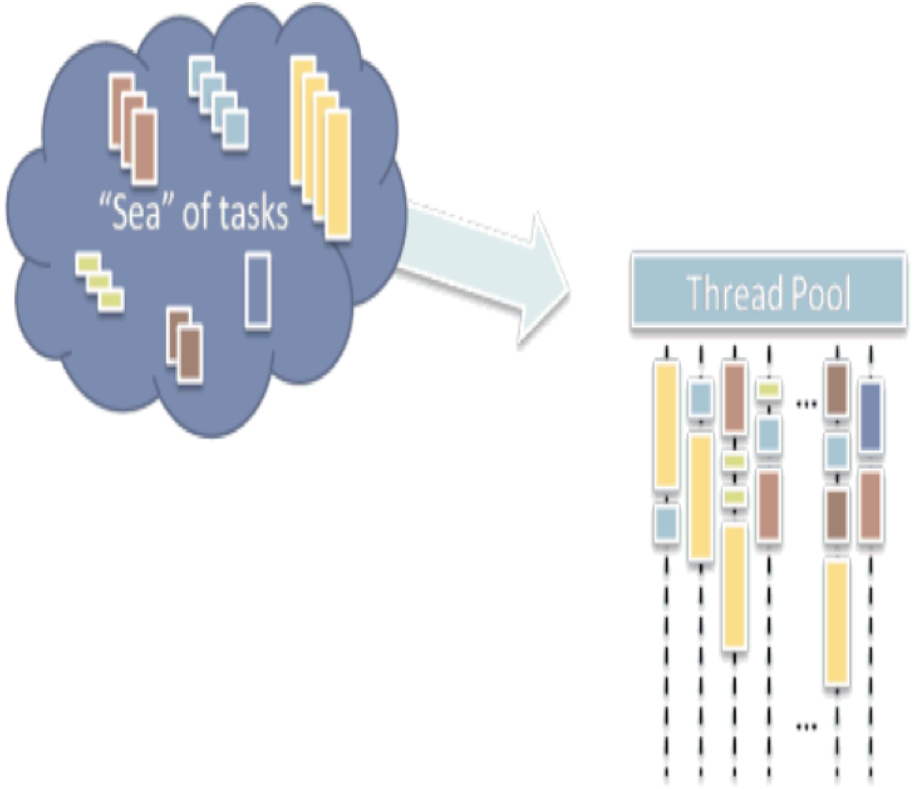
The Task-Based Approach

- It is based on the over-decomposition of the work: we divide the work in more tasks than processors. Tasks are assigned to a work-pool, and available processors take next task from the pool whenever they finish the job.
- The task-based programming approach is the best fit to solve the dynamic load balancing problem.



Work-Pool and Scheduler

- How Many threads ?
- Too few threads will undersubscribe the system → waste some of the available hardware resources.
- Too many threads → oversubscribe the system, causing the operating system to have overhead.
- One common way to perform the balancing act is to create a pool of threads.
- The application then dynamically schedules computations (tasks) on to threads in the thread pool.



Task-Based Approaches

- CILK → extension to C including keywords to handle parallel computing. Developed at MIT and then bought by Intel. For shared memory.
- Intel Thread Building Block (TBB) → Intel template library for C++. A typical code with TBB creates, synchronizes and destroy graphs of depend tasks. For shared memory.
- OpenMP Tasks (saw last week)

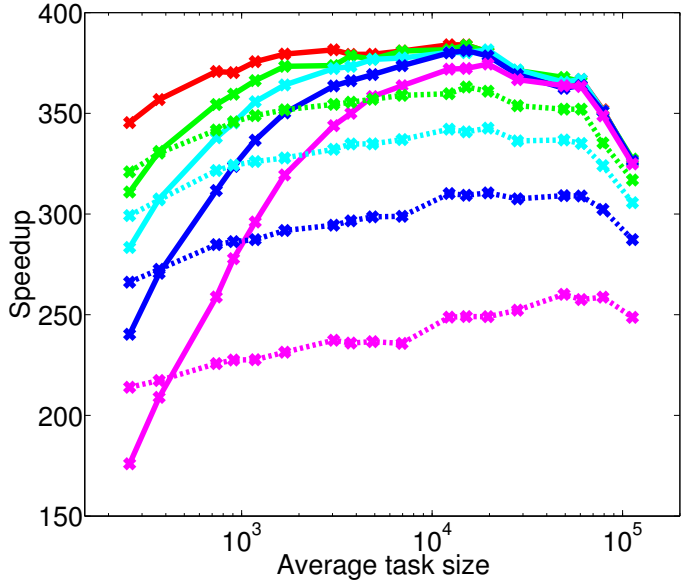
Task-Based Approaches

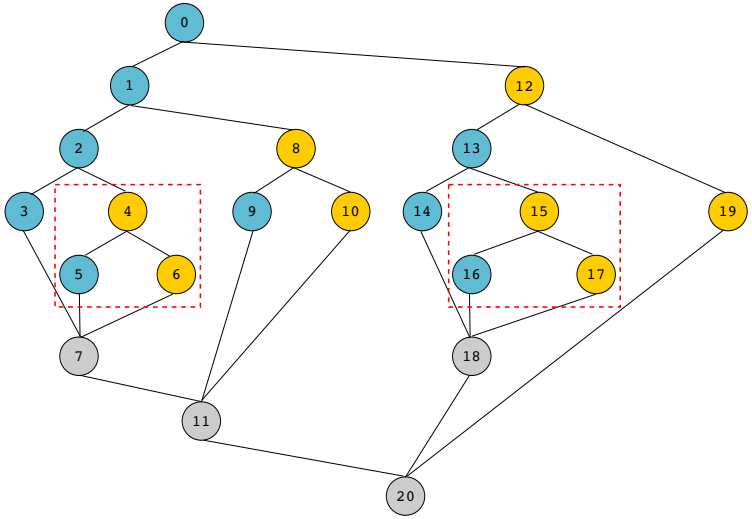
- CILK → extension to C including keywords to handle pa at MIT and then emory.
- Intel Thre tel template e with TBB creat graphs of depend
- OpenMP

ONLY FOR SHARED MEMORY MACHINES

Convenience of Task Based Approaches

- Task approaches solve the problem of dynamic of load balancing but with extra-cost of a scheduler (overhead).
- In order for an application to benefit from task approach, your algorithm need to be formulated in terms of tasks in a DAG graphs





Dana Akhemetova et al., *On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems*, CLUSTER2015 proceedings.

Conclusions

- Future programming approaches will address new challenges coming from HW:
  - Imbalance and delay propagation problem
    - Task-based approaches, Non-blocking collectives, PGAS
- Incremental vs Disruptive
  - New features in MPI
  - PGAS: UPC and CAF
  - Task-based programming approaches

Extra Slides

Fault tolerance

- The trend to extend performance of supercomputers is to increase the number of computing units. As a consequence, the number of faults is expected to increase, as one can expect that one of the millions components will always be broken.
- The current state of the art automatic and transparent fault tolerant techniques based on checkpoint/restart do not scale and will no longer work, and hardware methods based on replications are quite expensive.
  - → Need first to detect one or processes are failing, and communicate to all other processes (global)
  - → Dynamic process creation and destruction

New Emerging Applications: Big Data

- Exponential growth of data to be analyzed. Think about genomics, amazon cloud, astrophysics.
- Bill Gropp: “"*Big Data starts at 1.5 PB because that's what fits in memory on Blue Waters*”.
- SKA will produce 200GB/s of raw data.
- Parallel programming models are designed for solving PDE not for online processing!
- Need to support streaming models!