# Big Data Summer School

Lecture 1. Introduction and Parallel Computing

Wenguang CHEN
Tsinghua University

# What is big data?

- Big data is a broad term for data sets so large or complex that traditional data processing applications are inadequate.
- 3Vs: Volume， Variety， Velocity
- Challenges include analysis, capture, curation, search, sharing, storage, transfer, visualization, and information privacy.

# What is big data?

- From the computer system perspective
  - The volume of data and the performance expectation of data analysis become key factors of data processing system design, we call it's a big data problem
- No fixed size
  - Simple keyword query, TB – PB
  - Data mining, GB – TB
    - Consider a shortest path algorithm on a 100 million node gragh

# Big data examples


Astronomy


Weather


Genetics

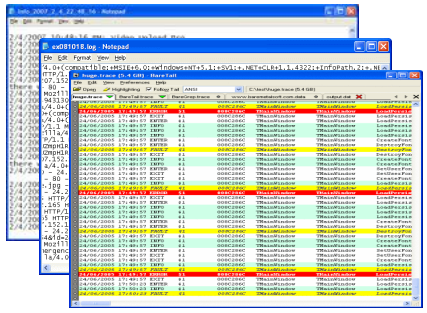

Health

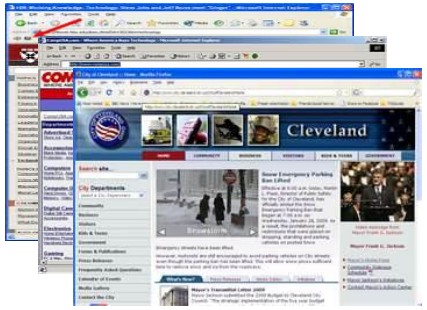

Economy


Physics


Others

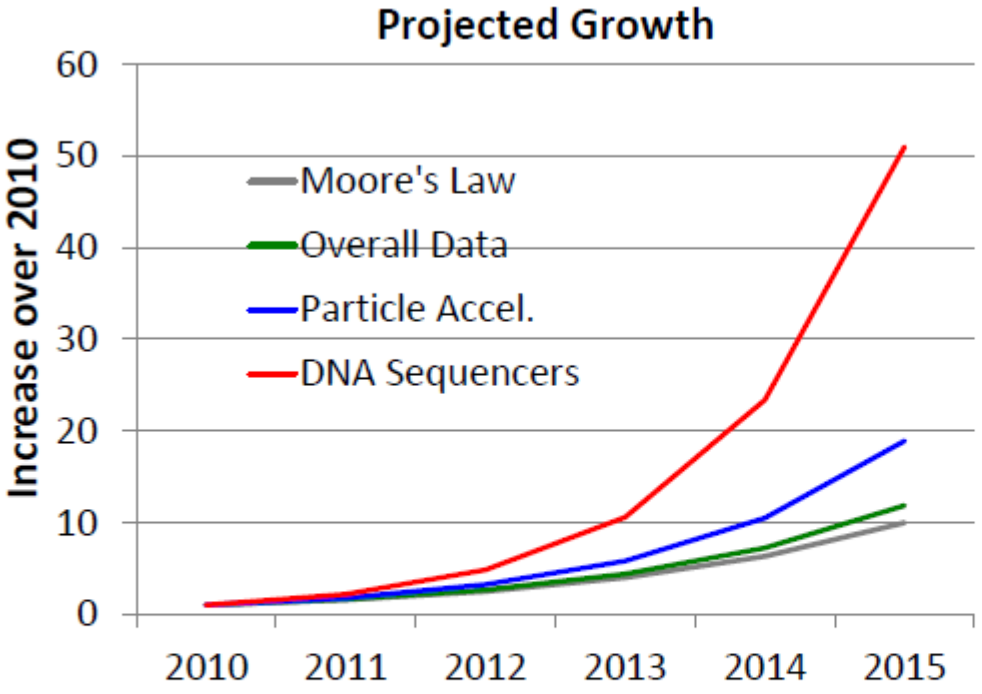

UGC

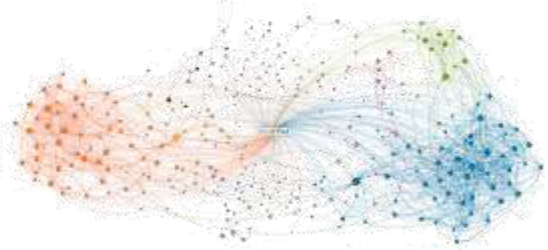

Deep Web



Multi-modal

# Data growth



Data Grows faster than Moore's Law

[IDC report, Kathy Yelick, LBNL]

# Social network data-Volume

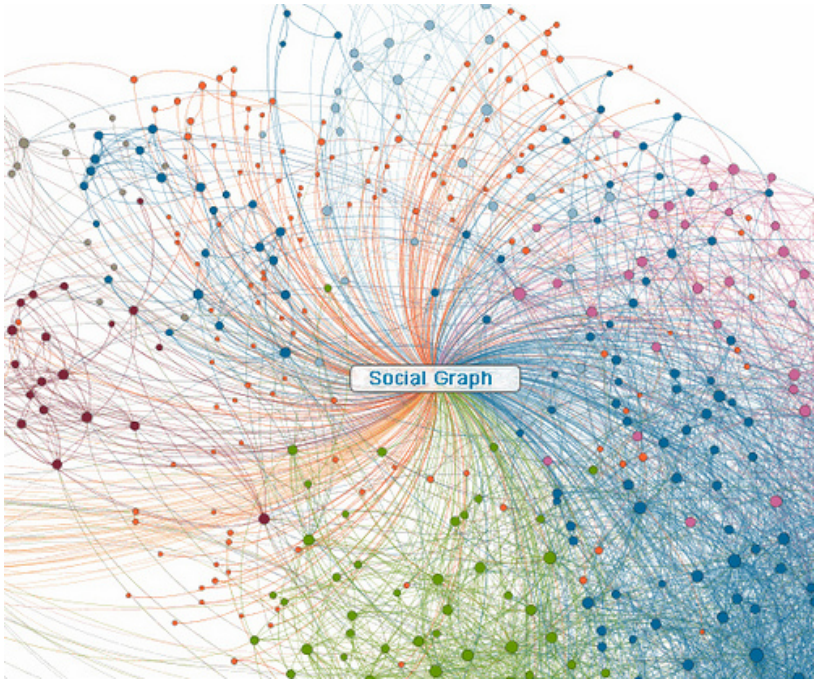
- Tweets of Sina weibo 10million users
  - At most 1000 tweets per user – 5TB
  - 300 million users ~ 100TB
  - Comments, images and video are
- User Profile
  - 100GB
- Connections between users
  - 100 million users, 10billion connections, ~100GB
  - Several billion users, ~ several terabytes

# Social network data-Velocity

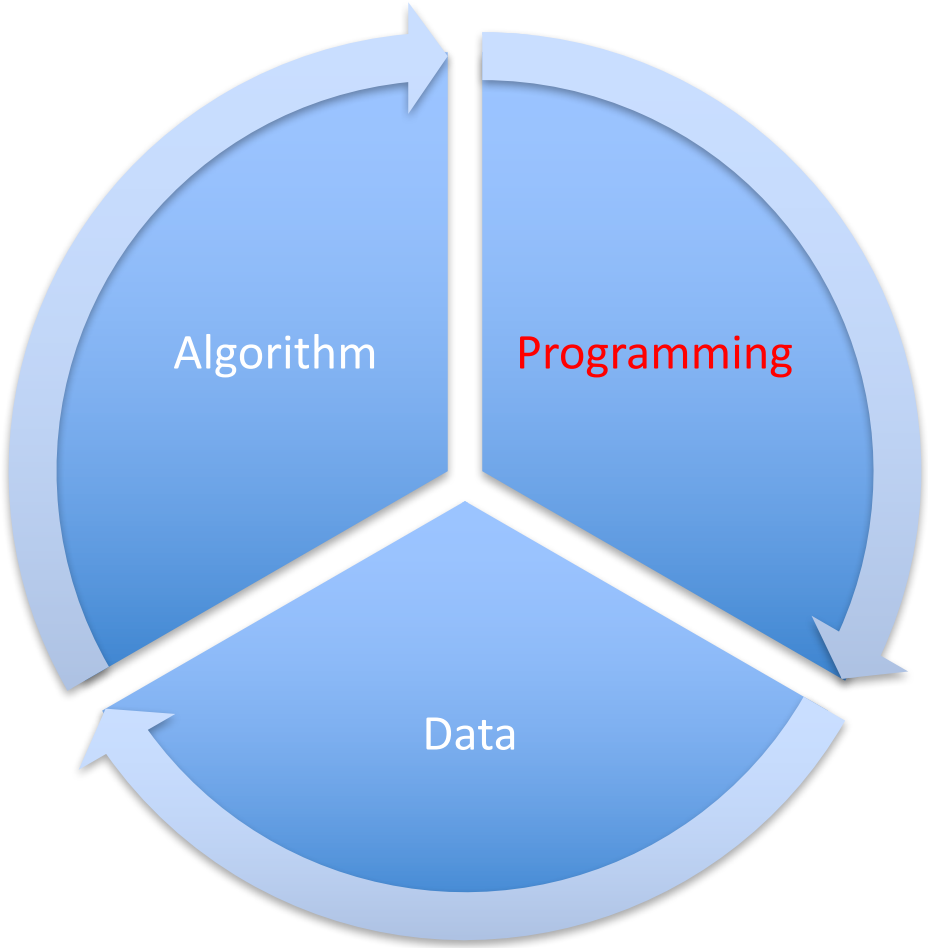
- Several hundred million tweets generated by Weibo
  - $512 * 10^8$ Byte ~= 50GB / day
- Evolution of connections
  - More users
  - 40 K new users / day
  - New connections and removed connections

# Social network data-Variety

- Tweets – Natural language
- Profile / Tags
- Connections – Graph
  - Unstructured data
- Retweets and comments
  - Graph and natural language



# Essentials for Big Data



# Paralellism

# Parallelism

**Sequential**

**Parallel**

**Pipelined**

- Parallelism is employed to provide higher performance with more cost
- Why this is beneficial?

# Motivations for Parallellism

- Require more resources to achieve the performance
  - CPU
  - Memory
  - I/O

# Fundamentals of Parallelism

- Task parallelism

```
calculate_direction_x(m);
calculate_direction_y(m);
calculate_direction_z(m);
```

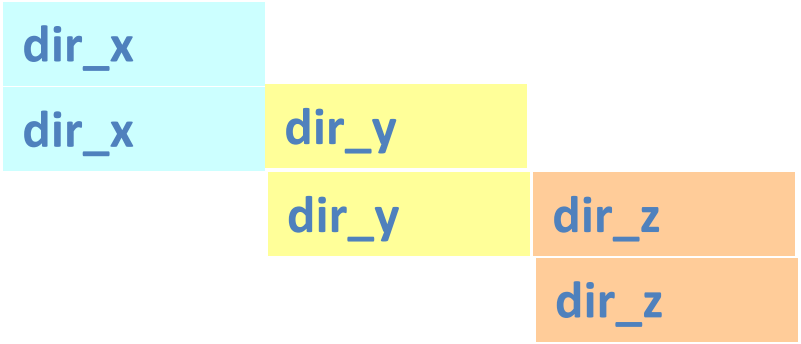

- Data parallelism

```
for ( i=0; i<N; i++)
   a[i] = b[i];
```

```
a[1:N] = b[1:N]
```

- Pipeline parallelism

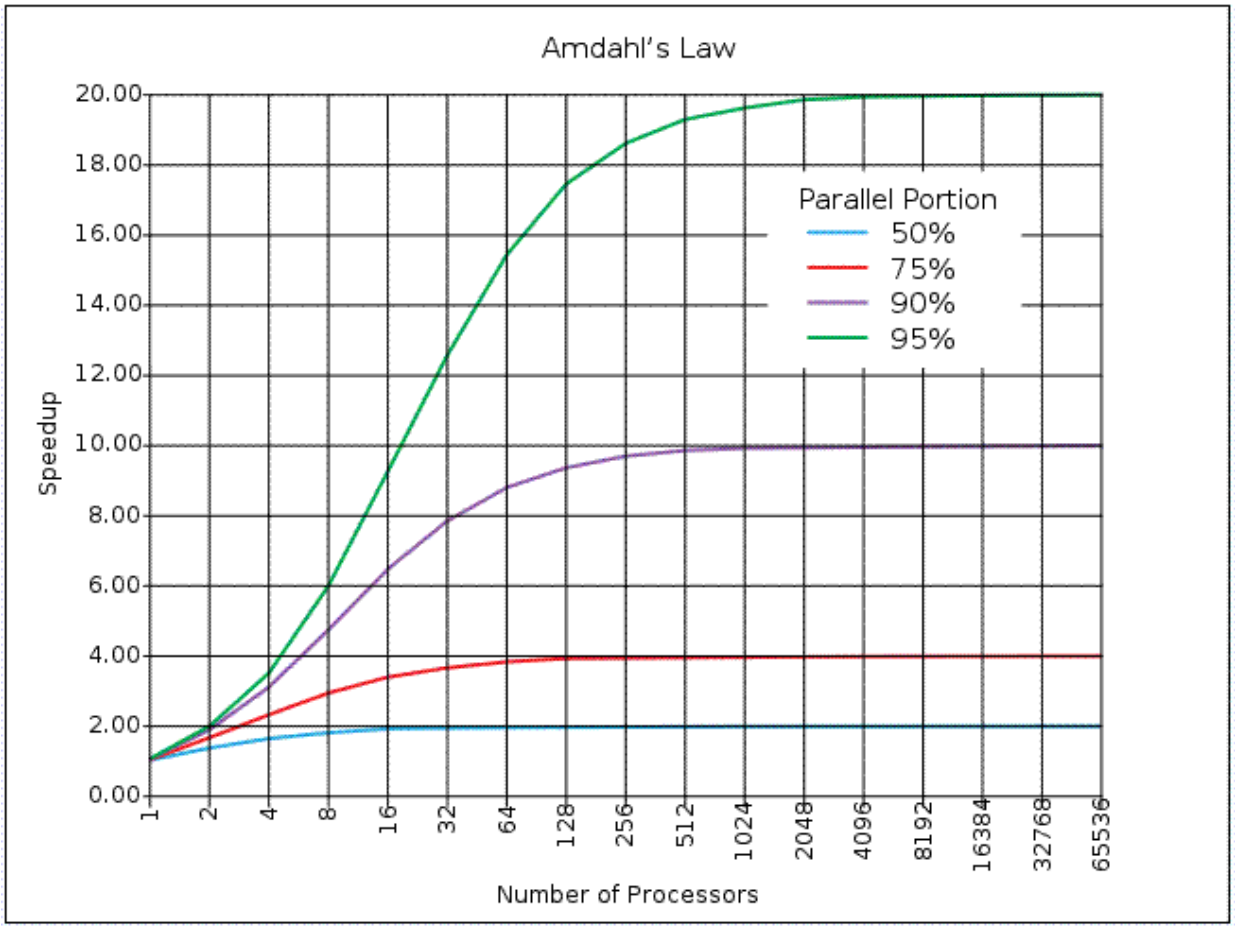
```
x = calculate_direction_x(m);
y = calculate_direction_y(x);
z = calculate_direction_z(y);
```



# Two laws in parallel computing(1)

- Amdahl's law

  If P is the proportion of a program that can be made parallel, the limit of speedup of whole program is bounded at 1/(1-P)



# Two laws in parallel computing(2)

- Gustafson's law
  - Amdahl's law assume the fixed problem size
  - If problem size can increase as the compute power increases, the speedup that can be achieved is much larger, which is
    - S = a(n) + p(1-a(n))
      - n, problem size
      - p, number of processors
      - a(n), serial portion at problem size n
    - S → p when a(n) → 0

Reevaluating Amdahl's Law ,Communications of the ACM 31(5),1988. pp. 532-533

# In the real world

- Gustafson's law rescues parallel computing
  - Larger machines are still useful
  - Many small chips can deliver good performance when the problem size is big
    - Blue Gene/L
- However, some applications do not have bigger data set
  - Genome alignment, social network
    - What happens when fixed problem size meet Moore's law?
- Real world are mix of Amdahl's law and Gustafson's law

# Challenges of Parallel Programming

- Parallelism identification
  - Rely on developers to specify
    - May need incremental parallelization to achieve the performance target
- Load balance
  - The completion time of a parallel program is its last thread's completion time
- Proper synchronization
  - Make sure the programs are parallelized correctly
    - Data race and deadlock free
    - Respect dependence of original program

# So what is the rule behind correct parallelism?

Let’s examine it by examples.

Can following loops be parallelized?

```
//Loop 1
for(int i=0; i<N; i++)
{
   a[i] = a[i]+1.0;
}

//Loop 2
for(int i=1; i<N; i++)
{
   a[i] = a[i]+a[i-1];
}
```

# How Loop 2 are executed in parallel?

```
for(int i=1; i<=4; i++)
{
    a[i] = a[i]+a[i-1];
}
```

A possible parallel execution of the loop:

**Thread 1**

```
a[1]=a[1]+a[0];
a[2]=a[2]+a[1];
```

**Thread 2**

```
a[3]=a[3]+a[2];
a[4]=a[4]+a[3];
```

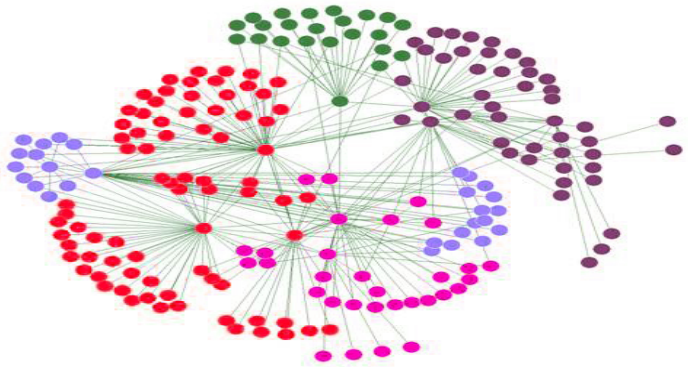
- Thread 2 may use the old a[2], before the right a[2] is calculated

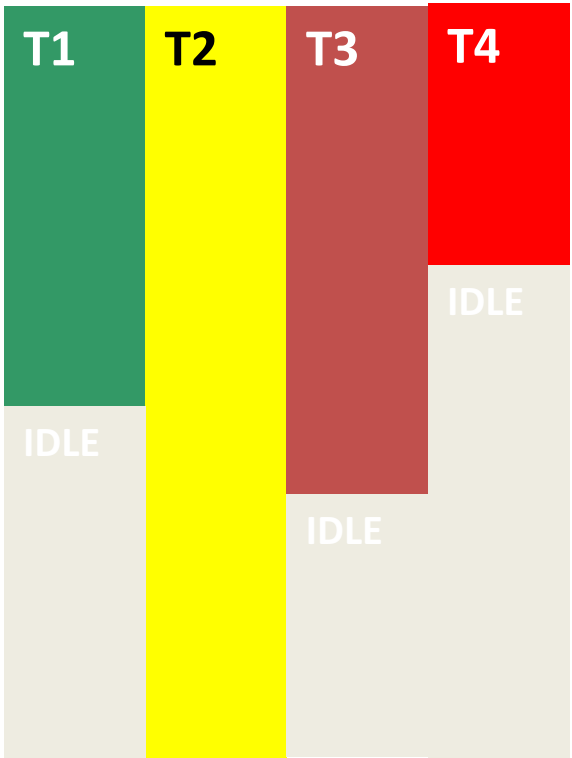
# Dependence blocks parallelization

- Can not parallelize a loop unless we know there’re no cross-iteration data dependence
- But aliasing and procedure boundary prevent compiler from accurate and efficient dependence test
- An optimization technique called speculative parallelization, which could get the 30% of performance improvement with 4 cores
  - Is this a good research direction? Why?

# Load Balance

- Both Amdahl and Gustafson's law assume perfect parallelization.
- Load imbalance is one of the most significant performance bottleneck
  - Speedup is damaged seriously
- Scheduling loops to processors
- Solutions?



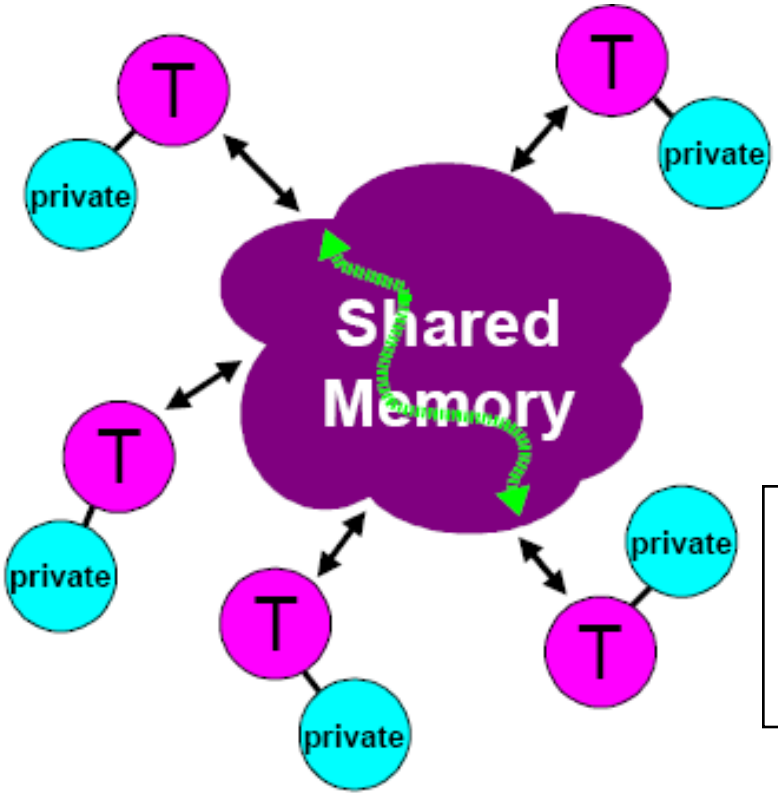


# How and Why Parallelism

- Avoid parallelism if possible
  - Check single threaded solution first
  - Keep the power issues in mind
- Difficult to get parallelism correctly and efficiently
  - Dependences
  - Load balancing
  - Data races and deadlocks
  - Uncertainty

OK, Let's see how to do parallelism

# Shared Memeory Programming Model




- Shared Memory
  - Thread still may have private data
  - Communications are implicitly done

| Thread 1 | Thread 2 |
|---|---|
| s1:a = b + c<br>S2:g = a + f | S3:f = e + d |

- Synchorinization is required to makes sure that S2 should execute AFTER s3

OpenMP, Pthreads

Windows Threads

# Examples: Measure memory bandwidth with threads

- Single threaded version

```
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result;
}
```

# Main threads

```
void *sum_thread(void *);

volatile long global_sum = 0; /* global */


main()
{
…
  for (int i = 0; i < N; i++)
       pthread_create(&tid[i], NULL, sum_thread, (void*)i);

   for (int i = 0; i < N; i++)
       pthread_join(tid[i], NULL);
}
```

# Thread

```
void *sum_thread(void *vargp)
{
        long sum;
        int myid = (long) vargp;
        int start = myid * sample_num / 4;
        int end = (myid+1) * sample_num / 4 - 1;

        sum = 0;
        for (int i=start; i<=end; i++)
            global_sum += x[i];

        return NULL;
}
```

# Thread

```
void *sum_thread(void *vargp)
{
        long sum;
        int myid = (long) vargp;
        int start = myid * sample_num / 4;
        int end = (myid+1) * sample_num / 4 - 1;

        sum = 0;
        for (int i=start; i<=end; i++)
            sum += x[i];

        sem_wait(&num_mutex);
        global_sum += sum;
        sem_post(&num_mutex);

        return NULL;
}
```

# Issues with Pthreads

- Flexible and powerful
  - Useful not only in data processing, but also in a wide range of systems
    - Network server etc.
- Complex for data processing
  - Pthread is not always portable( e.g. Windows )
  - Code to calculate addresses and loop bounds
  - May not be easy to specify scheduling schemes

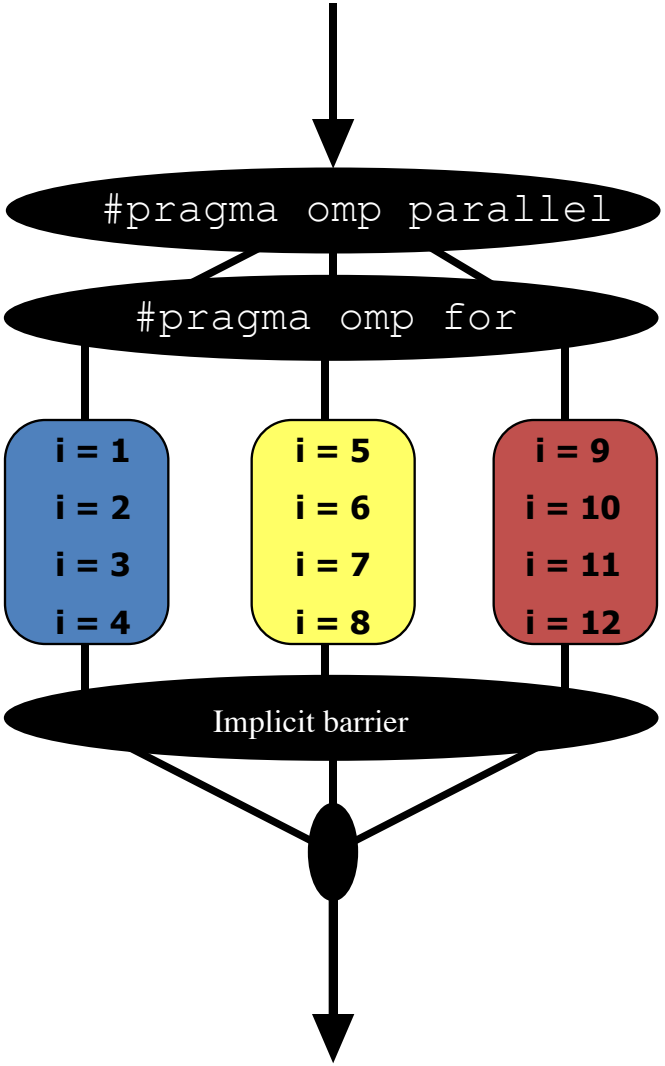
# Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing [Short Version]
  - "Standard" API for defining multi-threaded shared-memory programs
  - openmp.org – Talks, examples, forums, etc.
  - **computing.llnl.gov/tutorials/openMP/**
  - **See parlab.eecs.berkeley.edu/2012bootcampagenda**
    - **2 OpenMP lectures (slides and video) by Tim Mattson**
  - portal.xsede.org/online-training
  - www.nersc.gov/assets/Uploads/XE62011OpenMP.pdf
- High-level API
  - Preprocessor (compiler) directives ( ~ 80% )
  - Library Calls ( ~ 19% )
  - Environment Variables ( ~ 1% )

# 并行结构：Work-sharing Construct(1) - loop

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

```
#pragma omp parallel
#pragma omp for
   for(i = 1, i < 13, i++)
      c[i] = a[i] + b[i]
```



# Combining pragmas

- These two code segments are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++)
       res[i] = huge();
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
         res[i] = huge();
    }
```

# Assigning Iterations

- The schedule clause affects how loop iterations are mapped onto threads

**`schedule(static [,chunk])`**

- Blocks of iterations of size "chunk" to threads
- Round robin distribution
- Default=N/t

**`schedule(dynamic[,chunk])`**

- Threads grab "chunk" iterations
- When done with iterations, thread requests next set
- Default=1

**`schedule(guided[,chunk])`**

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"
- Default=1

**`schedule(runtime)`**

- OMP_SCHEDULE

# Schedule Clause Example

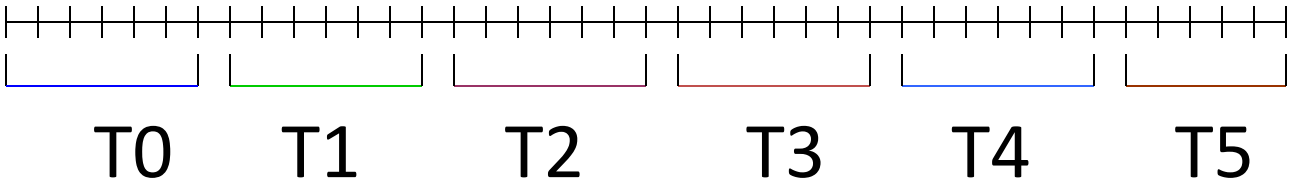
```
#pragma omp parallel for schedule (static)
    for( int i = 1; i <= 128; i ++ )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

```
#pragma omp parallel for schedule (static, 8)
    for( int i = 1; i <= 128; i ++ )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

```
#pragma omp parallel for schedule (dynamic)
    for( int i = 1; i <= 128; i ++ )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

# Static Scheduling

For $N_i$ iterations and $N_t$ threads, each thread gets one chunk of $N_i/N_t$ loop iterations:



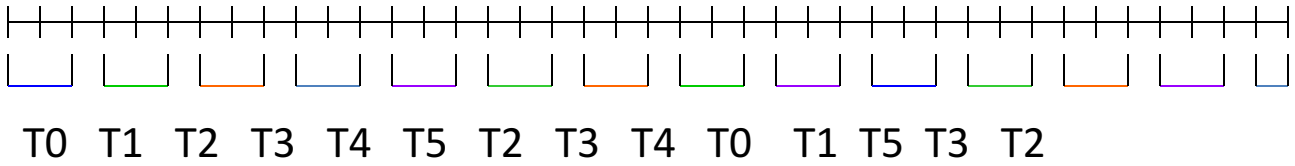

- Thread #0: iterations 0 through $N_i/N_t$-1
- Thread #1: iterations $N_i/N_t$ through $2N_i/N_t$-1
- Thread #2: iterations $2N_i/N_t$ through $3N_i/N_t$-1

…

- Thread #$N_t$-1: iterations $(N_t\text{-}1)N_i/N_t$ through $N_i$-1

# Dynamic Scheduling

For $N_i$ iterations and $N_t$ threads, each thread gets a fixed-size chunk of $k$ loop iterations:

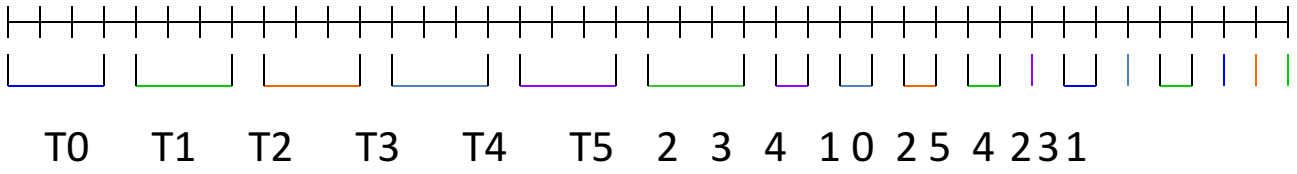



When a particular thread finishes its chunk of iterations, it gets assigned a new chunk. So, the relationship between iterations and threads is nondeterministic.

- Advantage: very flexible
- Disadvantage: high overhead – lots of decision making about which thread gets each chunk

# Guided Scheduling

For $N_i$ iterations and $N_t$ threads, initially each thread gets a fixed-size chunk of $k < N_i/N_t$ loop iterations:




After each thread finishes its chunk of k iterations, it gets a chunk of $k/2$ iterations, then $k/4$, etc. Chunks are assigned dynamically, as threads finish their previous chunks.

- Advantage over static: can handle imbalanced load
- Advantage over dynamic: fewer decisions, so less overhead

# Scheduling in OpenMP

| Schedule Clause | When To Use |
| --- | --- |
| STATIC | Predictable and similar work per iteration |
| DYNAMIC | Unpredictable, highly variable work per iteration |
| GUIDED | Special case of dynamic to reduce scheduling overhead |

# Example: Dot Product

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

What is Wrong?

# Protect Shared Data

- Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

# OpenMP Critical Construct

- **`#pragma omp critical [(lock_name)]`**
- Defines a critical region on a structured block

**Threads wait their turn –at a time, only one calls `consum()` thereby protecting RES from race conditions**

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for privte(B)
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical
    {
    consum (B, RES);
    }
  }
}
```

# Do we have something similar to this in OpenMP?

```
void *sum_thread(void *vargp)
{
        long sum;
        int myid = (long) vargp;
        int start = myid * sample_num / 4;
        int end = (myid+1) * sample_num / 4 - 1;

        sum = 0;
        for (int i=start; i<=end; i++)
            sum += x[i];

        sem_wait(&num_mutex);
        global_sum += sum;
        sem_post(&num_mutex);

        return NULL;
}
```

# OpenMP Reduction Clause

- **`reduction (`*`op`*` : `*`list`*`)`**

- The variables in "*list*" must be shared in the enclosing parallel region

- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the "op"
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

# C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

| Operand | Initial Value |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |
| ^ | 0 |

| Operand | Initial Value |
|---|---|
| & | ~0 |
| \| | 0 |
| && | 1 |
\|\| | 0 |

# Protect Shared Data

- Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)  reduction(+:sum)
for(int i=0; i<N; i++) {
#pragma omp critical
    sum += a[i] * b[i];
  }
  return sum;
}
```

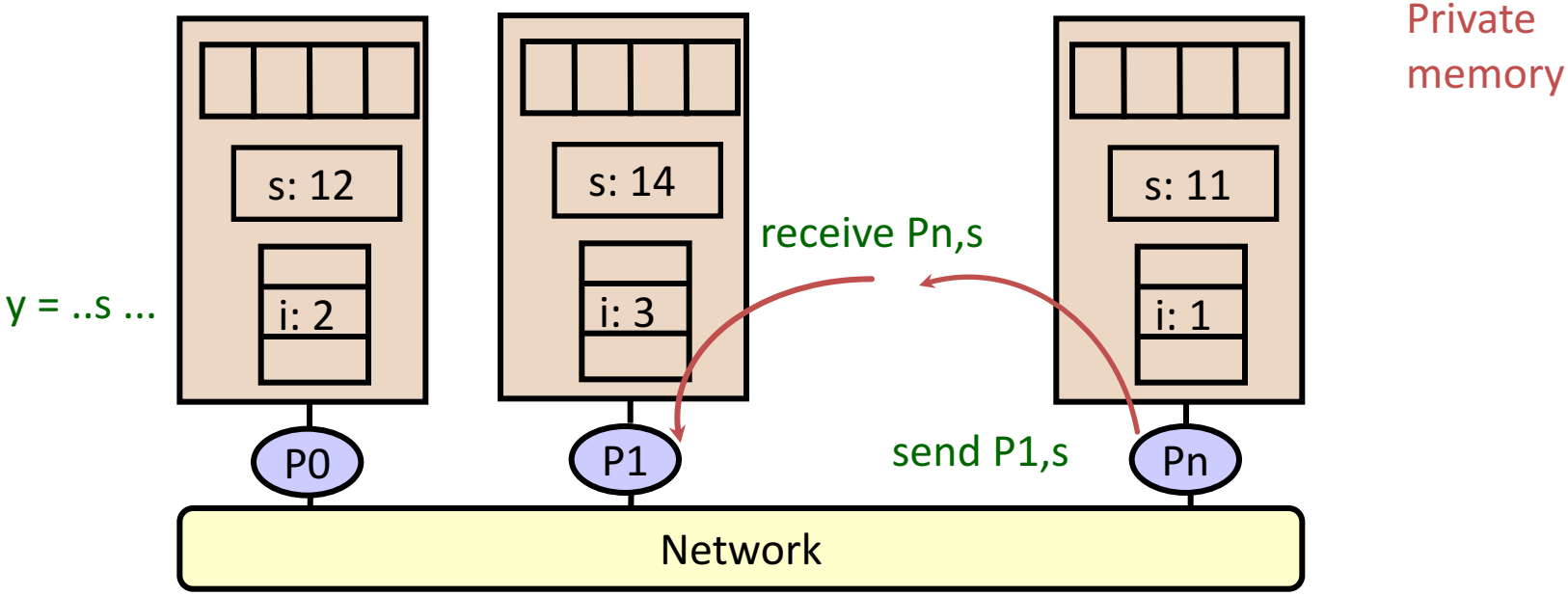
# We have talked a lot on single machine

- OpenMP (or other threading ) is good for a single machine
  - Shared memory space
  - Incremental parallelization
  - RAID can be used to speedup I/O
- We still have more to optimize for a single machine, will discuss later
  - NUMA for task allocation
  - Mmap for I/O
- Let’s try to use multiple machines now

# Task/Channel Model

- Parallel computation = set of tasks
- Task
  - Program
  - Local memory
  - Collection of I/O ports
- Tasks interact by sending messages through channels

- **The message passing paradigm**
  - **Parallel PROCESSES**
  - **Separate memory spaces**
  - **Communication through explicit message send/recv**
  - **Need rewrite code, incremental annotation is not sufficient**



# A Minimal MPI Program (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

# Better Hello (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

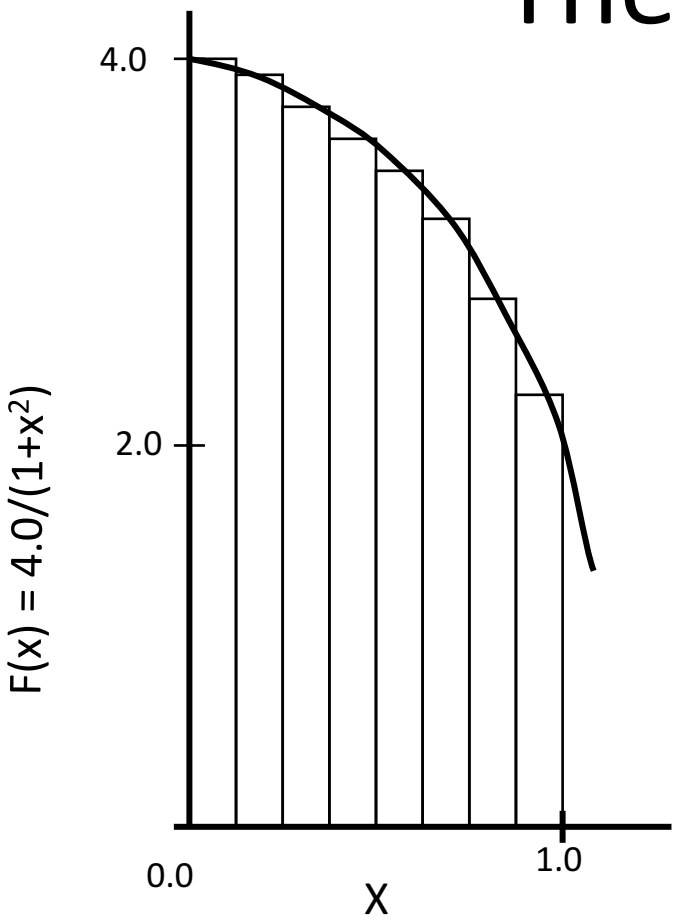
# Send_recv samples

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

tag=1234;source=0;destination=1;count=1;

 if(myid == source){
   buffer=5678;
   MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
   printf("processor %d  sent %d\n",myid,buffer);
}
if(myid == destination){
    MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
       printf("processor %d  got %d\n",myid,buffer);
    }

MPI_Finalize();
```

# MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECV`**
- Point-to-point (send/recv) isn't the only way...

# The pi example




$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

```
static long num_steps = 100000;
double step;
void main ()
{int I;
 double x, pi, sum = 0.0;
 step = 1.0/(double) num_steps;
 for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

# The OpenMP version

```
x=0;
sum = 0.0;
step = 1.0/(double) num_steps;

#pragma omp for reduction(+:sum) private(x)
for (i=0; i < num_steps; i=i+1){
      x=(i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
pi=step*sum;
```

What are needed in MPI if we wish each process do part of the calculation?

# Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

# Example: PI in C -1

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

# Example: PI in C - 2

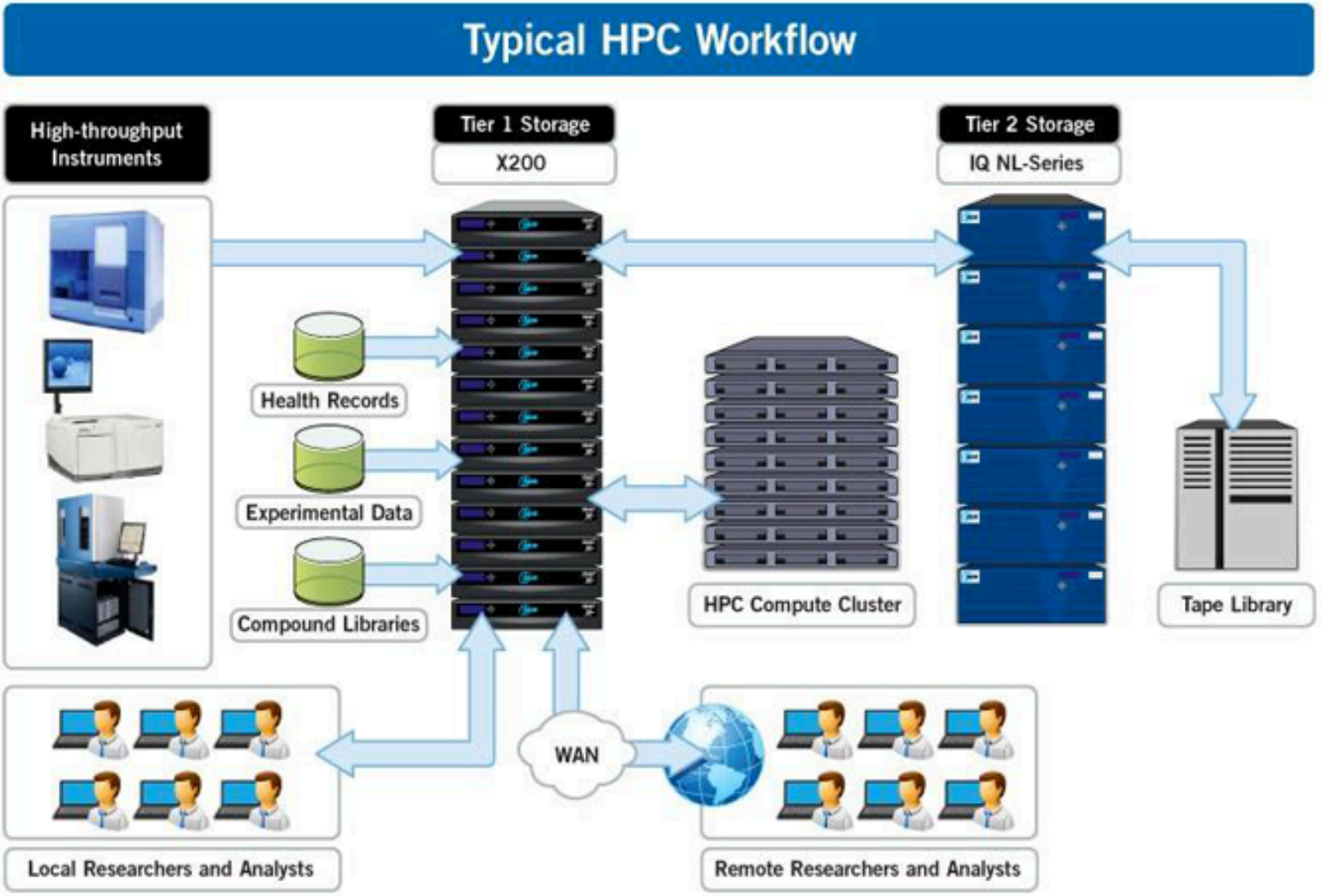
```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

# Alternative set of 6 Functions for Simplified MPI

- **`MPI_INIT`**
- **`MPI_FINALIZE`**
- **`MPI_COMM_SIZE`**
- **`MPI_COMM_RANK`**
- **`MPI_BCAST`**
- **`MPI_REDUCE`**

- What else is needed (and why)?

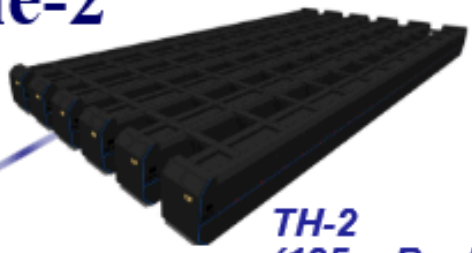
# MPI and HPC cluster architecture



# TIANHE - 2

| | |
|---|---|
| Perf | 54.9PFlops / 33.86PFlops |
| Nodes | 16000 |
| Mem | 1.4PB |
| Racks | 125+8+13+24=170 (720m$^2$) |
| Power | 17.8 MW (1.9GFlops/W) |
| Cool | Close-coupled chilled water cooling |

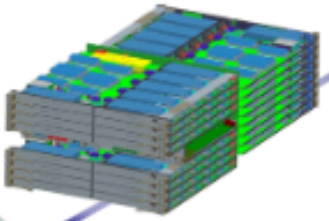
## Highlights of Tianhe-2



TH-2
(125 x Rack)

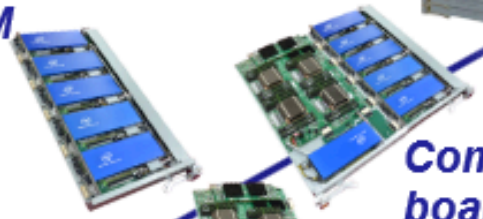
Rack
(8 x Frame)

TH-Express2



Frame
(8 x board)

APM



Compute board

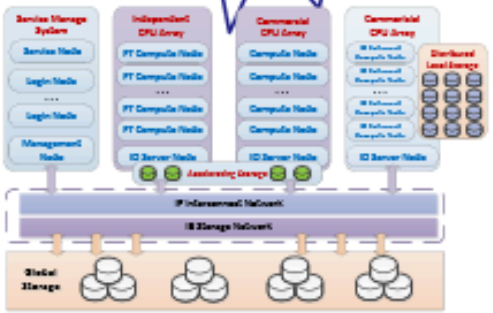
Phi
#48000

IVB
#32000

CPM

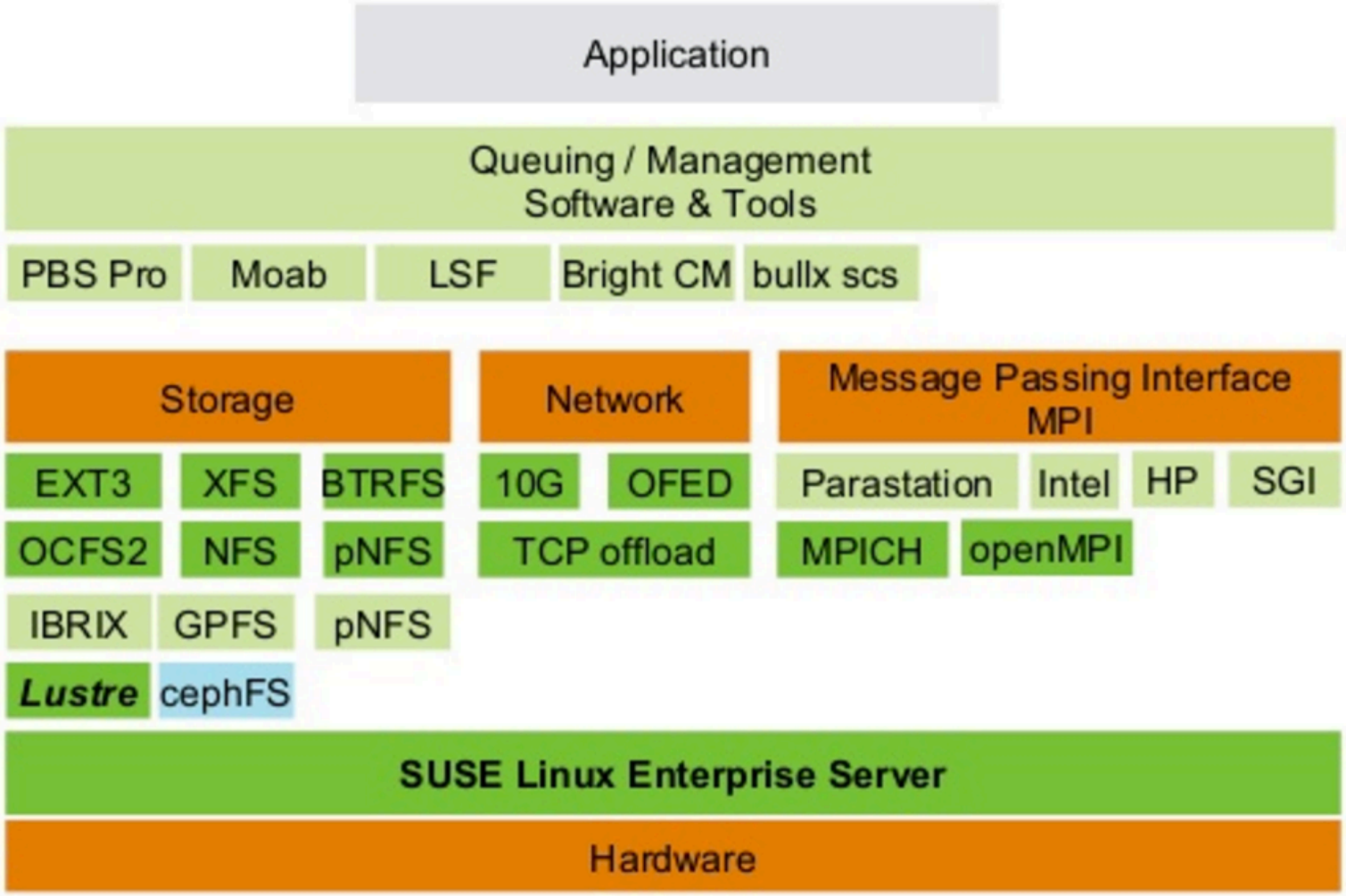
FT-1500
#4096

ION

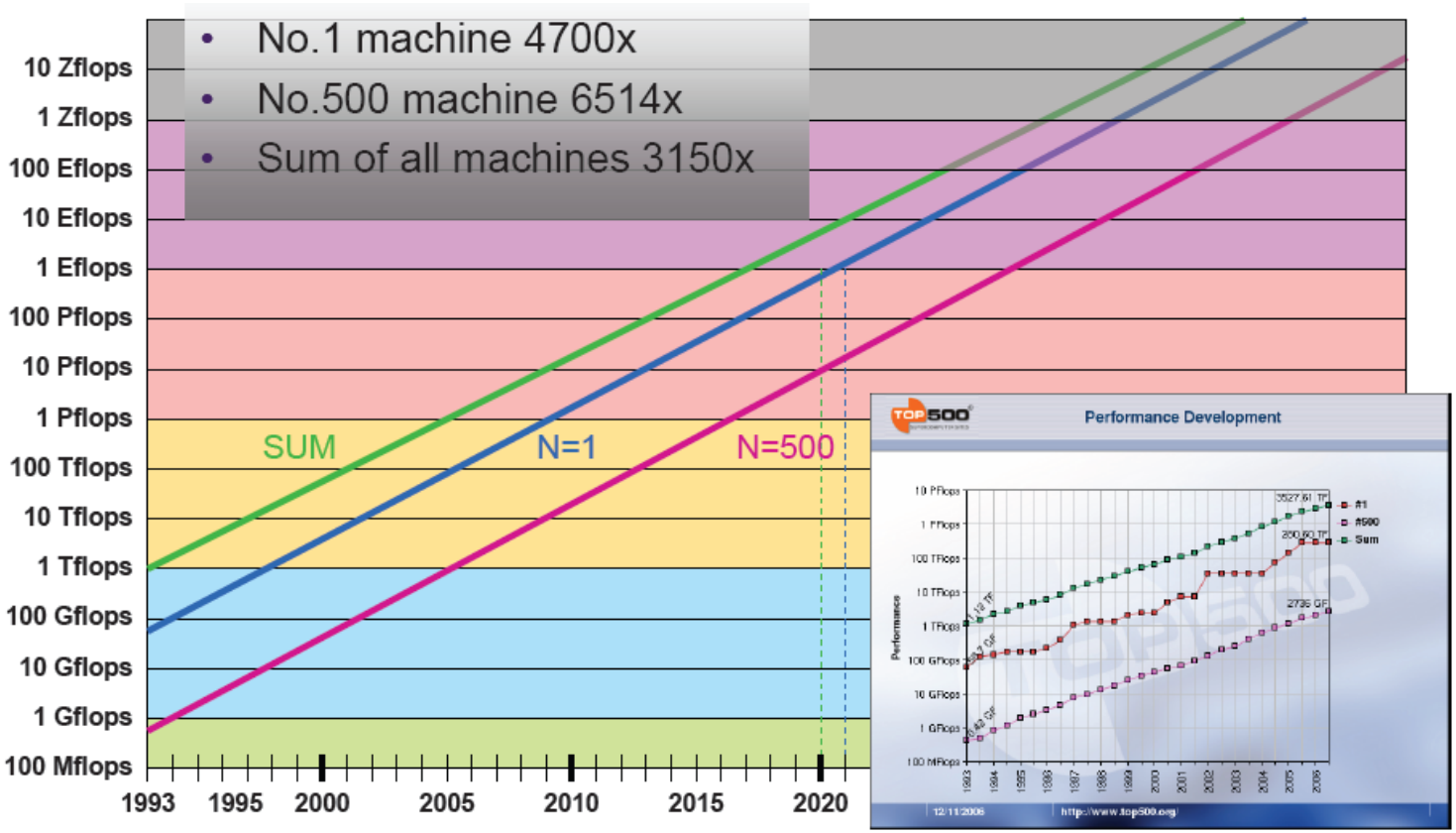


Hybrid Hierarchy shared storage System
H$^2$FS 12.4PB

# Typical Software Stack of HPC



# 10 Year for 1000X Performance

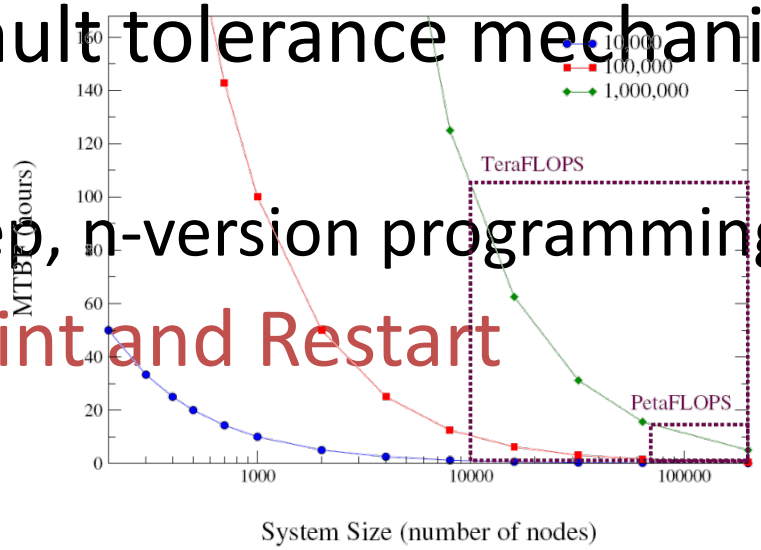



**1998(Tflops) →2008(Pflops) →2018(Eflops)**

**What are the challenges to build Eflops computers and beyond?**

# Availability

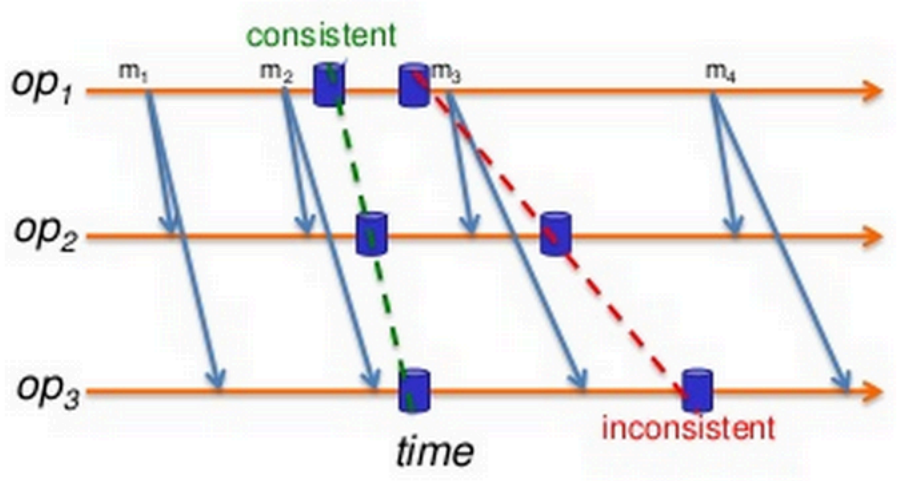
- MTBF of Petaflops machine is likely just a few hours*
  - What if an application runs 20 days or even longer?
  - Restart from the beginning does not help
- Classic fault tolerance mechanisms do not fit well
  - Lockstep, n-version programming…
- Checkpoint and Restart




* Fabrizio Petrini and Kei Davis and José Carlos Sancho. System-Level Fault-Tolerance in Large-Scale Parallel Machines with Buffered Coscheduling. FTPDS04, 2004.

# Checkpoint and Restart

- Applications can write its own checkpoint
  - Compact, only save necessary state variables
  - Not trivial, especially when multiple people are involved



# Automatic checkpoint and rollback recovery

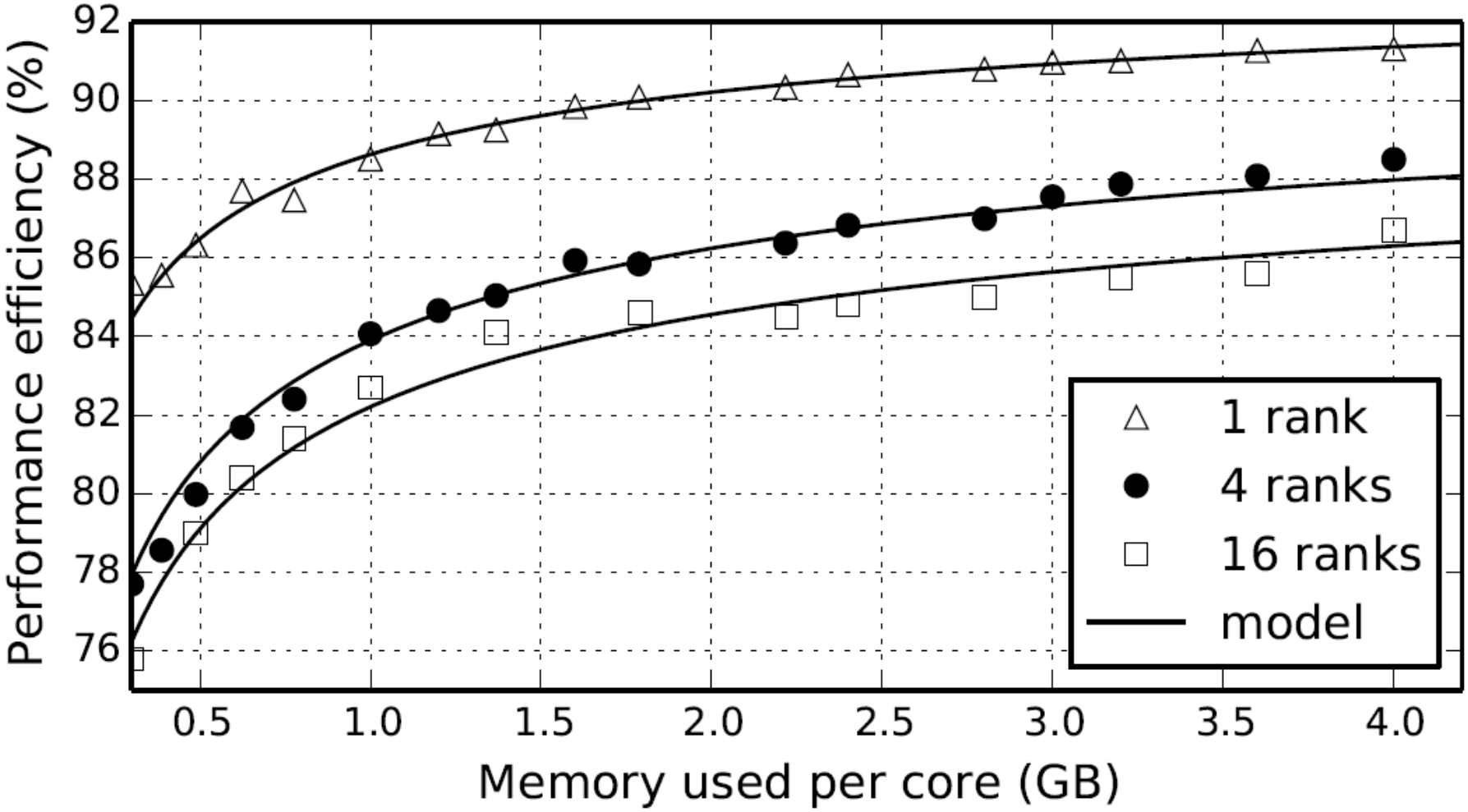
- Good in concept, not in production system yet
  - Heavy I/O overhead on checkpointing
  - Difficult to fit arbitrary program
    - complex process boundary
  - File and Virtual Memory are not consistent

# I/O overhead of Checkpointing

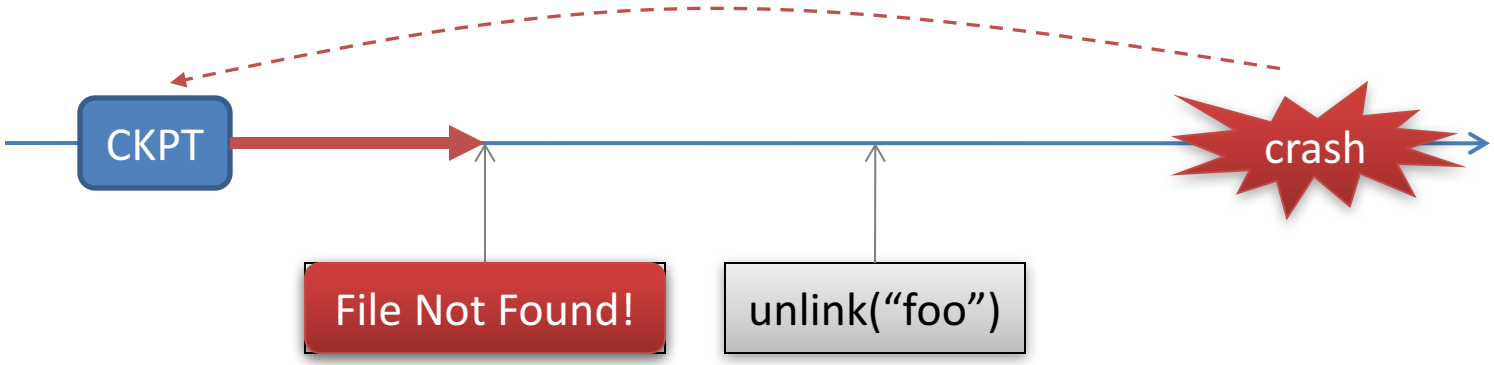
- The amount of time doing useful work
- Bandwidth, memory $\propto$ compute power
  - checkpoint stays constant
- Checkpoint interval: $\sqrt{2 \cdot MTTF \cdot t_{ckpt}}$

* Schroeder, B., Gibson, G.A., “Understanding failure in petascale computers.” SciDAC 2007

# In-Memory Checkpoint and Redundant Execution



# File Checkpointing

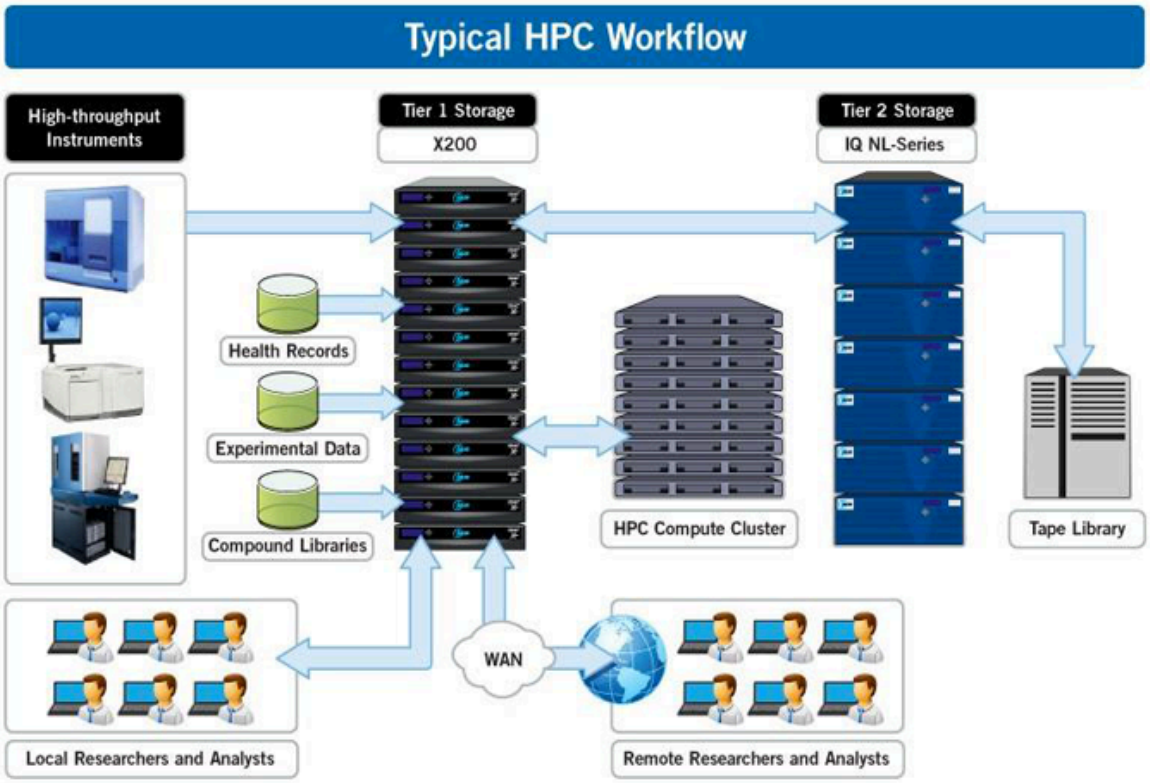



- We propose a way to solve the file system consistency problem by leveraging a user-level file system*
  - Buffer file changes until the next checkpoint

*Ruini Xue, Wenguang Chen, Weimin Zheng: CprFS: A User-Level File System to Support Consistent File States for Checkpoint and Restart. ICS 2008: 114-123

# Revisit HPC cluster architecture

- Separated storage and compute nodes
- Homogeneous compute nodes
- Batch scheduler
  - All or none scheduling policy
- Difficult fault-tolerance



# Challenges to Big data systems

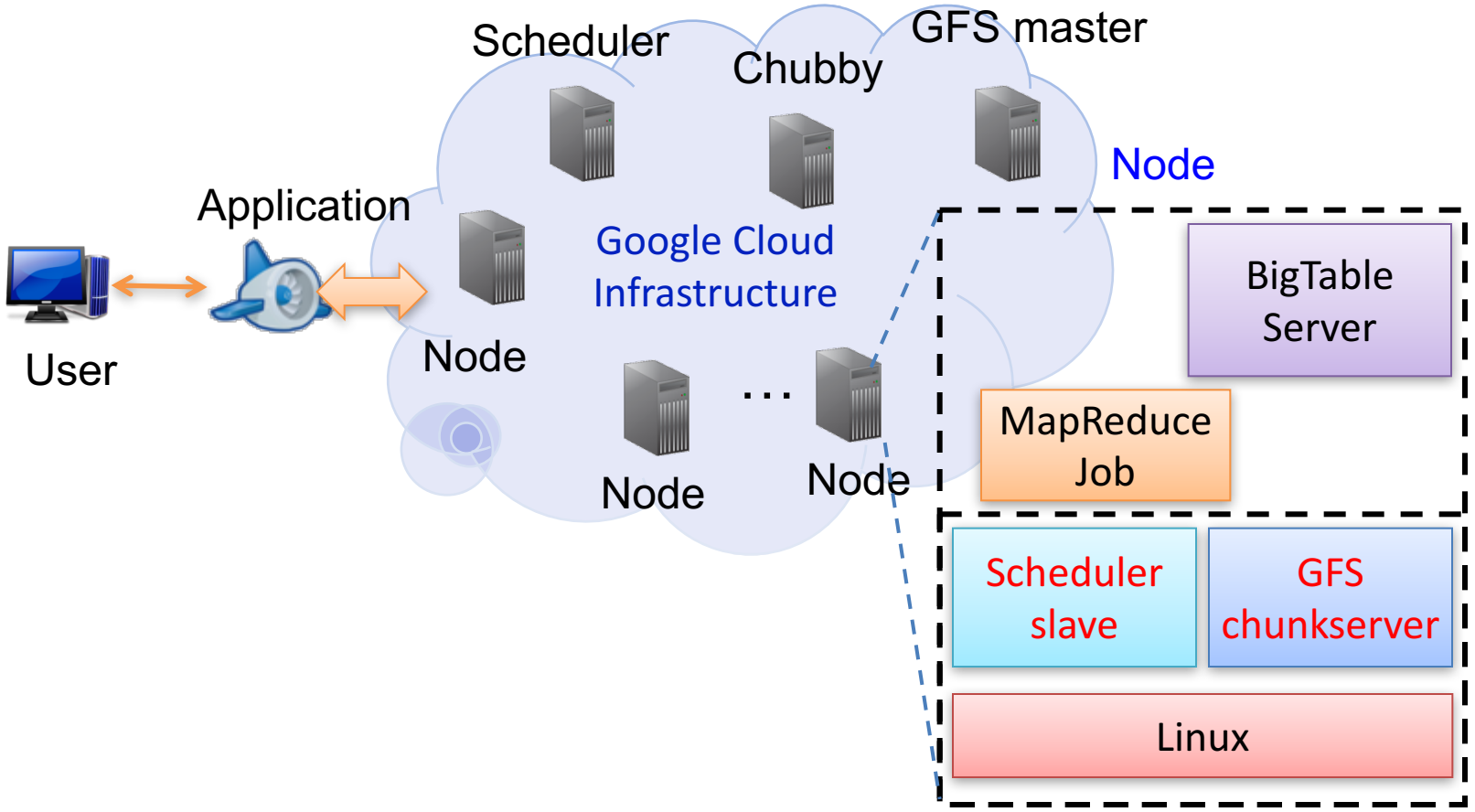
- More cost sensitive
  - Tend to use cheap computers whose MTBF may not as high as in HPC systems
  - Fault tolerance becomes more important
- Elastic demand
  - Can not fix the number of computers used
  - Adding/updating new machines all years long, heterogeneous nodes

# Challenges to Big data systems(cont'd)

- Production workload and analysis workload co-exist in the same cluster for better utilization
  - The required resources may not be fully available
  - May kill some tasks for jobs with higher priority

**Strong demands on fault tolerance and the support of heterogeneous nodes**

# Google Cloud Infrastructure




MapReduce: Simplified Data Processing on Large Clusters， Jeffrey Dean and Sanjay Ghemawat， OSDI 2004

The Google file system, Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung， SOSP 2003

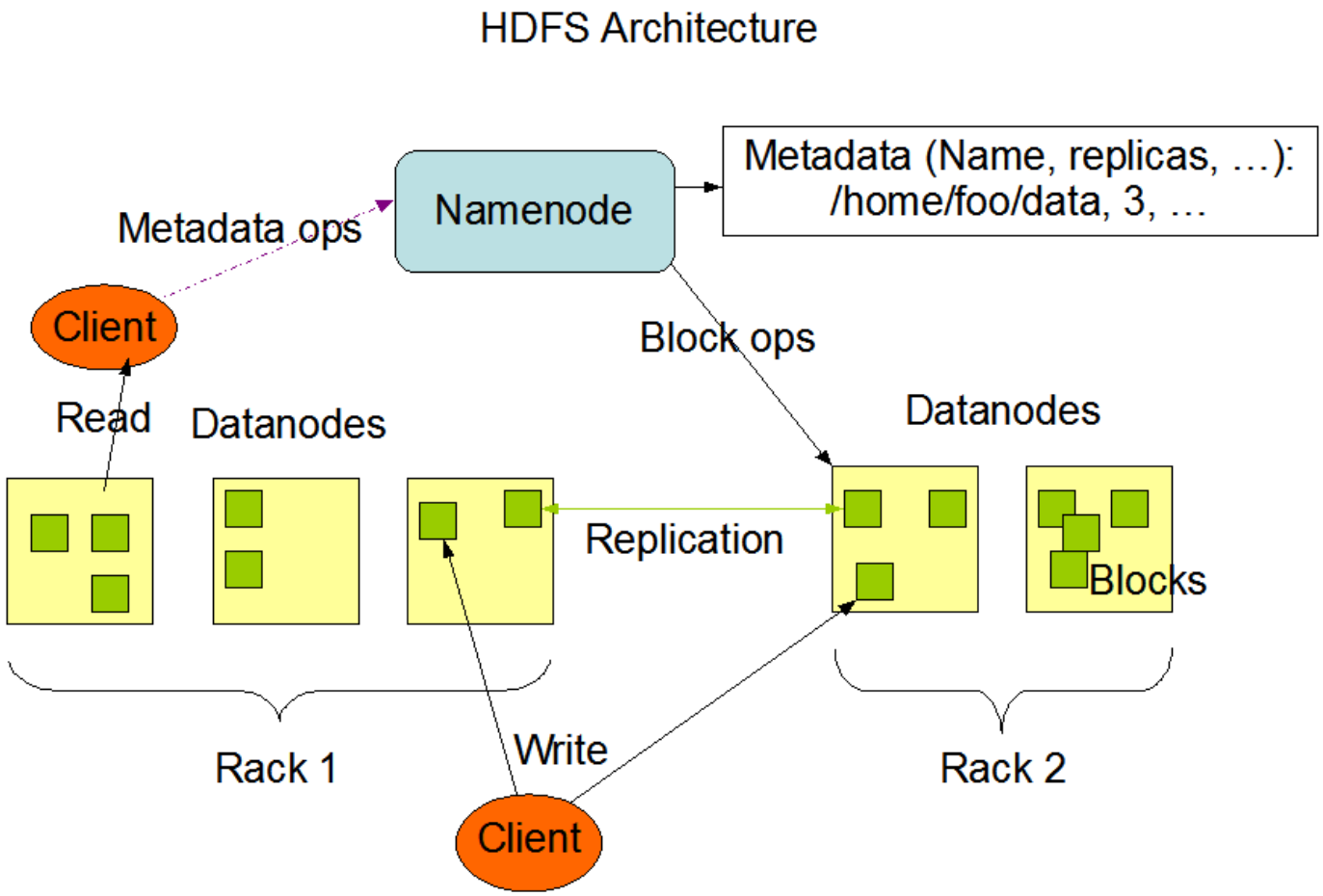
# A representative big data system infrastructure

- Merge compute and storage nodes
  - For better performance and more cost-effective
- Use GFS to support fault-tolerance for data
- Use MapReduce programming model
  - Support heterogeneous nodes
  - Support scheduling flexibility
  - Fault-tolerance on computing

# Open source big data infrastructure Hadoop

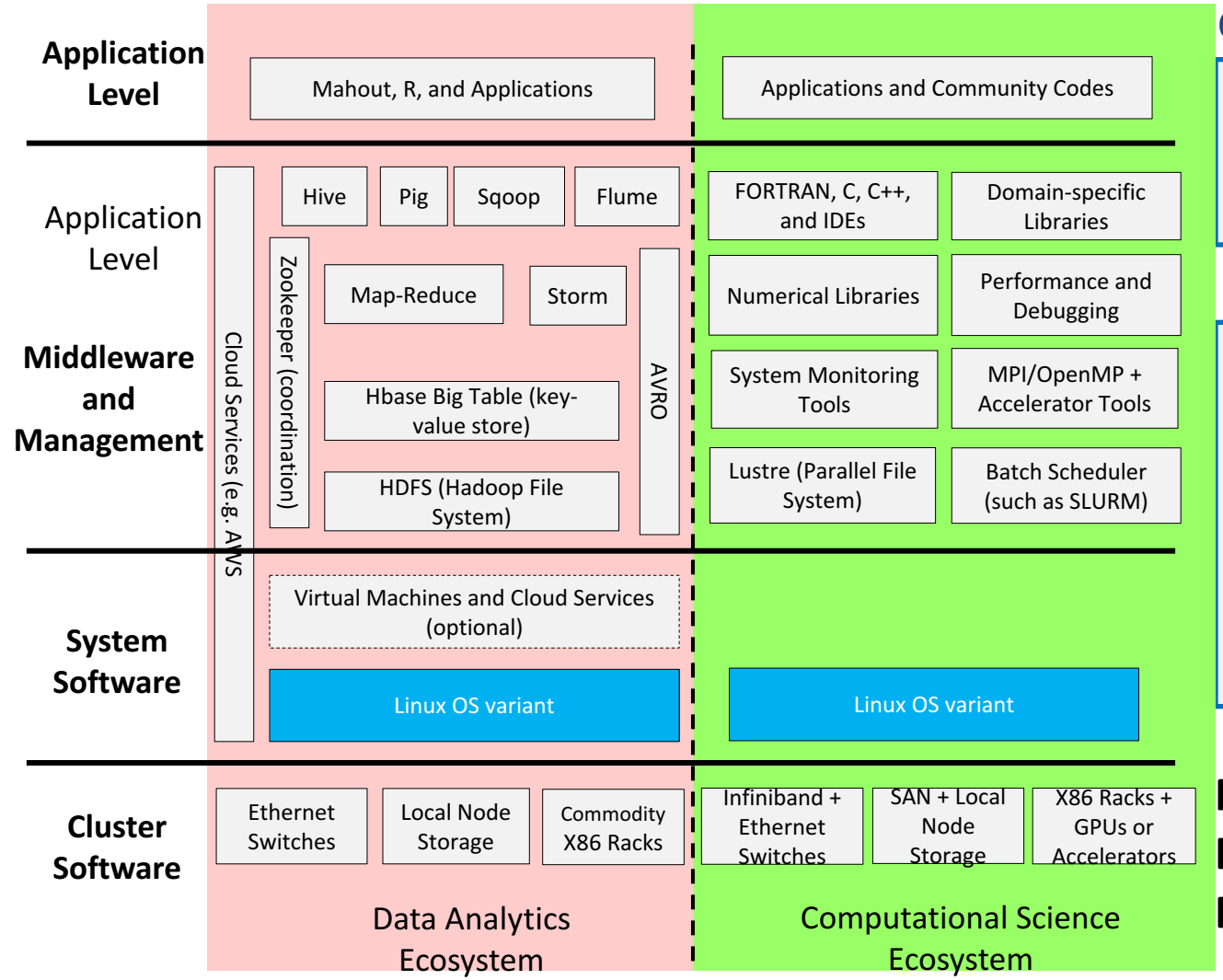
- GFS → HDFS
- MapReduce → A Java implementation
- Scheduler(Borg) → Yarn
- Chubby → ZooKeeper
- BigTable → HBase

# HDFS Architecture



# HDFS

- Use data replica to achieve high availability
  - Usually 3 copies are used (why?)
- Cost, performance and availability
  - Cost : -
  - Performance: +
  - Availability: +




## Computational Science

**FORTRAN,C,C++**: languages
**PAPI**: performance and debugging tool
**MPI/OpenMP**: multi-core parallel model
**SLURM**: batch scheduler
**Lustre**: parallel file system

## Data Analytic

**Mahout**: machine learning tool
**Hive**: data warehouse software
**Pig**: provide high level language for big data
**Sqoop**: exchange data with traditional database
**Flume**: log management
**Zookeeper**: maintaining consistency
**Storm**: real-time computation system.
**Hbase**: a distributed, scalable big data store.
**AVRO**: data serialization system.

**NOTE: The Divergence of Big Data and HPC Eco-Systems!**

Data analytics and computing ecosystem compared

**Inspired by "Exscale Computing and Big Data", DANIEL A. REED AND JACK DONGARRA , CACM July 2015**

# Why MPI is hard to schedule flexibly and tolerant fault?

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

tag=1234;source=0;destination=1;count=1;

 if(myid == source){
   buffer=5678;
   MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
   printf("processor %d  sent %d\n",myid,buffer);
}
if(myid == destination){
    MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
       printf("processor %d  got %d\n",myid,buffer);
    }

MPI_Finalize();
```

# Alternative set of 6 Functions for Simplified MPI

- **MPI_INIT**
- **MPI_FINALIZE**
- **MPI_COMM_SIZE**
- **MPI_COMM_RANK**
- **MPI_BCAST**
- **MPI_REDUCE**

# Revisit the MPI with collective communications

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.14159265358979323846264 3;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```

```
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is %.16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

# MPI model

- What are the obstacles of supporting heterogeneous node and fault tolerance
  - Knowing N at the beginning
  - Static partition of the workload

Any idea to solve the problem?

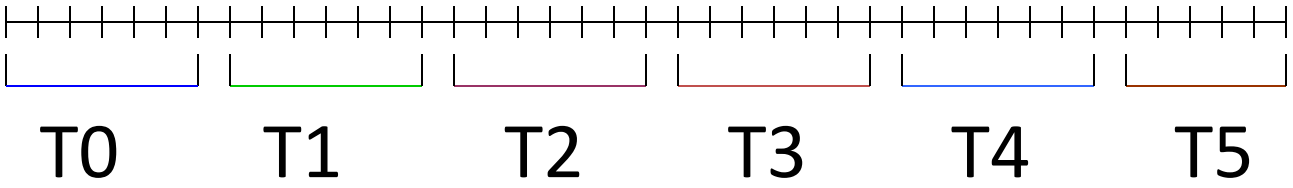
# The OpenMP version

```
x=0;
sum = 0.0;
step = 1.0/(double) num_steps;

#pragma omp for reduction(+:sum) private(x)
for (i=0; i < num_steps; i=i+1){
      x=(i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
pi=step*sum;
```

This is similar to the MPI version we have

# Revisit Static Scheduling

For $N_i$ iterations and $N_t$ threads, each thread gets one chunk of $N_i/N_t$ loop iterations:




- Thread #0: iterations 0 through $N_i/N_t$-1
- Thread #1: iterations $N_i/N_t$ through $2N_i/N_t$-1
- Thread #2: iterations $2N_i/N_t$ through $3N_i/N_t$-1

…

- Thread #$N_t$-1: iterations $(N_t\text{-}1)N_i/N_t$ through $N_i$-1

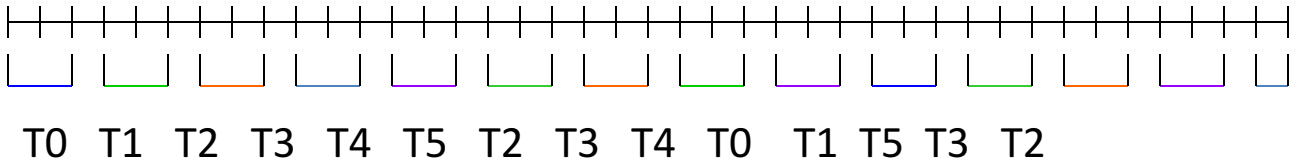
# The OpenMP version - Dynamic

```
x=0;
sum = 0.0;
step = 1.0/(double) num_steps;

#pragma omp for reduction(+:sum) private(x)
schedule(dynamic,1)
for (i=0; i < num_steps; i=i+1){
     x=(i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
pi=step*sum;
```

What are needed in MPI if we wish each process do part of the calculation?

# Revisit Dynamic Scheduling

For $N_i$ iterations and $N_t$ threads, each thread gets a fixed-size chunk of $k$ loop iterations:




When a particular thread finishes its chunk of iterations, it gets assigned a new chunk. So, the relationship between iterations and threads is nondeterministic.

- Advantage: very flexible
- Disadvantage: high overhead – lots of decision making about which thread gets each chunk

# Dynamic scheduling on distributed systems

- Doing a small piece of work, complete it and get the next piece
  - Input from the global file system, similar to the MPI_Bcast
- No N is required to know at the beginning
- Need a reduce support similar to MPI_Reduce

# The MapReduce Programming Model

- Borrows from functional programming
- Users implement interface of two functions

  - `map (in_key, in_value) -> (out_key, intermediate_value) list`

  - `reduce (out_key, intermediate_value list) -> out_value list`

# Map

- Records from the data source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line).
- map() produces one or more *intermediate* values along with an output key from the input.

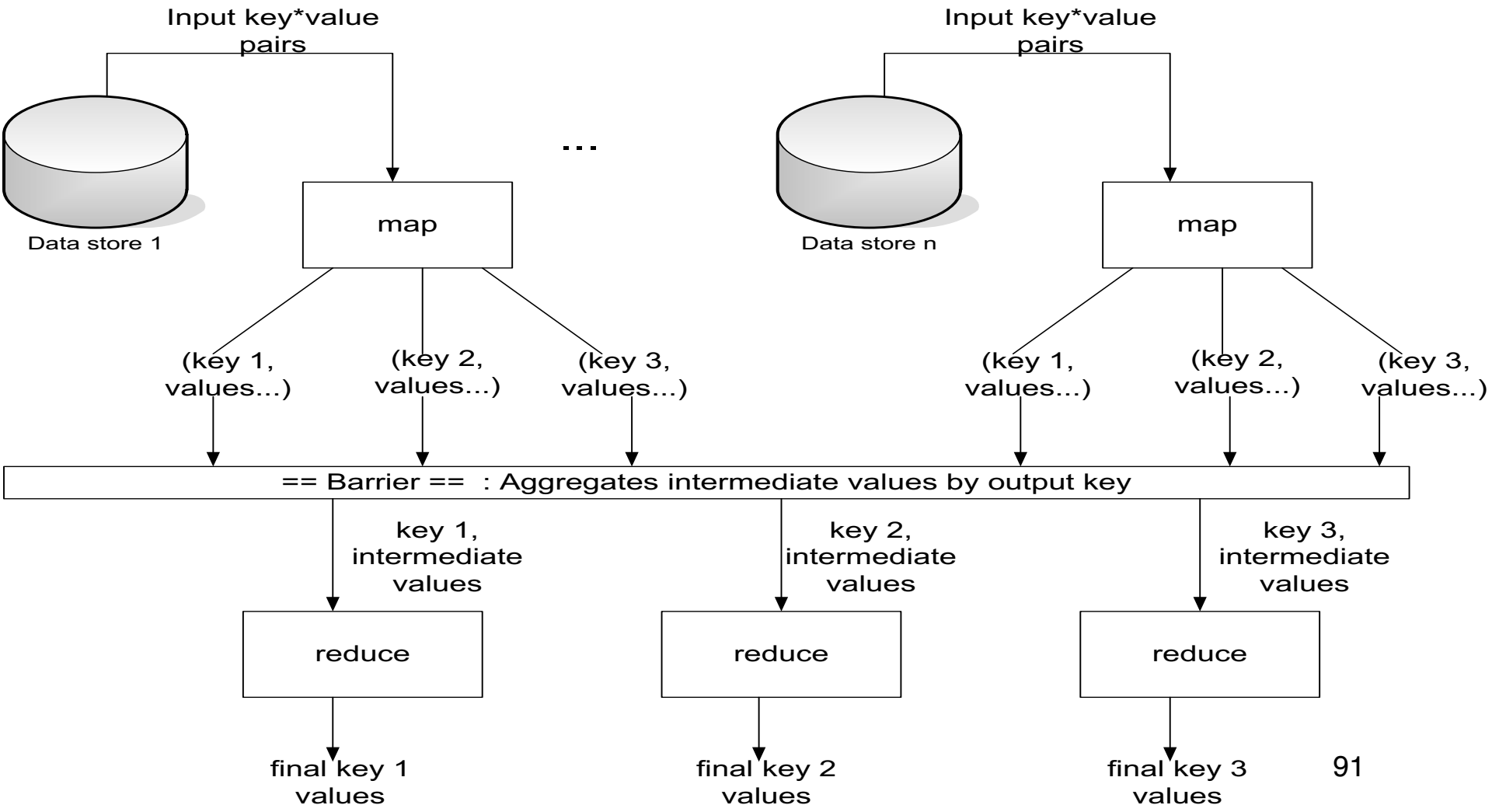
# The concept of Key-value pair

- Name – Value
  - int c = 5  ; c is the key, value is 5
- <c,5>,<b,0>, <a,2>,<c,6>,<c,7>
- What is the average of c?
- Use c to retrieve data from the list

# Reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)

# Architecture



# Parallelism

- map() functions run in parallel, creating different intermediate values from different input data sets
- reduce() functions also run in parallel, each working on a different output key
- All values are processed *independently*
- Bottleneck: reduce phase can't start until map phase is completely finished.

# Example: Count word occurrences

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
 Emit(AsString(result));
```

# Example vs. Actual Source Code

- Example is written in pseudo-code
- Actual implementation is in C++ or Java, using a MapReduce library
- True code is somewhat more involved (defines how the input key/values are divided up and accessed, etc.)

# Example

- Page 1: the weather is good
- Page 2: today is good
- Page 3: good weather is good.

# Map output

- Worker 1:
  - (the 1), (weather 1), (is 1), (good 1).
- Worker 2:
  - (today 1), (is 1), (good 1).
- Worker 3:
  - (good 1), (weather 1), (is 1), (good 1).

# Reduce Input

- Worker 1:
  - (the 1)
- Worker 2:
  - (is 1), (is 1), (is 1)
- Worker 3:
  - (weather 1), (weather 1)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 1), (good 1), (good 1), (good 1)

# Reduce Output

- Worker 1:
  - (the 1)
- Worker 2:
  - (is 3)
- Worker 3:
  - (weather 2)
- Worker 4:
  - (today 1)
- Worker 5:
  - (good 4)

# Map-Reduce Examples

# Problem 1 Distributed Grep

- Input
  - A key word  and documents to be searched
- Output
  - The line contains the keyword, with file name

```
[cwg@HPC-AMD phoenix_scheduler]$ grep main *
MapReduceScheduler.h:/* The main MapReduce engine. This is the function called by the application.
MapReduceScheduler.h: * also organizes and maintains the data which is passed from application to
```

# MR Grep

- Map
  - Emits the result of local grep
- Reduce
  - Output what it get from Map

# Problem 2 Inverted Index

- Input
  - A few text documents
- Output
  - (word, docIDList)

# MR-II

- Map
  - Parse document and emit (word, DocID)
- Reduce
  - Emit ( word, DocIDList )

# Iterative MapReduce: K-Means as an Example

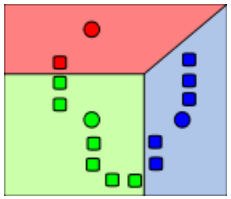
# K-Means Algorithm

- Input:
  - a set of P points in N-dimension space
- Output:
  - distribute them into K clusters, so that points closer to each other is in the same cluster
- This is an NP-hard problem
  - Even when K=2 or D=2

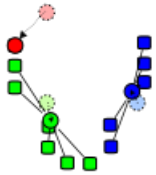
# K-Means Algorithm (cont.)

- randomly generate K points as the means
- do
  - for each point in the point set
    - find the closest mean I, assign the point to cluster I, forming K clusters
  - calculate new means for each cluster
- until no changes are made to the clusters

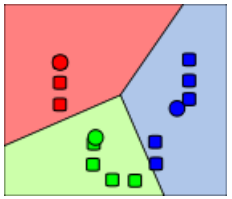
1) *k* initial "means" (in this case *k*=3) are randomly selected from the data set (shown in color).



2) *k* clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means.



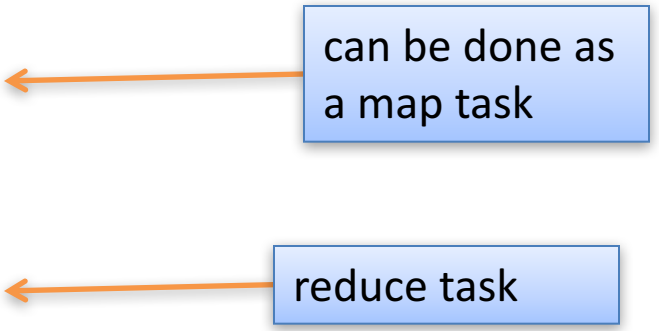
3) The centroid of each of the *k* clusters becomes the new means.



4) Steps 2 and 3 are repeated until convergence has been reached.

# Expressing an Iteration in MapReduce

- for each point in the point set
  - find the closest mean l, assign the point to cluster l, forming new K clusters
- calculate new means for each cluster




```
map(point p):
        for each mean in K means
                find the closet mean M to this point
        emit_intermediate (index of M, p)

reduce(index of cluster i, list of points lp):
        calculate the new mean M' for points in lp
        emit(M')
```

# But it only completes one iteration

- Can we express the whole K-Means algorithms with Map-Reduce
  - Yes, but we have to extend the basic Map-Reduce model
  - Connect output of a Map-Reduce iteration to the input of the next iteration
  - That what we called Iterative Map-Reduce

# MapReduce Conclusions

- MapReduce has proven to be a useful abstraction
  - Greatly simplifies large-scale computations at Google
  - Functional programming paradigm can be applied to large-scale applications
  - Fun to use: focus on problem, let library deal w/ messy details
    - Automatic fault tolerance and load balancing

# Thanks