

CSE 262

Lecture 12

Communication overlap

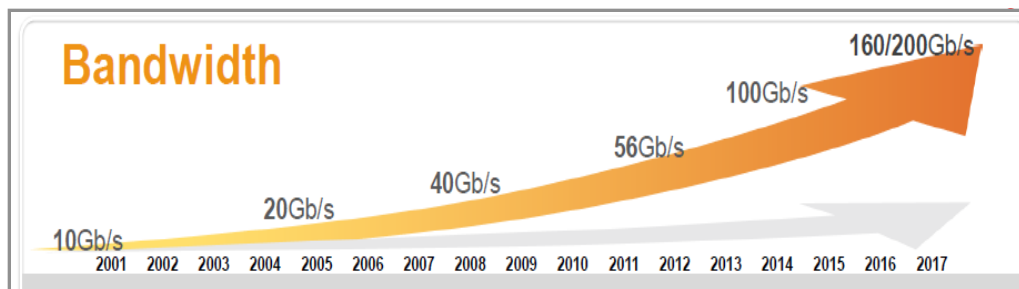
Announcements

- A problem set has been posted, due next Thursday

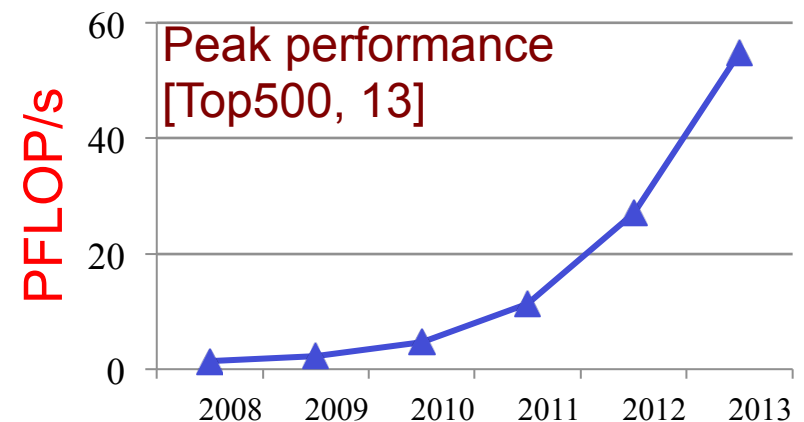
Technological trends of scalable HPC systems

- Growth: cores/socket rather than sockets
- Hybrid processors
- Complicated software-managed parallel memory hierarchy
- Memory/core is shrinking
- **Communication costs increasing relative to computation**

2X/ 3-4 years

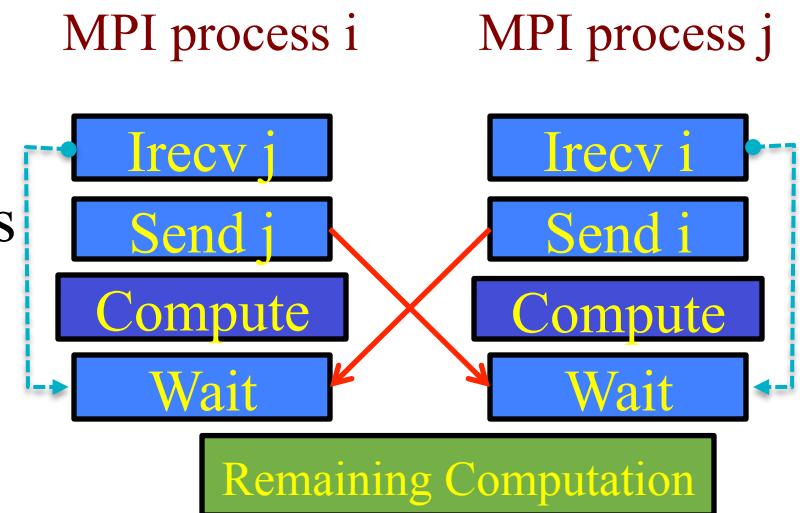


2X/year



Reducing communication costs in scalable applications

- **Tolerate** or avoid them [Demmel et al.]
- Difficult to reformulate MPI apps to overlap communication with computation
 - ◆ Enables but does support communication hiding
 - ◆ Split phase coding
 - ◆ Scheduling
- Implementation policies become entangled with correctness
 - ◆ Non-robust performance
 - ◆ High software development costs

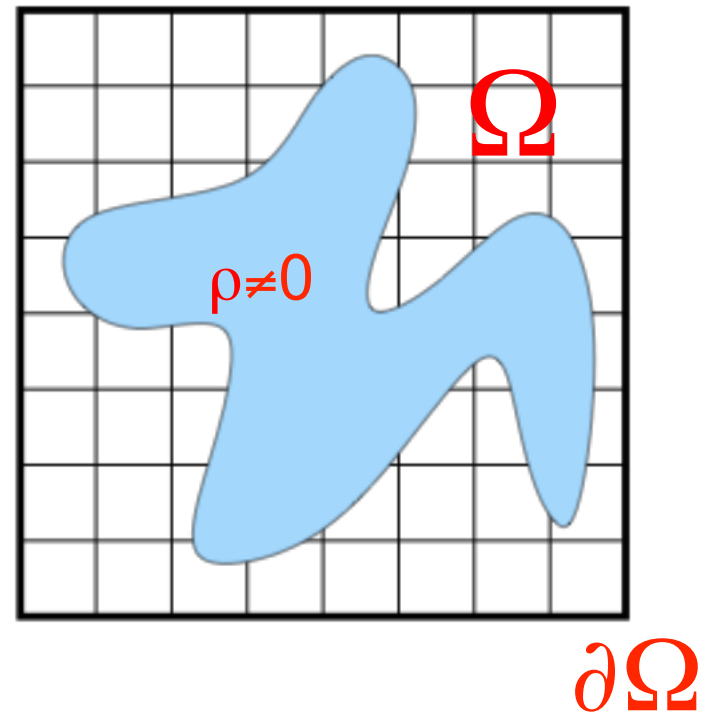


Motivating application

- Solve Laplace's equation in 3 dimensions with Dirichlet Boundary conditions

$$\Delta\varphi = \rho(x,y,z), \quad \varphi=0 \text{ on } \partial\Omega$$

- Building block: iterative solver using Jacobi's method (7-point stencil)

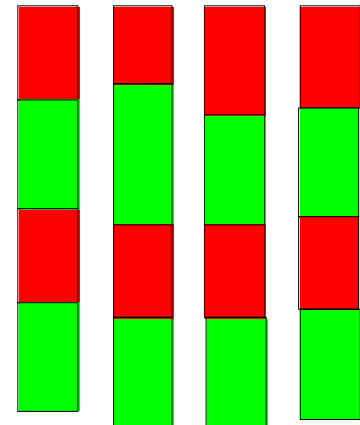
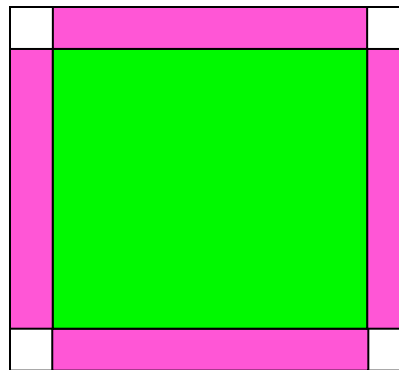
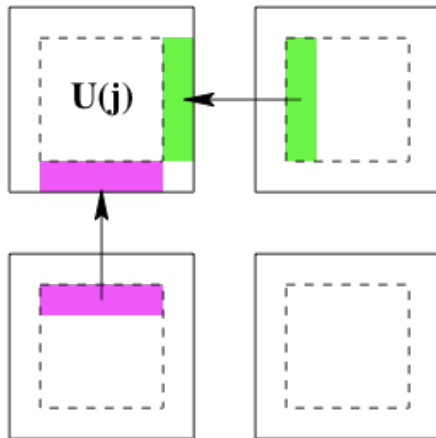


for (i,j,k) in 1:N x 1:N x 1:N

$$u'[i][j][k] = (u[i-1][j][k] + u[i+1][j][k] + \\ u[i][j-1][k] + u[i][j+1][k] + \\ u[i][j][k+1] + u[i][j][k-1]) / 6.0$$

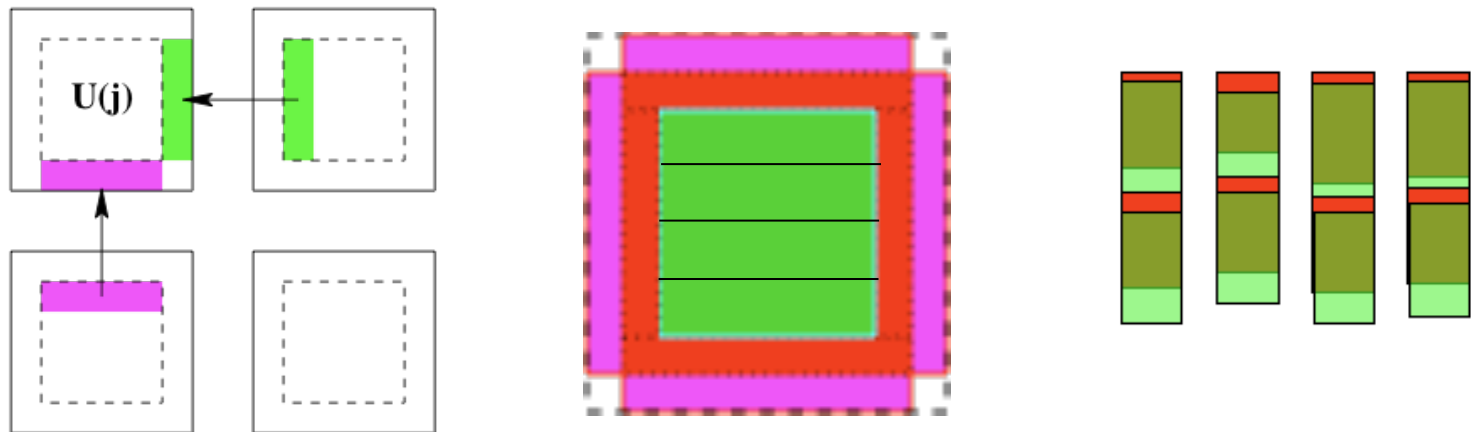
Classic message passing implementation

- Decompose domain into sub-regions, one per process
 - ♦ Transmit **halo regions** between processes
 - ♦ Compute **inner region** after communication completes
- Loop carried dependences impose a strict ordering on communication and computation



Latency tolerant variant

- Only a subset of the domain exhibits **loop carried dependences** with respect to the **halo region**
- Subdivide the domain to remove some of the dependences
- We may now sweep the **inner region** in parallel with **communication**
- Sweep the **annulus** after communication finishes



MPI Encoding

MPI_Init(); MPI_Comm_rank(); MPI_Comm_size();

Data initialization

MPI_Send/MPI_Isend

MPI_Recv/MPI_Irecv

Computations

MPI_Finalize();



A few implementation details

- Some installations of MPI cannot realize overlap with MPI_IRecv and MPI_Isend
- We can use multithreading to handle the overlap
- We let one or more processors (proxy thread(s)) handle communication

S. Fink, PhD thesis, UCSD, 1998

Baden and Fink, “Communication overlap in multi-tier parallel algorithms,” SC98

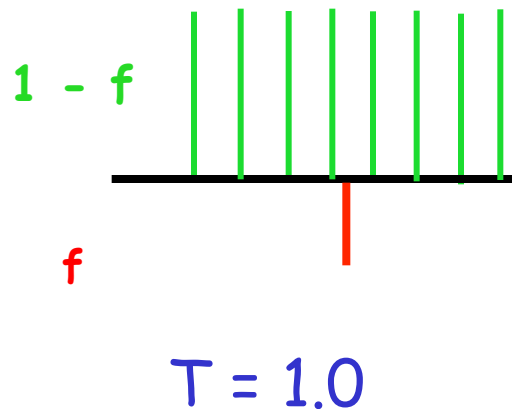
A performance model of overlap

- Assumptions

p = number of processors per node

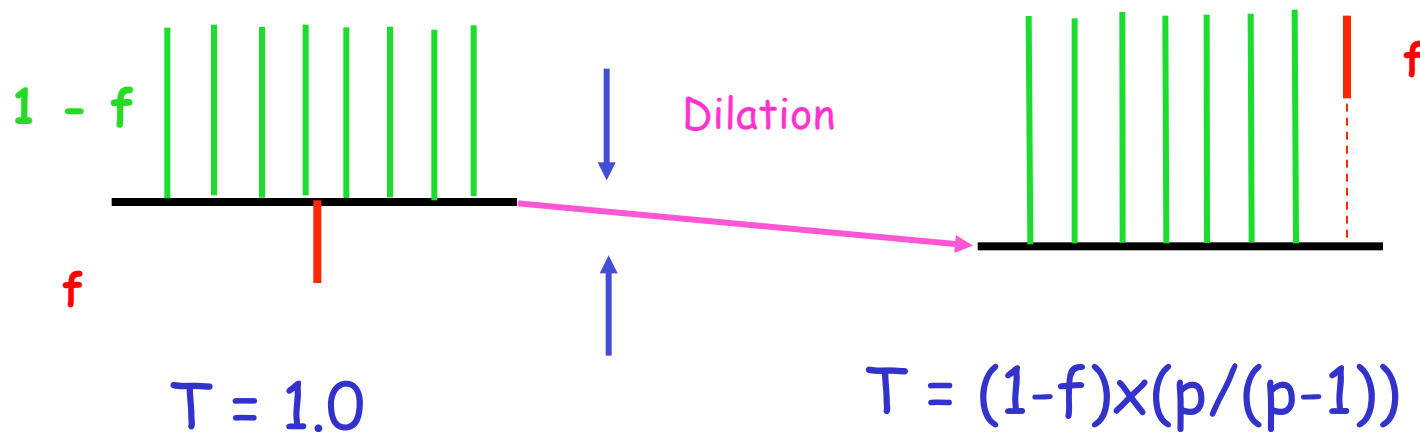
running time = 1.0

$f < 1$ = communication time
(i.e. not overlapped)



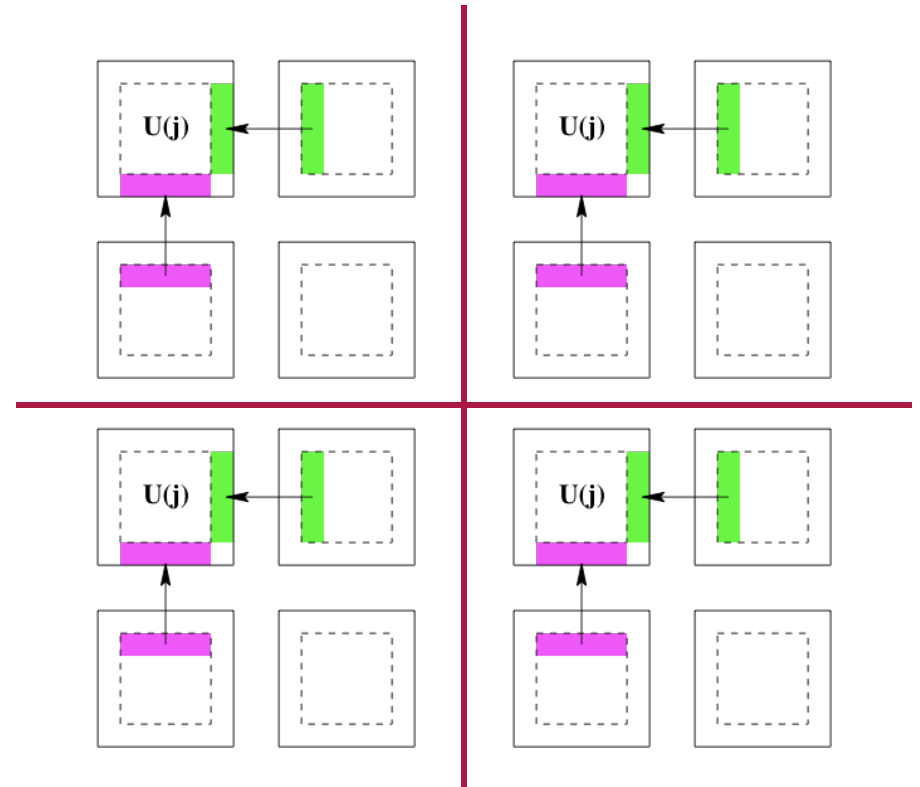
Performance

- When we displace computation to make way for the proxy, computation time *increases*
- Wait on communication drops to zero, ideally
- When $f < p/(2p-1)$: improvement is $(1-f) \times (p/(p-1))^{-1}$
- Communication bound: improvement is $1/(1-f)$



Processor Virtualization

- Virtualize the processors by overdecomposing
- AMPI [Kalé et al.]
- When an MPI call blocks, thread yields to another virtual process
- How do we inform the scheduler about ready tasks?

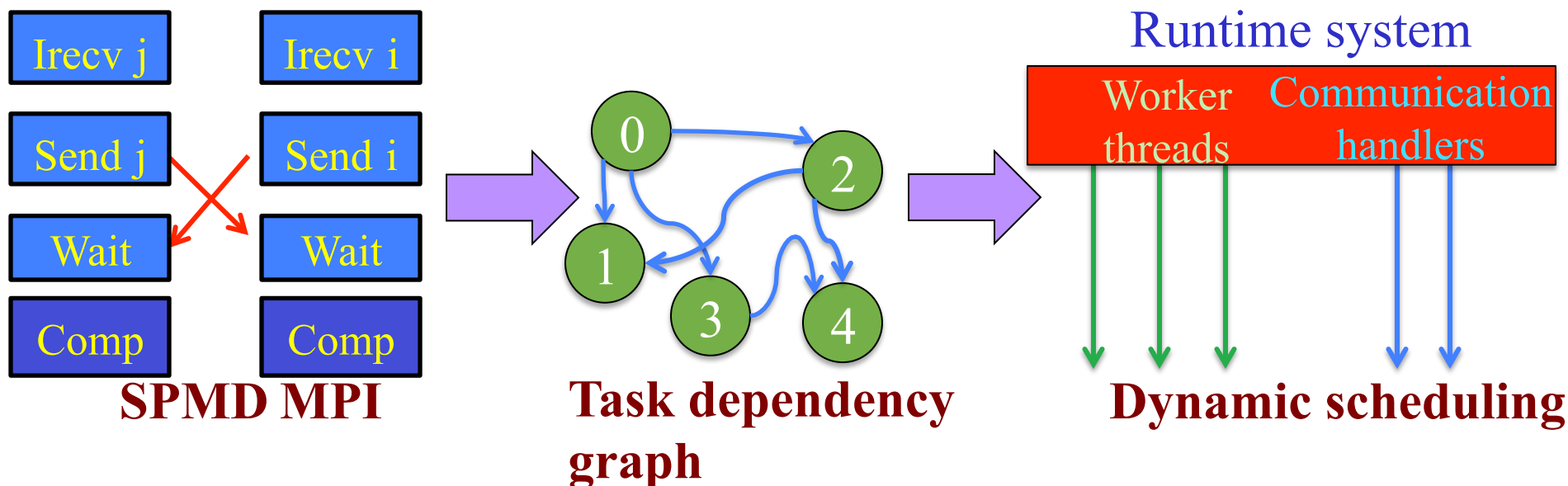


Observations

- The exact execution order depends on the data dependence structure: communication & computation
- We don't have to hard code a particular overlap strategy
- We can alter the behavior by changing the data dependences, e.g. disable overlap, or by varying the on-node decomposition geometry
- For other algorithms we can add priorities to force a preferred ordering
- Applies to many scales of granularity (i.e. memory locality, network, etc.)

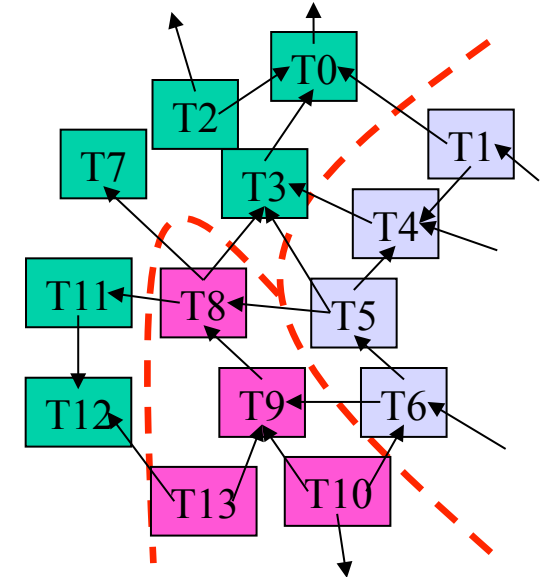
An alternative way to hide communication

- Reformulate MPI code into a data-driven form
 - ◆ Decouple scheduling and communication handling from the application
 - ◆ Automatically overlap communication with computation



Tarragon - Non-SPMD, Graph Driven Execution

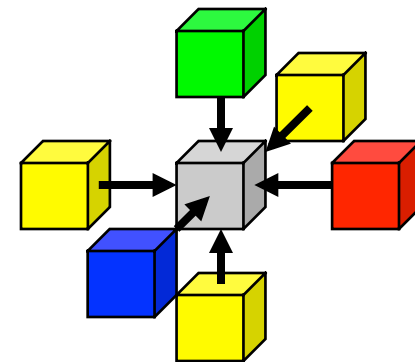
- Pietro Cicotti [Ph.D., 2011]
- Automatically tolerate communication delays via a **Task Precedence Graph**
 - ♦ Vertices = computation
 - ♦ Edges = dependences
- Inspired by Dataflow and Actors
 - ♦ Parallelism ~ independent tasks
 - ♦ Task completion $\Leftrightarrow$ Data Motion
- Asynchronous task graph model of execution
 - ♦ Tasks run according to availability of the data
 - ♦ Graph execution semantics independent of the schedule



Task Graph

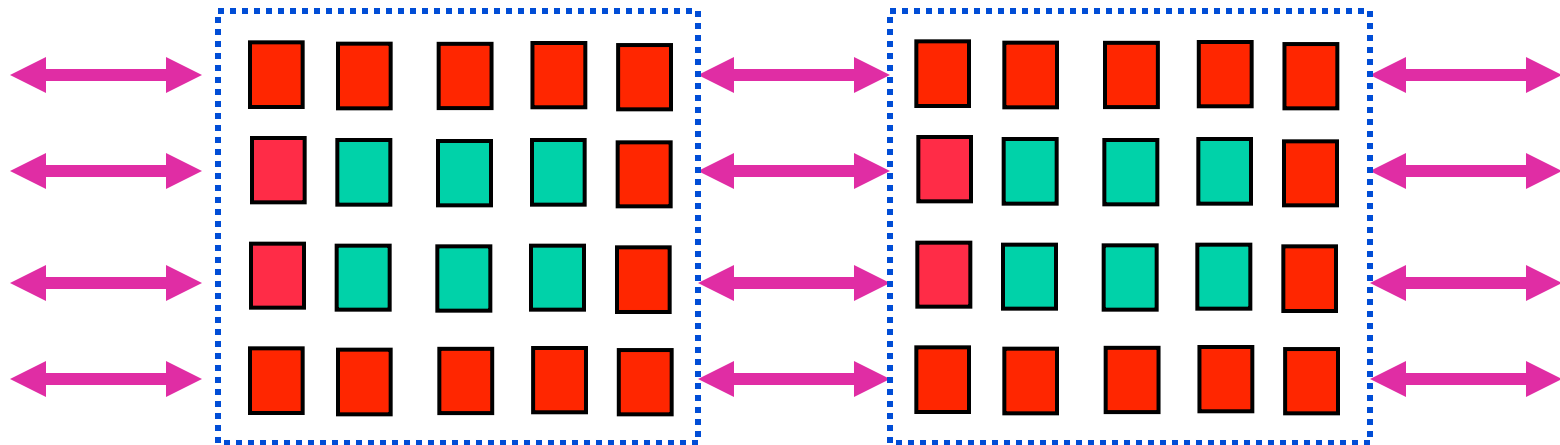
- Represents the program as a task precedence graph encoding **data dependences**
- Background run time services support dataflow execution of the graph
- Virtualized tasks: many to each processor
- The graph maintains *meta-data* to inform the scheduler about runnable tasks

```
for (i,j,k) in 1:N x 1:N x 1:N  
  u[i][j][k] = .....
```



Graph execution semantics

- Parallelism exists among independent tasks
- Independent tasks may execute concurrently
- A task is **runnable** when its data dependences have been met
- A task **suspends** if its **data dependences** are not met
- Computation and data motion are coupled activities
- Background services manage graph execution
- The **scheduler** determines which task(s) to run next
- Scheduler and application are only vaguely aware of one another
- Scheduler doesn't affect graph execution semantics

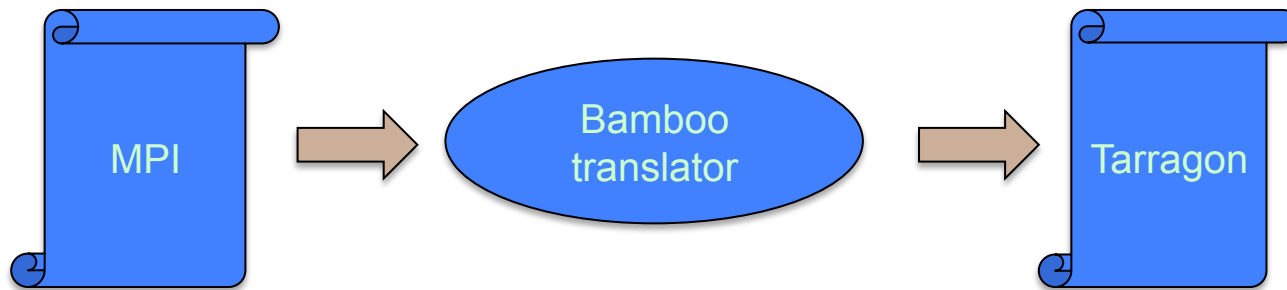


Code reformulatoin

- Reformulating code by hand is difficult
- Observation: For every MPI program there is a corresponding dataflow graph ...
... determined by the matching patterns of sends and receives invoked by the running program
- Can we come up with a deterministic procedure for translating MPI code to Tarragon, using dynamic and static information about MPI call sites?
- Yes! Bamboo: custom, domain specific-translator
Tan Nguyen (PhD, 2014, UCSD)

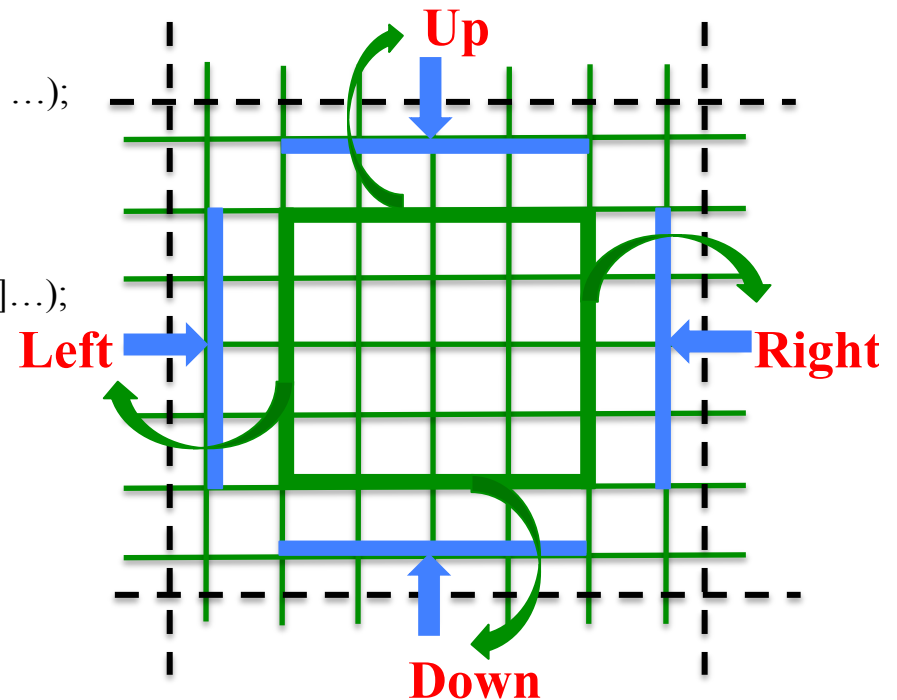
Bamboo

- Uses a deterministic procedure for translating MPI code to Tarragon, using dynamic and static information about MPI call sites
- A custom, domain-specific translator
 - ◆ MPI library primitives → primitive language objects
 - ◆ Collects static information about MPI call sites
 - ◆ Relies on some programmer annotation
 - ◆ Targets Tarragon library, supports scheduling and data motion [Pietro Cicotti '06, '11]



Example: MPI with annotations

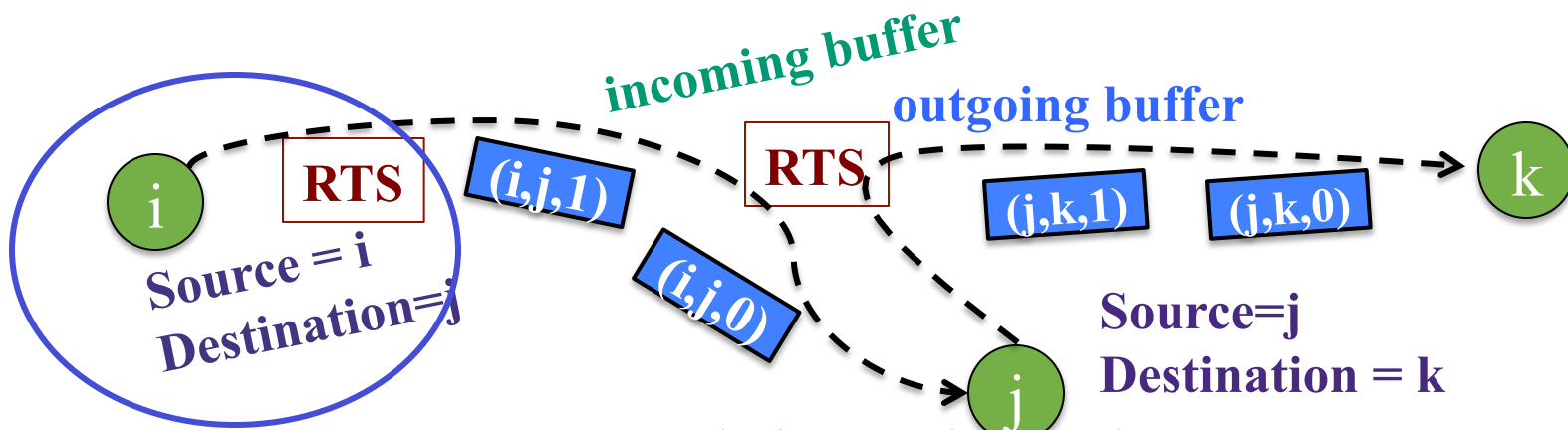
```
1 for (iter=0; iter<maxIters && Error > ε; iter++)
2 {
3   #pragma bamboo olap
4   {
5     #pragma bamboo receive
6     { for each dim in 4 dimensions
7       if(hasNeighbor[dim]) MPI_Irecv(..., neighbor[dim], ...);
8     }
9     #pragma bamboo send
10    { for each dim in 4 dimensions
11      if(hasNeighbor[dim]) MPI_Send( ..., neighbor[dim]...);
12      MPI_Waitall(...);
13    }
14  }
15  update (Uold, Un); swap (Uold, Un); lerror = Err(Uold);
16 }
17 MPI_Allreduce(lError, Error); //translated automatically
18 }
```



Send and Receive are
independent

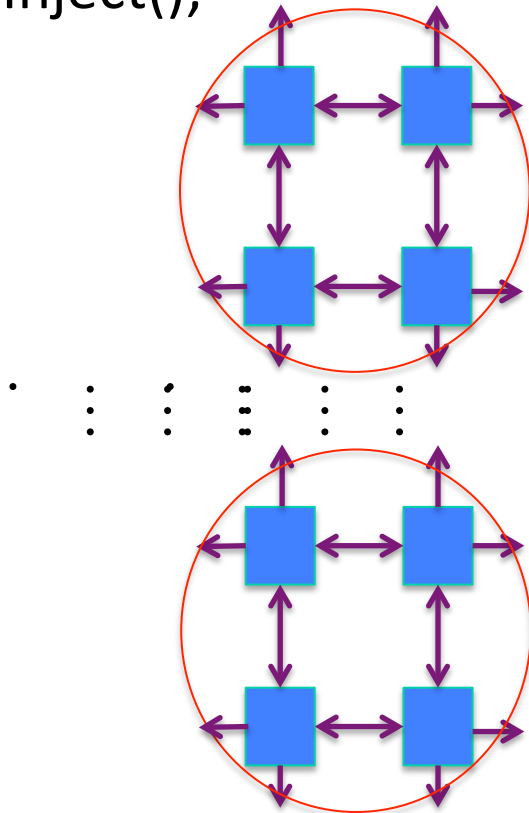
Task definition and communication

- Bamboo instantiates MPI processes as tasks
 - ◆ User-defined threads + user level scheduling (not OS thread)
 - ◆ Tasks communicate via messages, which are not imperative
- Mapping processes $\rightarrow$ tasks
 - ◆ Send $\rightarrow$ put (), RTS handles delivery
 - ◆ Recvs $\rightarrow$ firing rule: task is ready to run when input conditions are met, firing rule processing handled by RTS
 - ◆ No explicit receives; when a task is runnable, its input conditions have been met by definition



Translated code

```
class task {
  void Init();
  void Execute();
  void Inject();
};
```



Init:

```
pack ghostcells
  put(ghostcells, left/right/up/down edges)
  _state = WAIT;
```

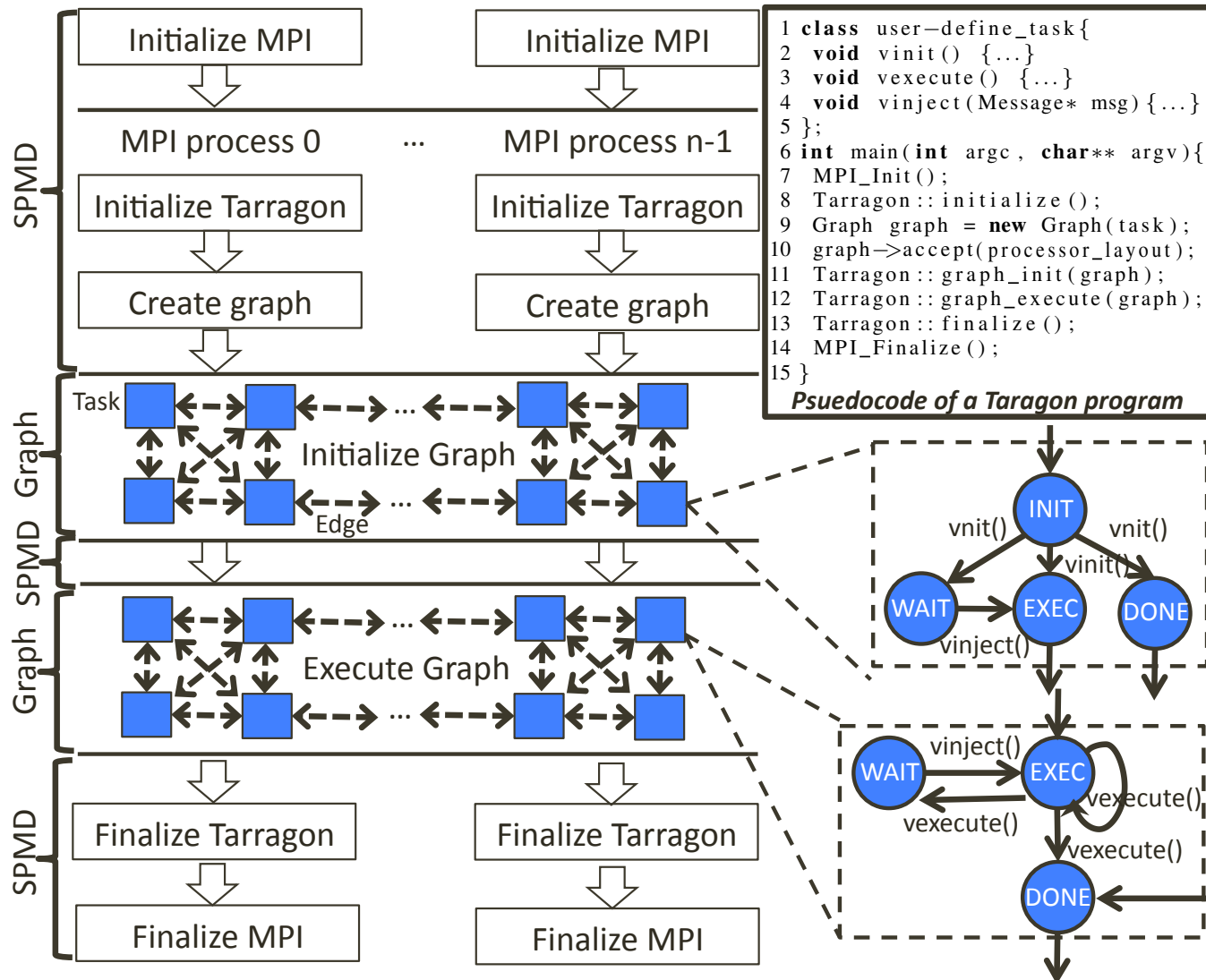
Inject:

```
if(received from left & right & up & down)
  _state = EXEC;
```

Exec:

```
If (it <= number_of_iterations)
  unpack and update ghostcells
  for(j =1; j< localN-1; j++)
    for(i =1; i< localN-1; i++){
      V(j,i)=c*(U(j-1,i)+U(j+1,i)+U(j,i-1)+U(j,i+1))
      swap(U,V);
    }
  pack ghostcells
  put(ghostcells, left/right/up/down edges);
  _state = WAIT;
  it++;
}else _state = DONE;
```

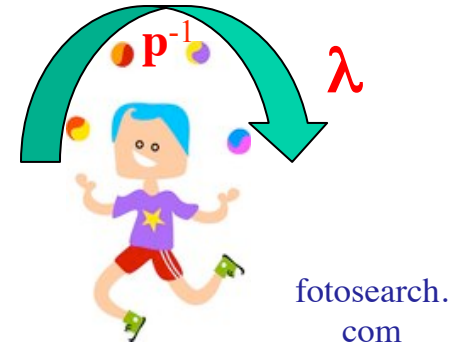
Control flow



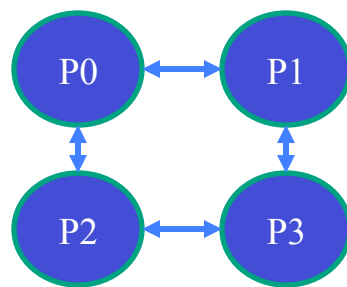
Virtualization

- Recall Little's law:

$$\text{Concurrency} = \text{Latency} \times \text{Effective throughput}$$

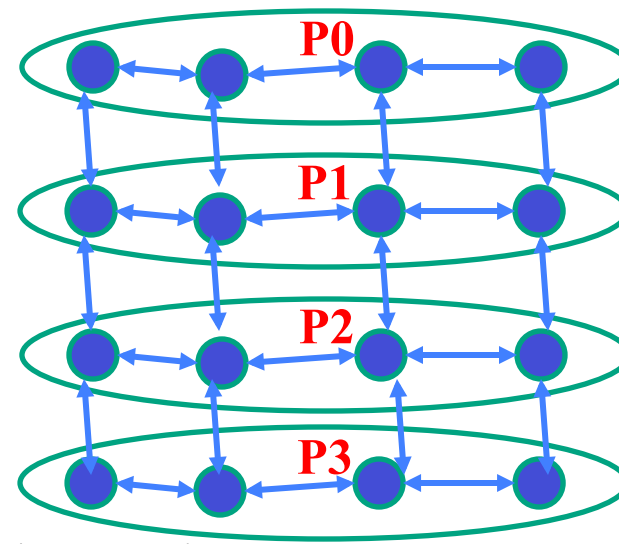


- Latency hiding requires more parallelism (and memory)
- Virtualization: multiple tasks per core
- AMPI [Huang et al., '03] and FGMPi [Kamal, '13]



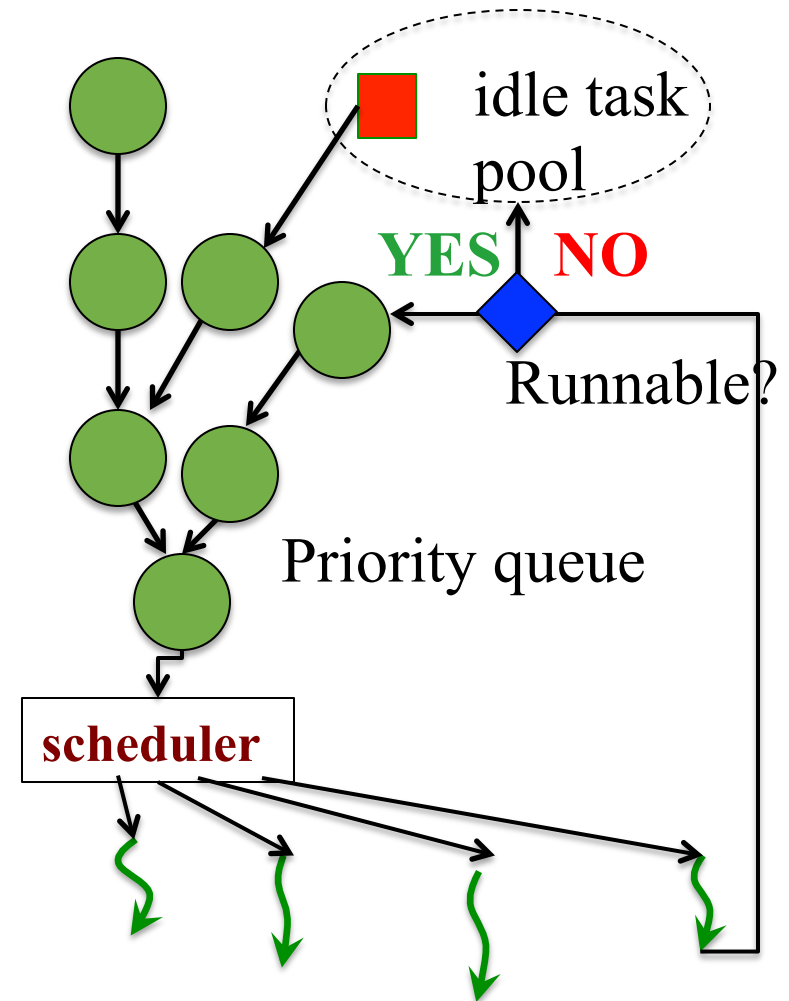
MPI

virtualize



Task Scheduling

- Data-driven execution model
 - ♦ A task is executable when its input is available
 - ♦ Tasks can't hold control on waiting
- Non-preemptive prioritization model
 - ♦ The scheduler can't stop a task
 - ♦ Among runnable tasks, those with higher priorities executed first
 - ♦ Task may volunteer to yield control



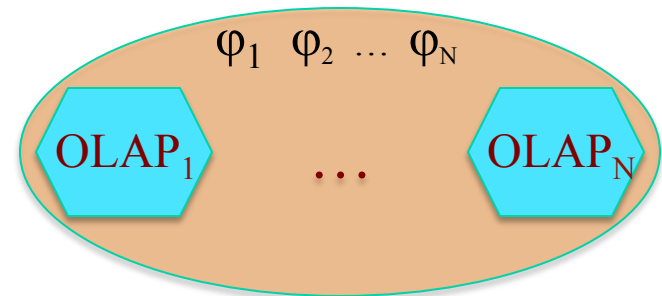
Bamboo Programming Model

- ***Olap-regions:*** task switching point
 - ♦ Data availability is checked at entry
 - ♦ Only 1 olap may be active at a time
 - ♦ When a task is ready, some olap region's input conditions have been satisfied
- ***Send blocks***
 - ♦ Hold send calls only
 - ♦ Enable the olap-region
- ***Receive blocks:***
 - ♦ Hold receive and/or send calls
 - ♦ Receive calls are input to olap-region
 - ♦ Send calls are output to an olap-region
- Activities in send blocks must be independent of those in receive blocks
- MPI_Wait/MPI_Waitall can reside anywhere within the olap-region

```
1  #pragma bamboo olap
2  {
3      #pragma bamboo send
4      {...}
5      #pragma bamboo receive
6      {...}
7  }
```

...
Computation
....

```
10 #pragma bamboo olap
11 {...}
....
```

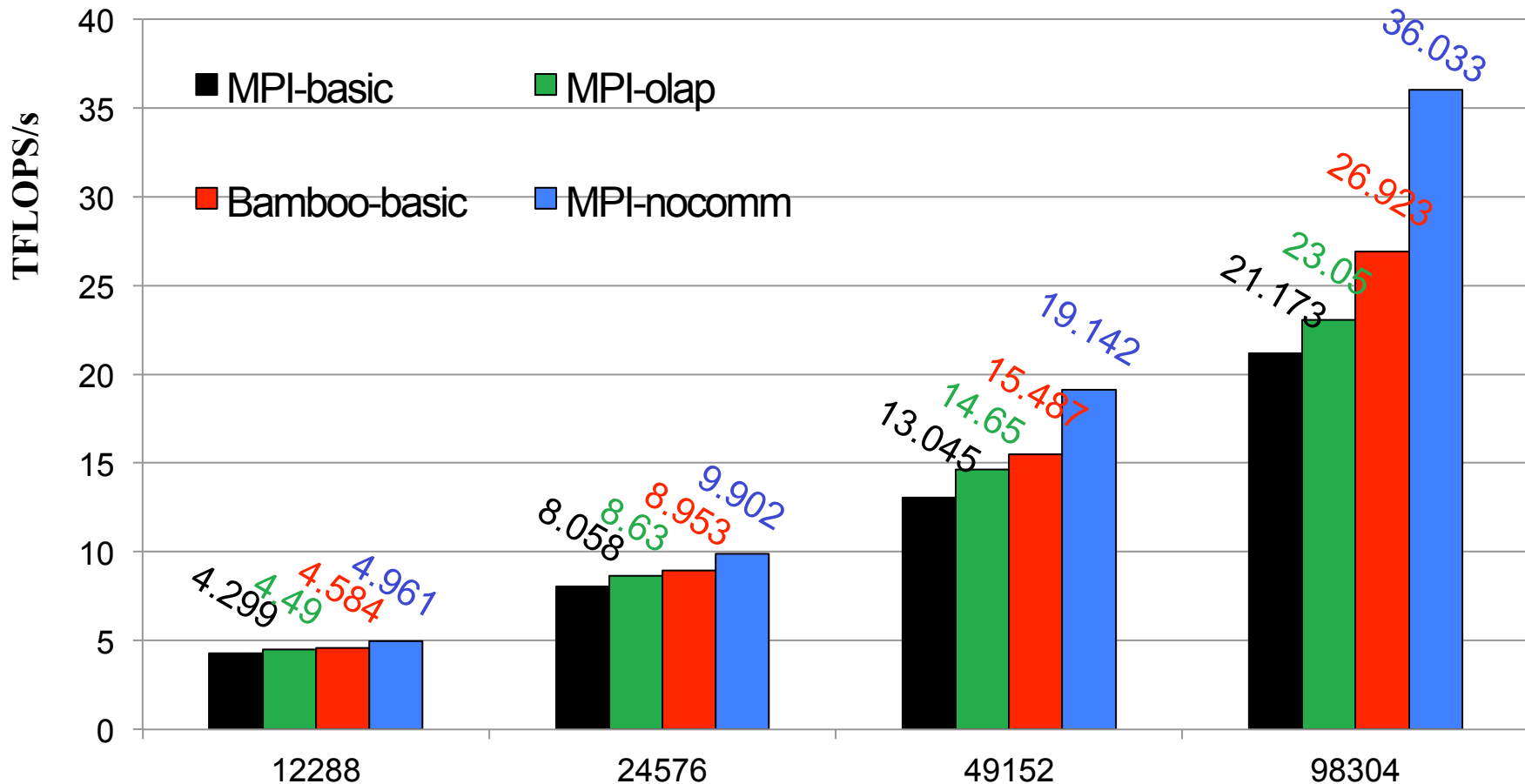


Results

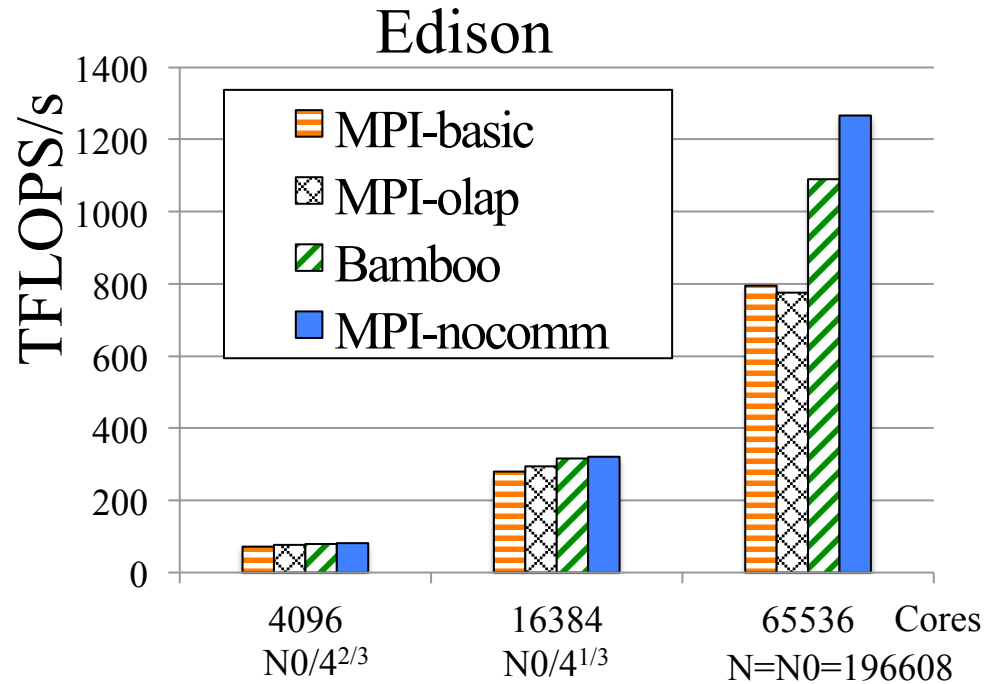
- Stampede at TACC
 - ♦ 102,400 cores; dual socket Sandy Bridge processors
 - ♦ K20 GPUs
- Cray XE-6 at NERSC (Hopper)
 - ♦ 153,216 cores; dual socket 12-core Magny Cours
 - ♦ 4 NUMA nodes per Hopper node, each with 6 cores
 - ♦ 3D Toroidal Network
- Cray XC30 at NERSC (Edison)
 - ♦ 133,824 cores; dual socket 12-core Ivy Bridge
 - ♦ Dragonfly Network

Stencil application performance (Hopper)

- Solve 3D Laplace equation, Dirichlet BCs ($N=3072^3$)
7-point stencil $\Delta u = 0$, $u=f$ on $\partial\Omega$
- Added 4 Bamboo pragmas to a 419 line MPI code



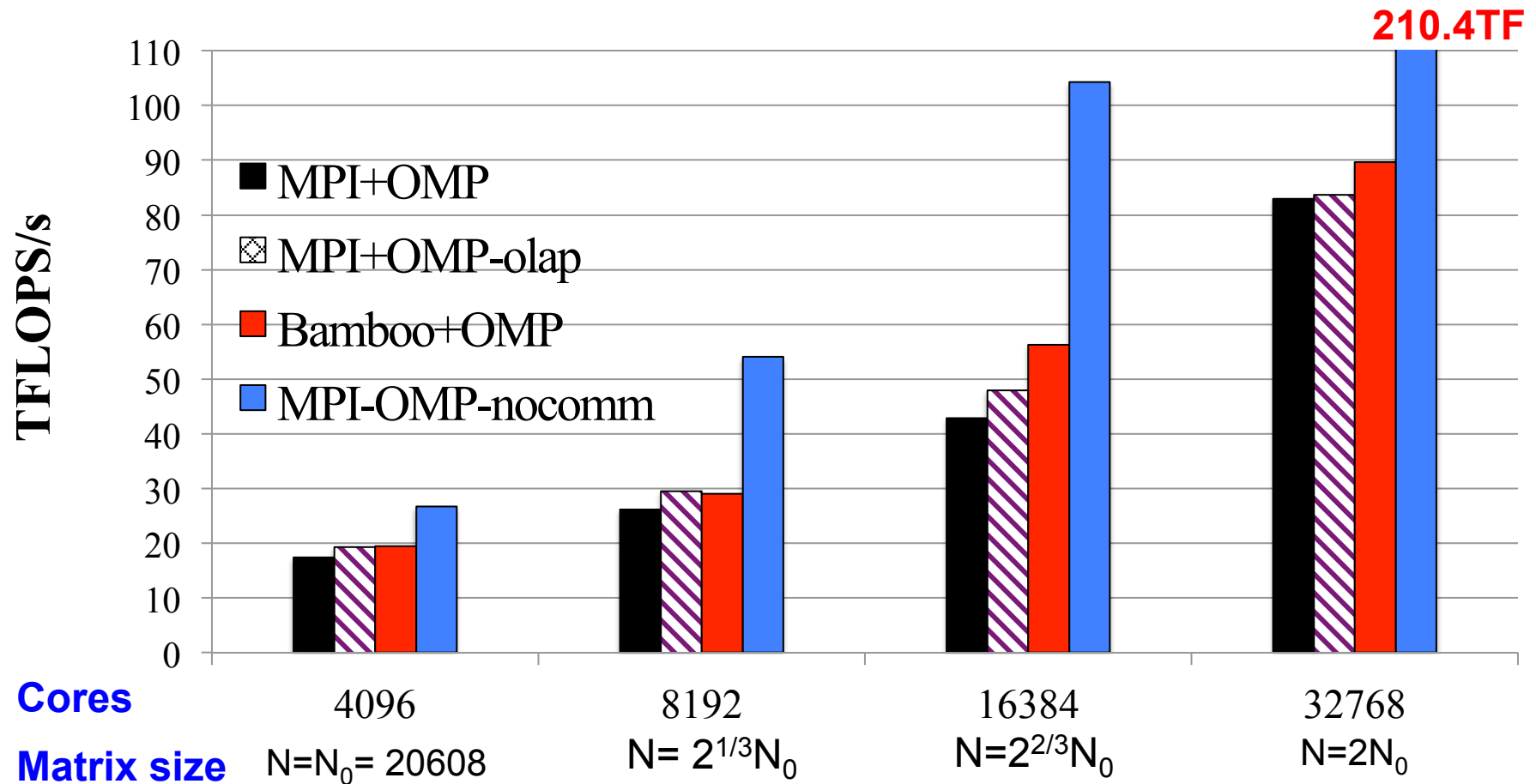
2D Cannon - Weak scaling study



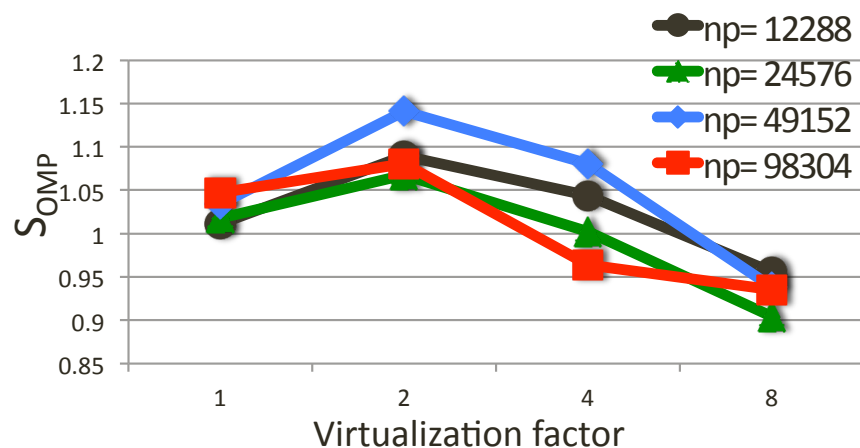
- Communication cost: 11% - 39%
- Bamboo improves MPI-basic 9%-37%
- Bamboo outperforms MPI-olap at scale

Communication Avoiding Matrix Multiplication (Hopper)

- Pathological matrices in Planewave basis methods for *ab-initio* molecular dynamics ($N_g^3 \times N_e$), For Si: $N_g=140$, $N_e=2000$
- Weak scaling study, used OpenMP, 23 pragmas, 337 lines



Virtualization Improves Performance



$c=2, VF=8$

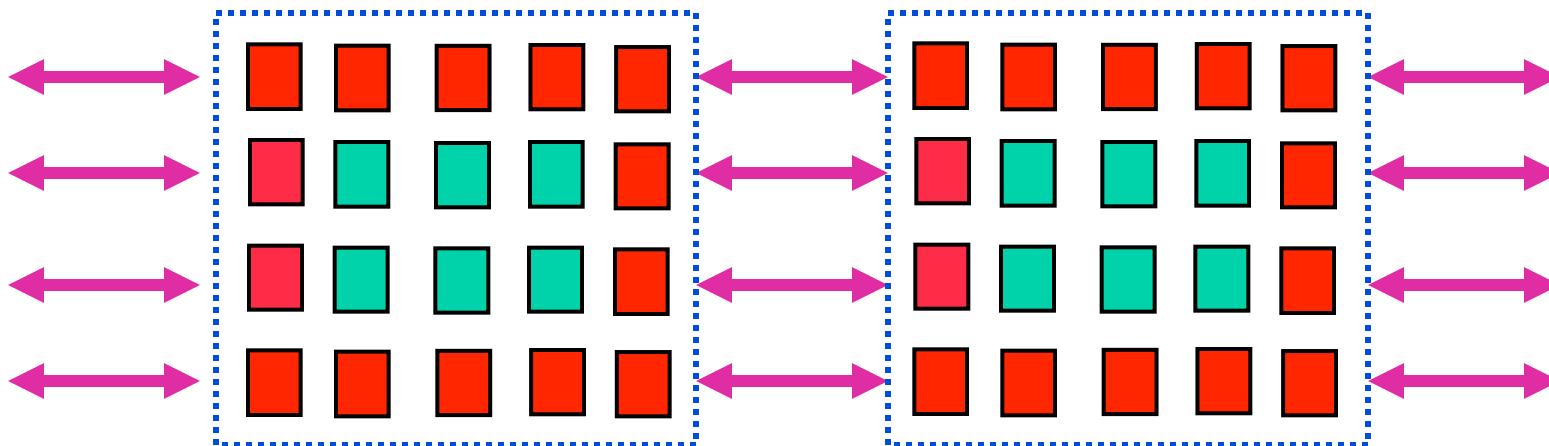
$c=2, VF=4$

$c=2, VF=2$

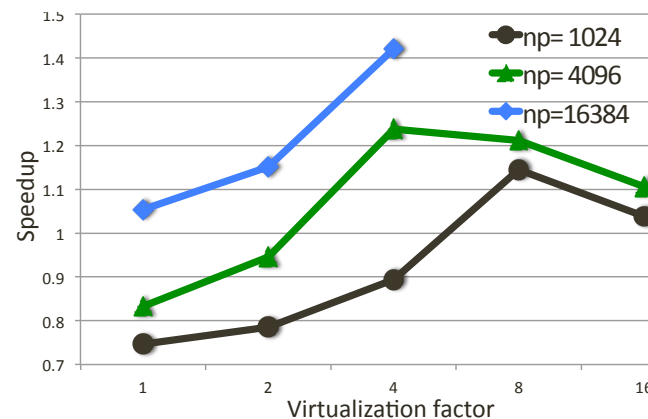
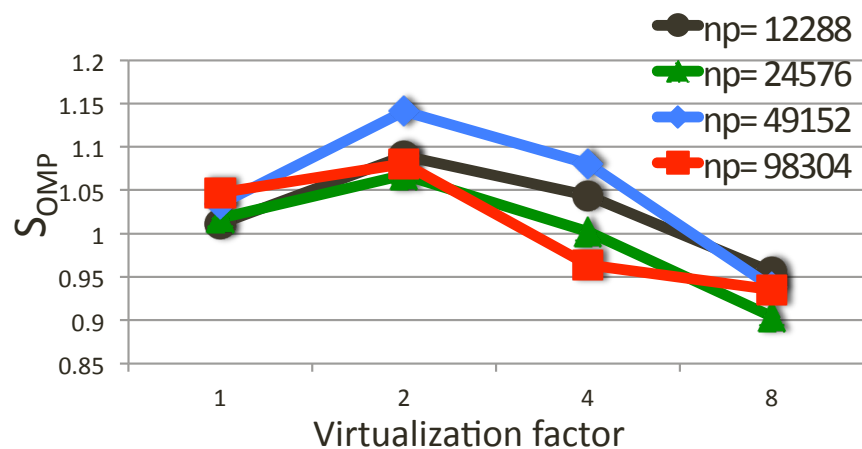
$c=4, VF=2$

Jacobi (MPI+OMP)

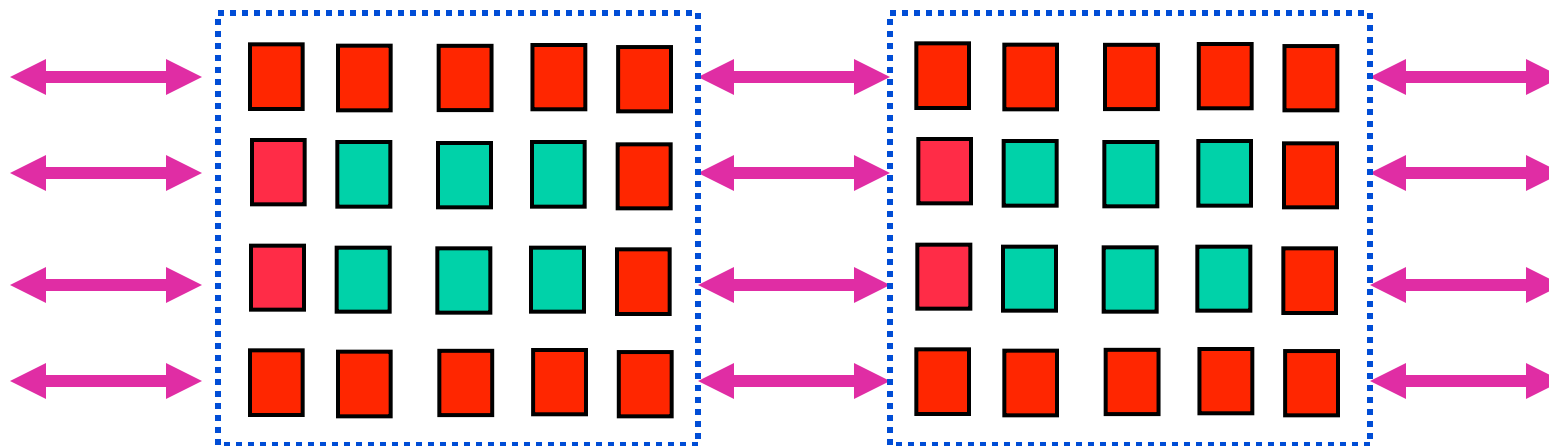
Cannon 2.5D (MPI)



Virtualization Improves Performance



Jacobi (MPI+OMP)



Cannon 2D (MPI)

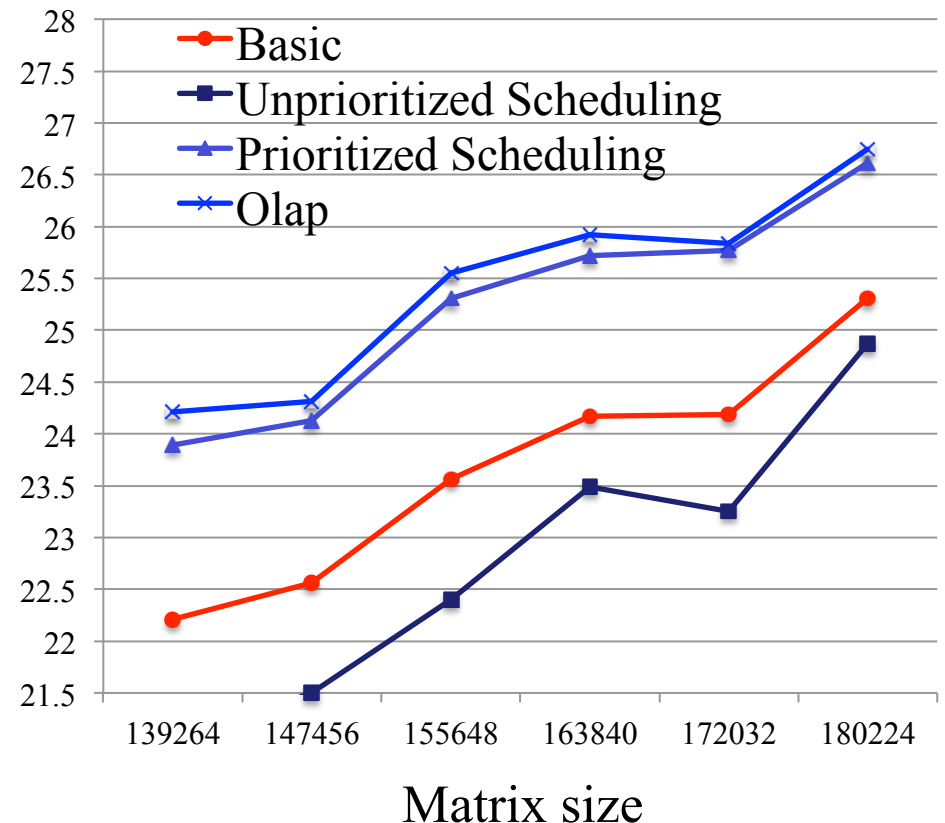
High Performance Linpack (HPL) on Stampede

- Solve systems of linear equations using LU factorization
- Latency-tolerant *lookahead* code is complicated

Finished part of L	Finished part of U	
	D	U_i
	L_i	$A_i = A_i - L_i * U_i$

- Results
 - Bamboo meets the performance of the highly-optimized version of HPL
 - Uses the far simpler non-lookahead version
 - Task prioritization is crucial
 - Bamboo improves the baseline version of HPL by up to 10%

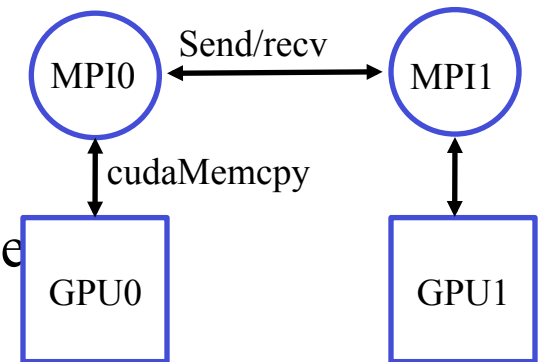
TFLOP/S



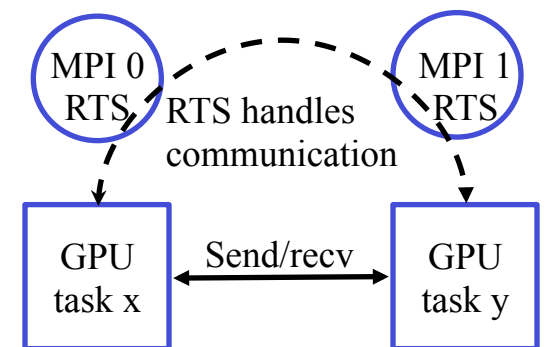
2048 cores on Stampede

Bamboo on multiple GPUs

- MPI+CUDA programming model
 - ◆ CPU is host and GPU works as a device
 - ◆ Host-host with MPI and host-device with CUDA
 - ◆ Optimize MPI and CUDA portions separately
- Need a GPU-aware programming model
 - ◆ Allow a device to transfer data to another device
 - ◆ Compiler and runtime system handle the data transfer
 - ◆ Hide both host-device and host-host communication automatically



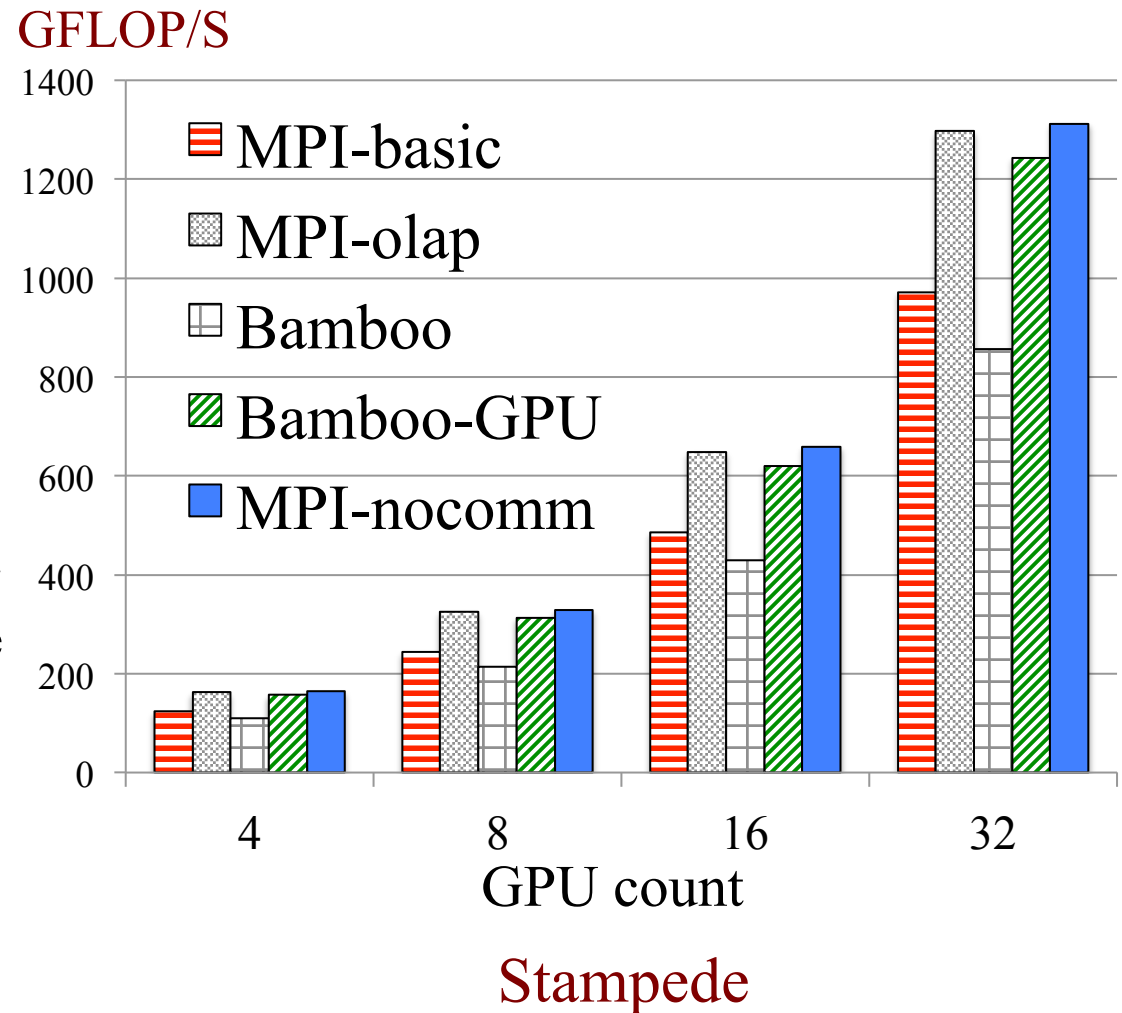
MPI+CUDA



GPU-aware model

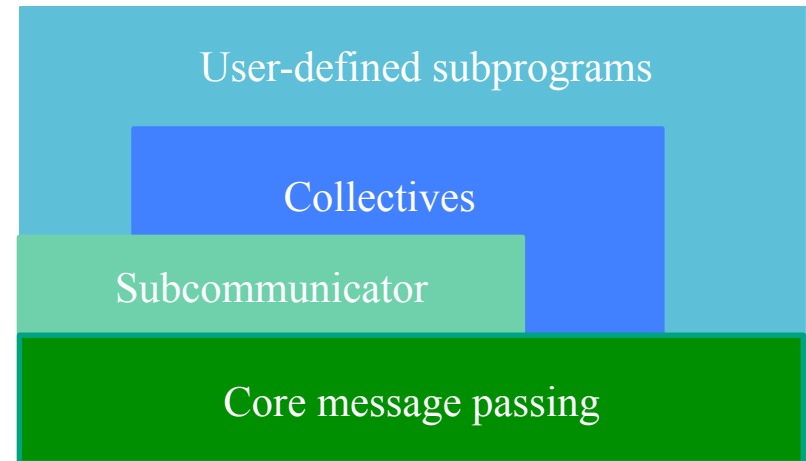
3D Jacobi – Weak Scaling Study

- Results on Stampede
 - ♦ Bamboo-GPU outperforms MPI-basic
 - ♦ Bamboo-GPU and MPI-olap hide most communication overheads
- Bamboo-GPU improves performance by
 - ♦ Hide Host-Host transfer
 - ♦ Hide Host-Device transfer
 - ♦ Tasks residing in the same GPU send address of the message

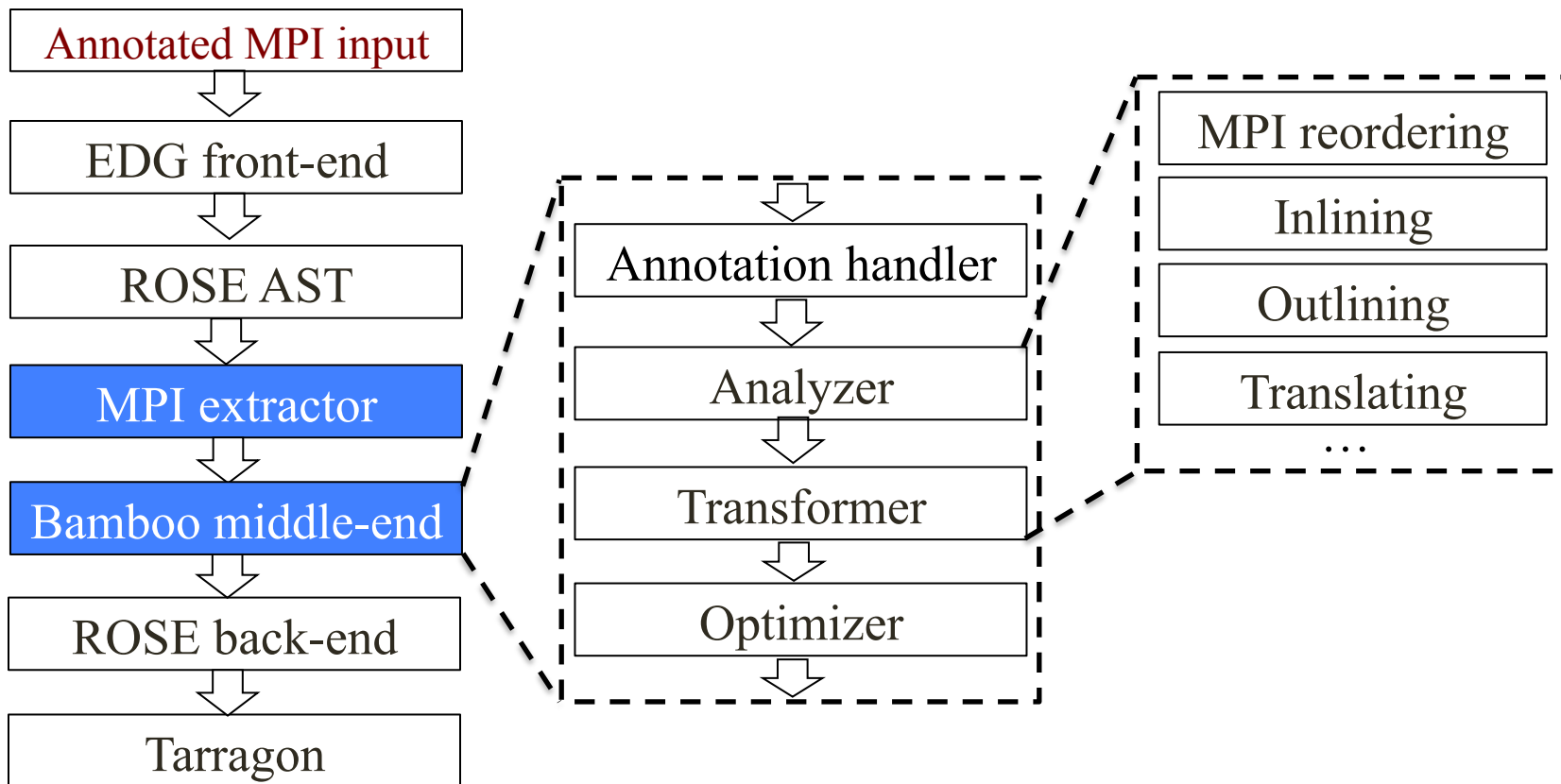


Bamboo Design

- Core message passing
 - ♦ Support point-to-point routines
 - ♦ Require programmer annotation
 - ♦ Employ Tarragon runtime system [Cicotti 06, 11]
- Subcommunicator layer
 - ♦ Support MPI_Comm_split
 - ♦ No annotation required
- Collectives
 - ♦ A framework to translate collectives
 - ♦ Implement common collectives
 - ♦ No annotation required
- User-defined subprograms
 - ♦ A normal MPI program



Bamboo Translator

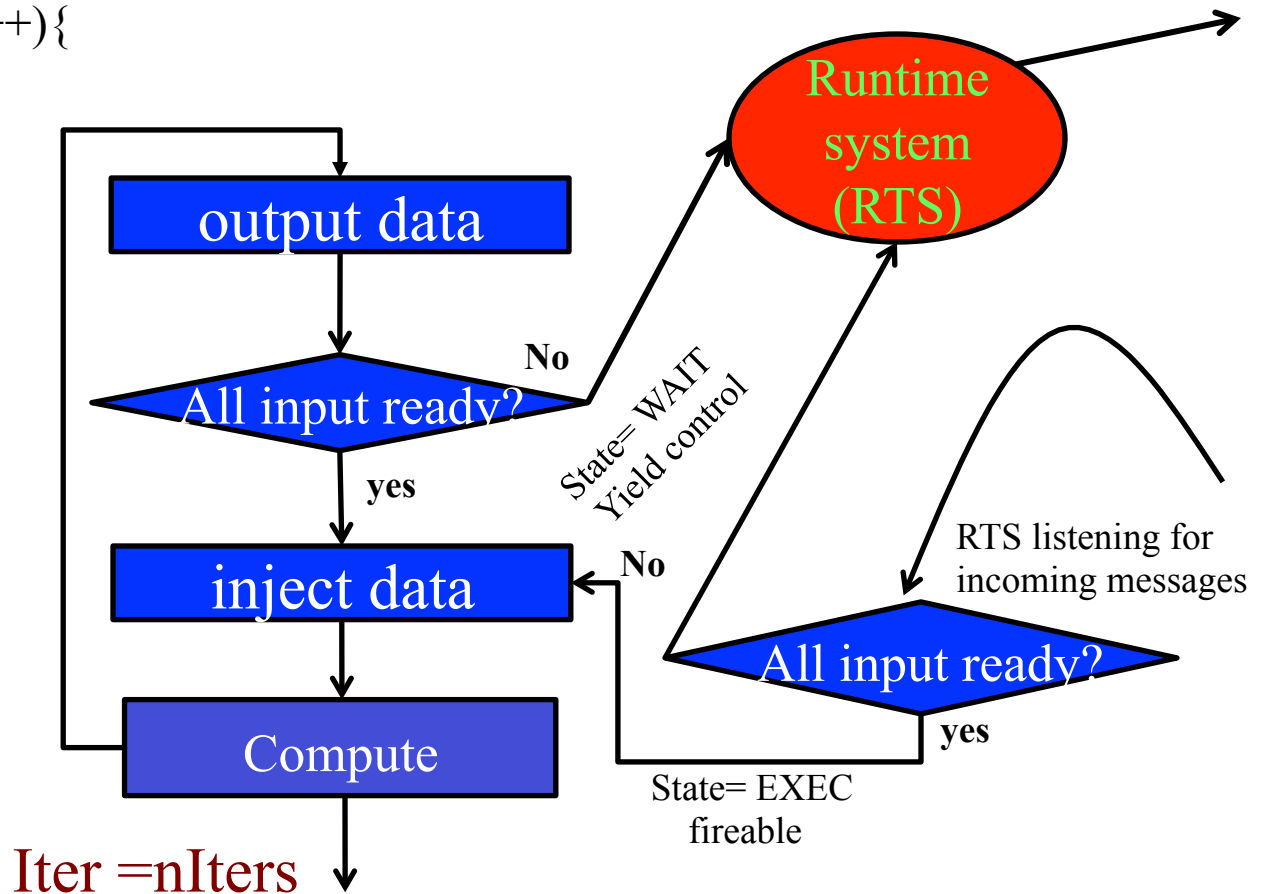


Bamboo Transformations

- **Outlining**
 - ◆ TaskGraph definition: fill various Tarragon methods with input source code blocks
- **MPI Translation:** capture MPI calls and generate calls to Tarragon
 - ◆ Some MPI calls removed, e.g. Barrier(), Wait()
 - ◆ Conservative static analysis to determine task dependencies
- **Code reordering:** reorder certain code to accommodate Tarragon semantics

Code outlining

```
for (int iter=0; iter<nIters; iter++){  
  #pragma bamboo olap  
  {  
    #pragma bamboo send  
    { ... }  
    #pragma bamboo receive  
    { ... }  
  }  
  compute  
}
```



Firing & yielding Rule Generation

- Extract source information in all Recv and iRecv calls of each olap-region, including associated if and for statements
- A task is fireable if and only if it receives messages from all source

```
for(source = 0 to n)
  if(source%2==0)
    MPI_Recv from
    source
```

Recv(0) $\wedge$ Recv (2)
 $\wedge \dots \wedge$ Recv(n)



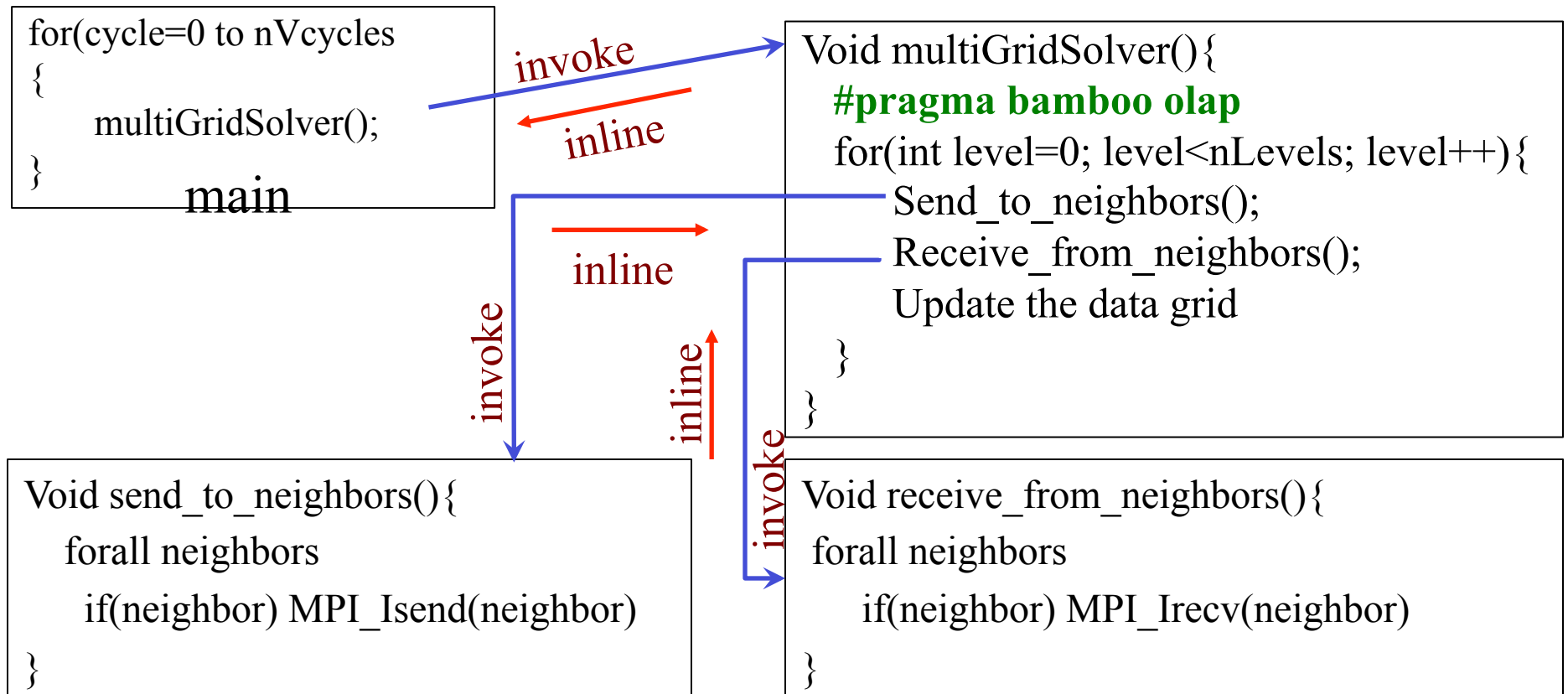
```
bool test(){
  for (source =0 to n)
    if(source%2==0)
      if(notArrivedFrom(source))
      {
        return false;
      }
  return true;
}
```

Firing rule: while (messageArrival) {
 return test ();
 }

Yiedling rule: yield = ! test();

Inter-procedural translation

Bamboo inlines functions that either directly or indirectly hold MPI calls only



Collective implementation

Main file

```
error = MPI_Barrier(comm);
```



Rename collective call

```
Bamboo_Barrier(comm);
```



Merge library's AST to main
Then inline collective call

```
comm_0 = comm;  
#pragma bamboo olap  
for (int step = 1; step < size; step<=1) {  
    MPI_Send(1 byte to (rank+step)%size, comm_0);  
    MPI_Recv(1 byte from (rank-step+size)%size, comm_0);  
}  
error = SUCCESS;
```



Next translation process

A collective Library

```
int Bamboo_Barrier(MPI_Comm comm){  
    #pragma bamboo olap  
    for (int step = 1; step < size; step<=1) {  
        MPI_Send(1 byte to (rank+step)%size, comm);  
        MPI_Recv(1 byte from (rank-step+size)%size, comm);  
    }  
    return SUCCESS;  
}
```