MPI and UPC Programming Distributed Memory Machines and Clusters

Kathy Yelick

yelick@cs.berkeley.edu

http://www.cs.berkeley.edu/~yelick/

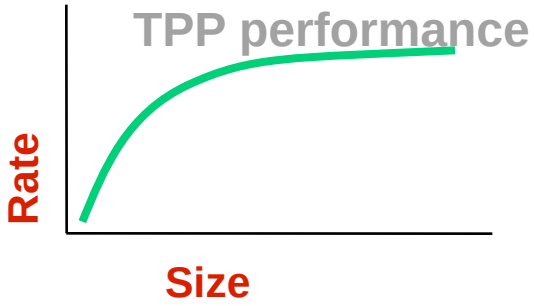
http://upc.lbl.gov

http://titanium.cs.berkeley.edu

TOP500

- Listing of the 500 most powerful Computers in the World
- Yardstick: R_{max} from Linpack

$Ax=b$, *dense problem*



- Updated twice a year:

  ISC‘xy in Germany, June xy

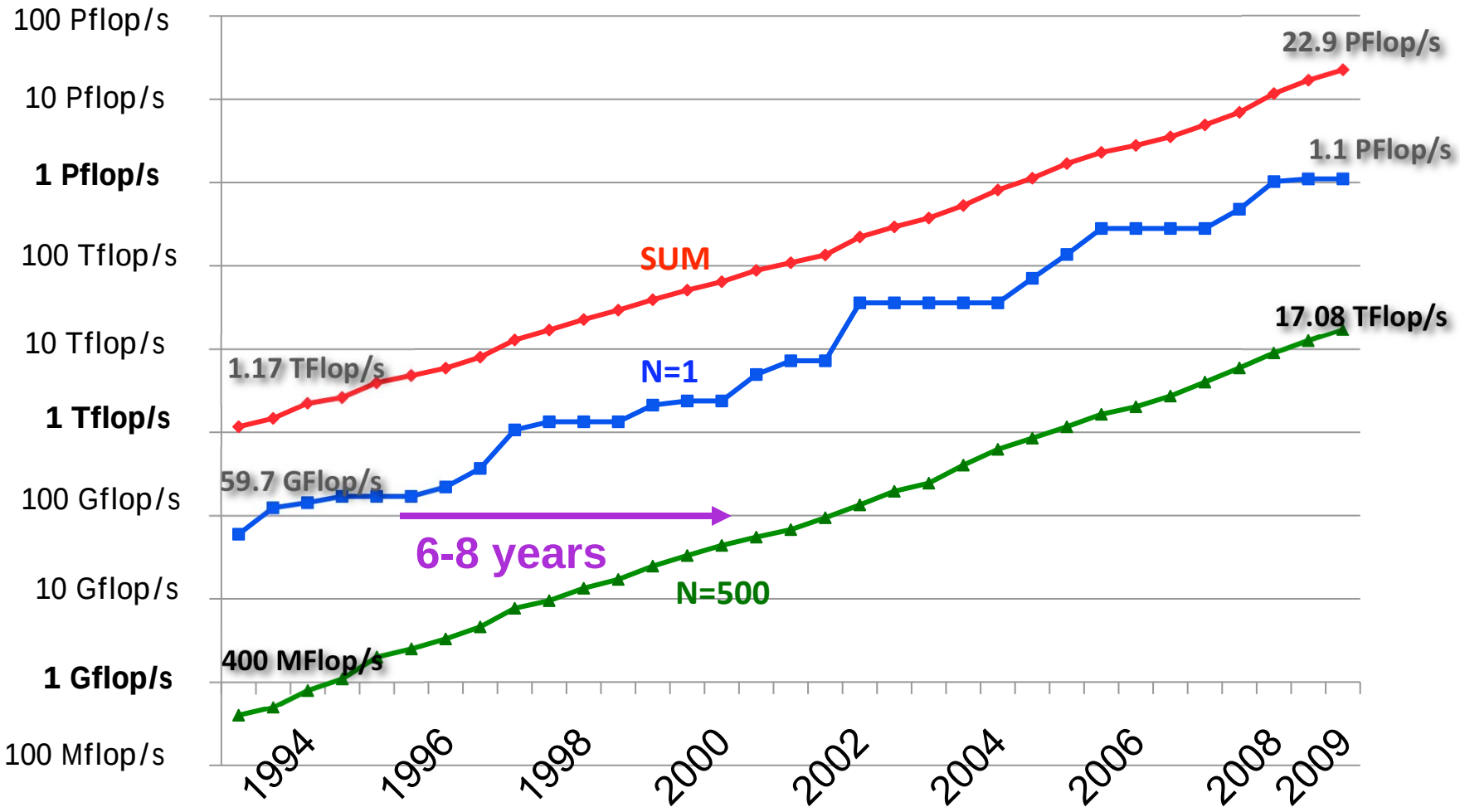
  SC‘xy in USA, November xy

- All data available from **www.top500.org**

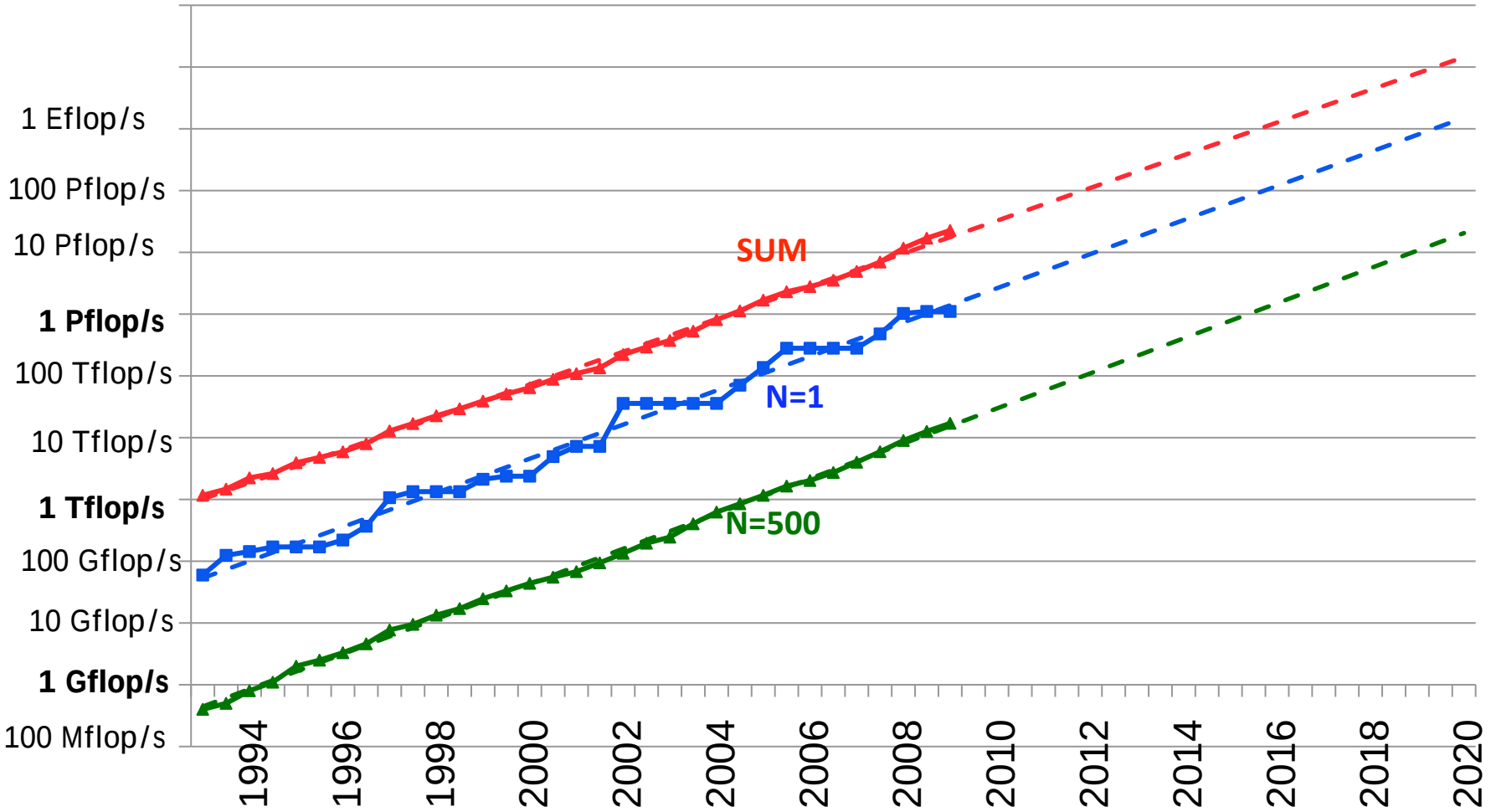
33rd List: The TOP10

1	DOE/NNSA/LANL	IBM	Roadrunner BladeCenter QS22/LS21	USA	129,600	1,105.0	2.48
2	Oak Ridge National Laboratory	Cray Inc.	Jaguar Cray XT5 QC 2.3 GHz	USA	150,152	1,059.0	6.95
3	Forschungszentrum Juelich (FZJ)	IBM	Jugene Blue Gene/P Solution	Germany	294,912	825.50	2.26
4	NASA/Ames Research Center/ NAS	SGI	Pleiades SGI Altix ICE 8200EX	USA	51,200	487.0	2.09
5	DOE/NNSA/LLNL	IBM	BlueGene/L eServer Blue Gene Solution	USA	212,992	478.2	2.32
6	University of Tennessee	Cray	Kraken Cray XT5 QC 2.3 GHz	USA	66,000	463.30	
7	Argonne National Laboratory	IBM	Intrepid Blue Gene/P Solution	USA	163,840	458.61	1.26
8	TACC/U. of Texas	Sun	Ranger SunBlade x6420	USA	62,976	433.2	2.0
9	DOE/NNSA/LLNL	IBM	Dawn Blue Gene/P Solution	USA	147,456	415.70	1.13
10	Forschungszentrum Juelich (FZJ)	Sun/Bull SA	JUROPA NovaScale /Sun Blade	Germany	26,304	274.80	1.54

Performance Development

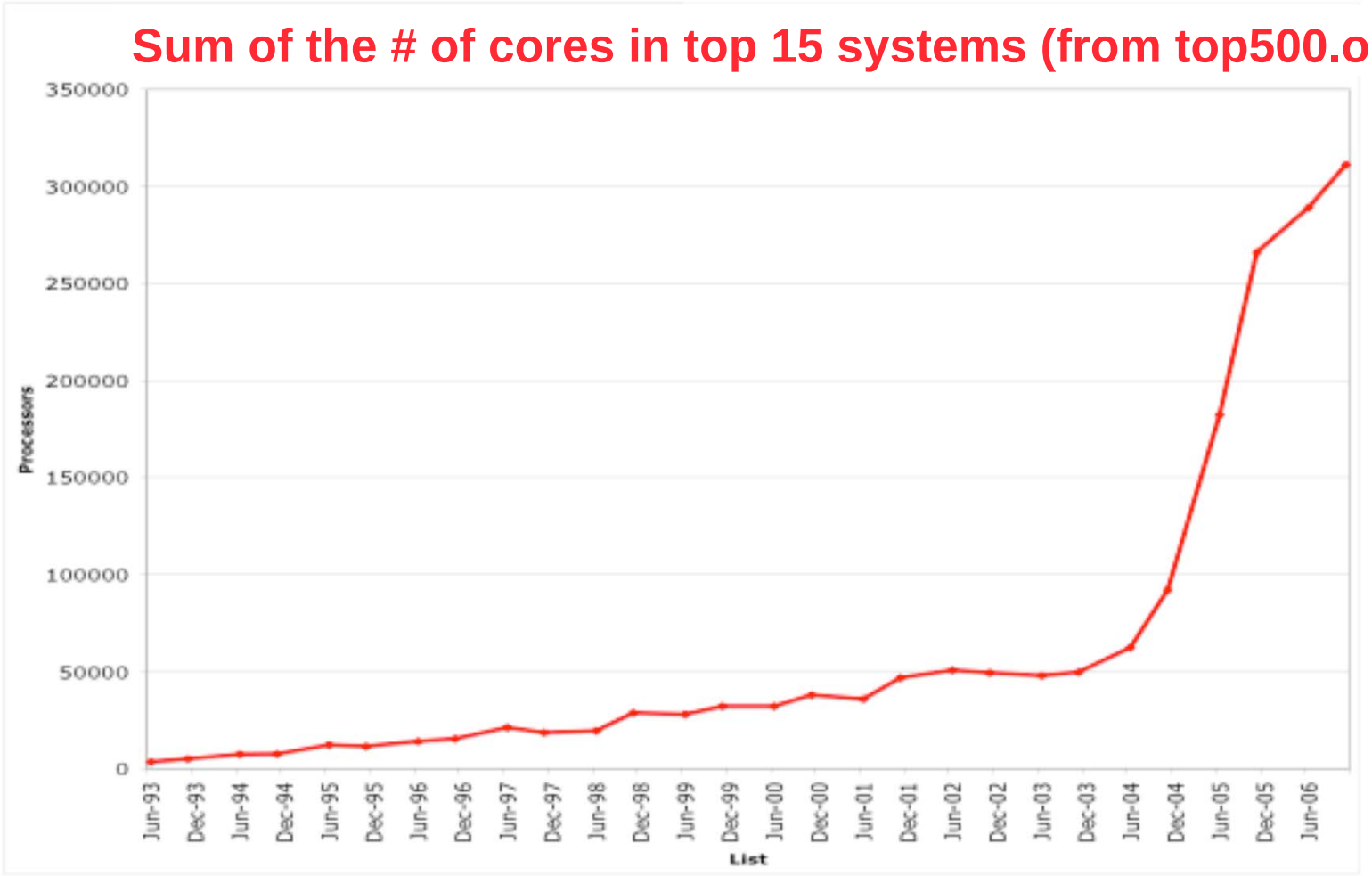


Performance Development Development



Concurrency Has Increased Dramatically

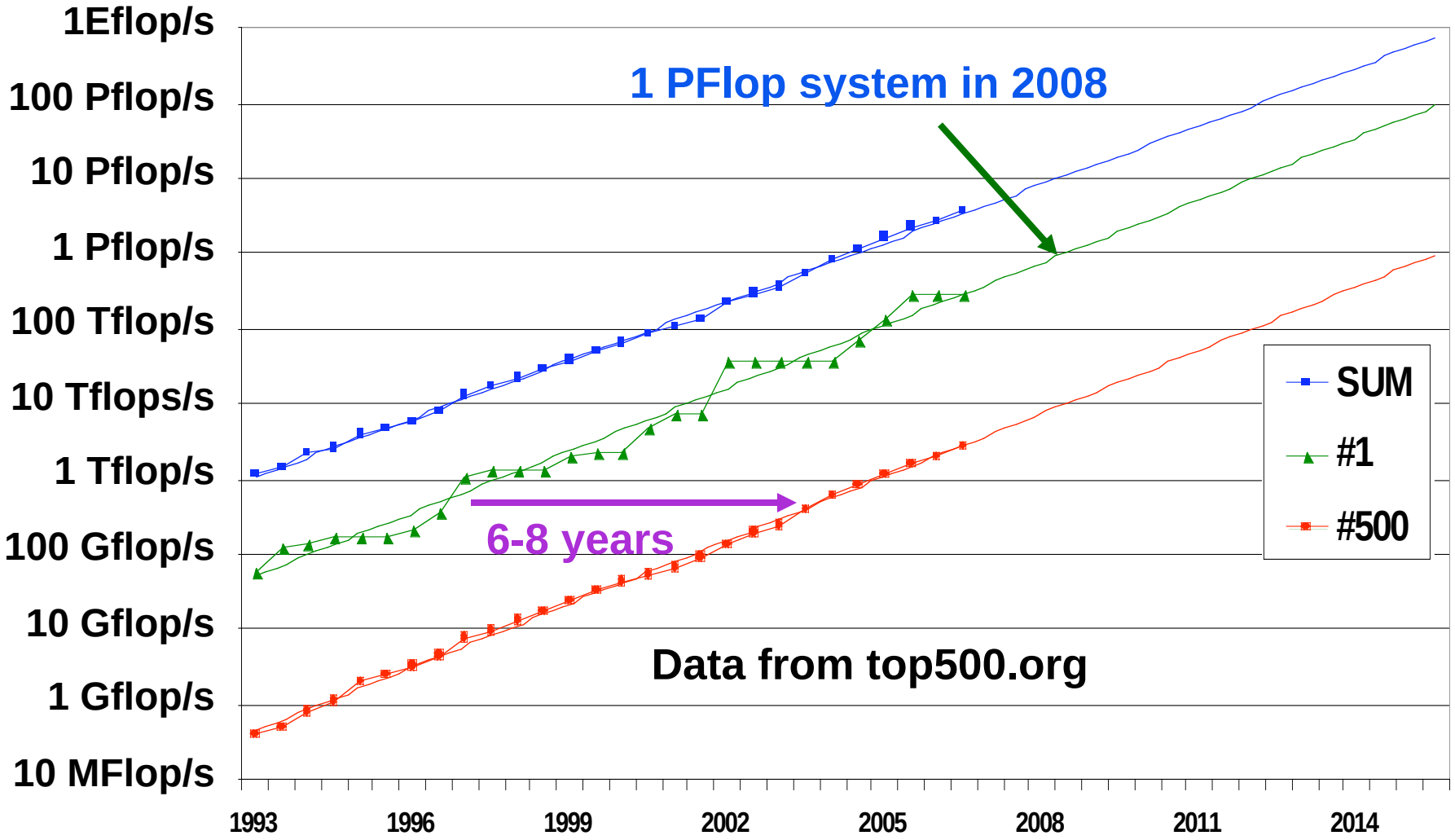
Sum of the # of cores in top 15 systems (from top500.org)



Exponential wave of increasing concurrency for forseeable future!

1M cores sooner than you think!

Petaflop with ~1M Cores Common by 2015?



Slide source Horst Simon, LBNL

Programming With MPI

- MPI is a library
  - All operations are performed with routine calls
  - Basic definitions in
    - mpi.h for C
    - mpif.h for Fortran 77 and 90
    - MPI module for Fortran 90 (optional)
- First Program:
  - Create 4 processes in a simple MPI job
  - Write out process number
  - Write out some variables (illustrate separate name space)

Slide source: Bill Gropp, ANL

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

Slide source: Bill Gropp, ANL

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;
}
```

Slide source: Bill Gropp, ANL

Hello (Fortran)

```
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

Slide source: Bill Gropp, ANL

Hello (C++)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "I am " << rank << " of " << size <<
            "\n";
    MPI::Finalize();
    return 0;
}
```

Slide source: Bill Gropp, ANL

Notes on Hello World

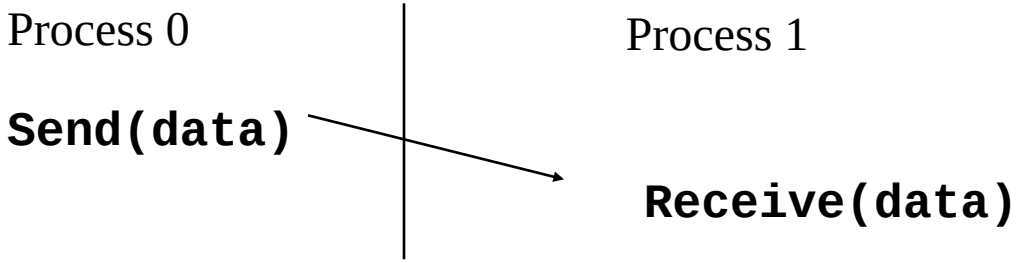
- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `printf/print` statements
- I/O not part of MPI-1but is in MPI-2
  - print and write to standard output or error not part of either MPI-1 or MPI-2
  - output order is undefined (may be interleaved by character, line, or blocks of characters),
- The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide

```
mpirun –np 4 a.out
```

Slide source: Bill Gropp, ANL

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Slide source: Bill Gropp, ANL

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**

Slide source: Bill Gropp, ANL

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays

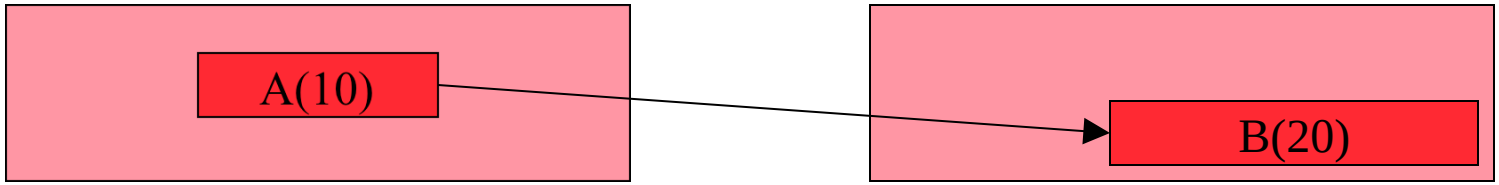
Slide source: Bill Gropp, ANL

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive
- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes

Slide source: Bill Gropp, ANL

MPI Basic (Blocking) Send



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

MPI_SEND(start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`** (rank within **`comm`**)
- When this function returns, the buffer (A) can be reused, but the message may not have been received by the target process.

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (**`source`** and **`tag`**) message is received
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched on or **`MPI_ANY_TAG`**
- Receiving fewer than **`count`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

Slide source: Bill Gropp, ANL

A Simple MPI Program

```
#include “mpi.h”
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &status );
    printf( “Received %d\n”, buf );
  }

  MPI_Finalize();
  return 0;
}
```

Note: Fortran and C++ versions are in online lecture notes

Slide source: Bill Gropp, ANL

A Simple MPI Program (Fortran)

```
      program main
      include 'mpif.h'
      integer rank, buf, ierr, status(MPI_STATUS_SIZE)

      call MPI_Init(ierr)
      call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
C Process 0 sends and Process 1 receives
      if (rank .eq. 0) then
         buf = 123456
         call MPI_Send( buf, 1, MPI_INTEGER, 1, 0,
     *                  MPI_COMM_WORLD, ierr )
      else if (rank .eq. 1) then
         call MPI_Recv( buf, 1, MPI_INTEGER, 0, 0,
     *                  MPI_COMM_WORLD, status, ierr )
         print *, "Received ", buf
      endif
      call MPI_Finalize(ierr)
      end
```

Slide source: Bill Gropp, ANL

A Simple MPI Program (C++)

```
#include “mpi.h”
#include <iostream>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI::Init(argv, argc);
  rank = MPI::COMM_WORLD.Get_rank();

  // Process 0 sends and Process 1 receives
  if (rank == 0) {
    buf = 123456;
    MPI::COMM_WORLD.Send( &buf, 1, MPI::INT, 1, 0 );
  }
  else if (rank == 1) {
    MPI::COMM_WORLD.Recv( &buf, 1, MPI::INT, 0, 0 );
    std::cout << “Received “ << buf << “\n”;
  }

  MPI::Finalize();
  return 0;
}
```

Slide source: Bill Gropp, ANL

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

- In Fortran:

```
integer recvd_tag, recvd_from, recvd_count
integer status(MPI_STATUS_SIZE)
call MPI_RECV(..., MPI_ANY_SOURCE, MPI_ANY_TAG, .. status, ierr)
tag_recvd  = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, datatype, recvd_count, ierr)
```

Slide source: Bill Gropp, ANL

Retrieving Further Information

- **`Status`** is a data structure allocated in the user's program.
- In C++:

```
int recvd_tag, recvd_from, recvd_count;
MPI::Status status;
Comm.Recv(..., MPI::ANY_SOURCE, MPI::ANY_TAG, ...,
          status )
recvd_tag   = status.Get_tag();
recvd_from  = status.Get_source();
recvd_count = status.Get_count( datatype );
```

Slide source: Bill Gropp, ANL

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
  - Operators include: MPI_MAX, MPI_MIN, MPI_PROD, MPI_SUM,…
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency
  - Can use a more efficient algorithm than you might choose for simplicity (e.g., P-1 send/receive pairs for broadcast or reduce)
  - May use special hardware support on some systems

Slide source: Bill Gropp, ANL

Example: PI in C - 1

```
#include "mpi.h"
#include <math.h>
  #include <stdio.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 # of intervals: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;
```

Slide source: Bill Gropp, ANL

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;
}
```

Slide source: Bill Gropp, ANL

Example: PI in Fortran - 1

```
      program main
      include 'mpif.h'
      integer done, n, myid, numprocs, i, rc
      double pi25dt, mypi, pi, h, sum, x, z
      data done/.false./
      data PI25DT/3.14159265358979323846264643/
      call MPI_Init(ierr)
      call MPI_Comm_size(MPI_COMM_WORLD,numprocs, ierr )
      call MPI_Comm_rank(MPI_COMM_WORLD,myid, ierr)
      do while (.not. done)
        if (myid .eq. 0) then
         print *,"Enter the number of intervals: (0 quits)"
         read *, n
        endif
        call MPI_Bcast(n, 1, MPI_INTEGER, 0,
     *                 MPI_COMM_WORLD, ierr )
        if (n .eq. 0) goto 10
```

Slide source: Bill Gropp, ANL

Example: PI in Fortran - 2

```
      h   = 1.0 / n
      sum = 0.0
      do i=myid+1,n,numprocs
        x = h * (i - 0.5)
        sum += 4.0 / (1.0 + x*x)
      enddo
      mypi = h * sum
      call MPI_Reduce(mypi, pi, 1, MPI_DOUBLE_PRECISION,
     *                MPI_SUM, 0, MPI_COMM_WORLD, ierr )
      if (myid .eq. 0) then
          print *, "pi is approximately “, pi,
     *       “, Error is “, abs(pi - PI25DT)
    enddo
10  continue
      call MPI_Finalize( ierr )
    end
```

Slide source: Bill Gropp, ANL

Example: PI in C++ - 1

```
#include "mpi.h"
#include <math.h>
#include <iostream>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.14159265358979323846264643;
  double mypi, pi, h, sum, x, a;
  MPI::Init(argc, argv);
  numprocs = MPI::COMM_WORLD.Get_size();
  myid     = MPI::COMM_WORLD.Get_rank();
  while (!done)  {
    if (myid == 0) {
      std::cout << "Enter the # of intervals: (0 quits) ";
      std::cin >> n;;
    }
    MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0 );
    if (n == 0) break;
```

Slide source: Bill Gropp, ANL

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::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE,
                           MPI::SUM, 0);
    if (myid == 0)
      std::cout << "pi is approximately " << pi <<
            ", Error is " << fabs(pi - PI25DT) << "\n";
  }
  MPI::Finalize();
 return 0;
}
```

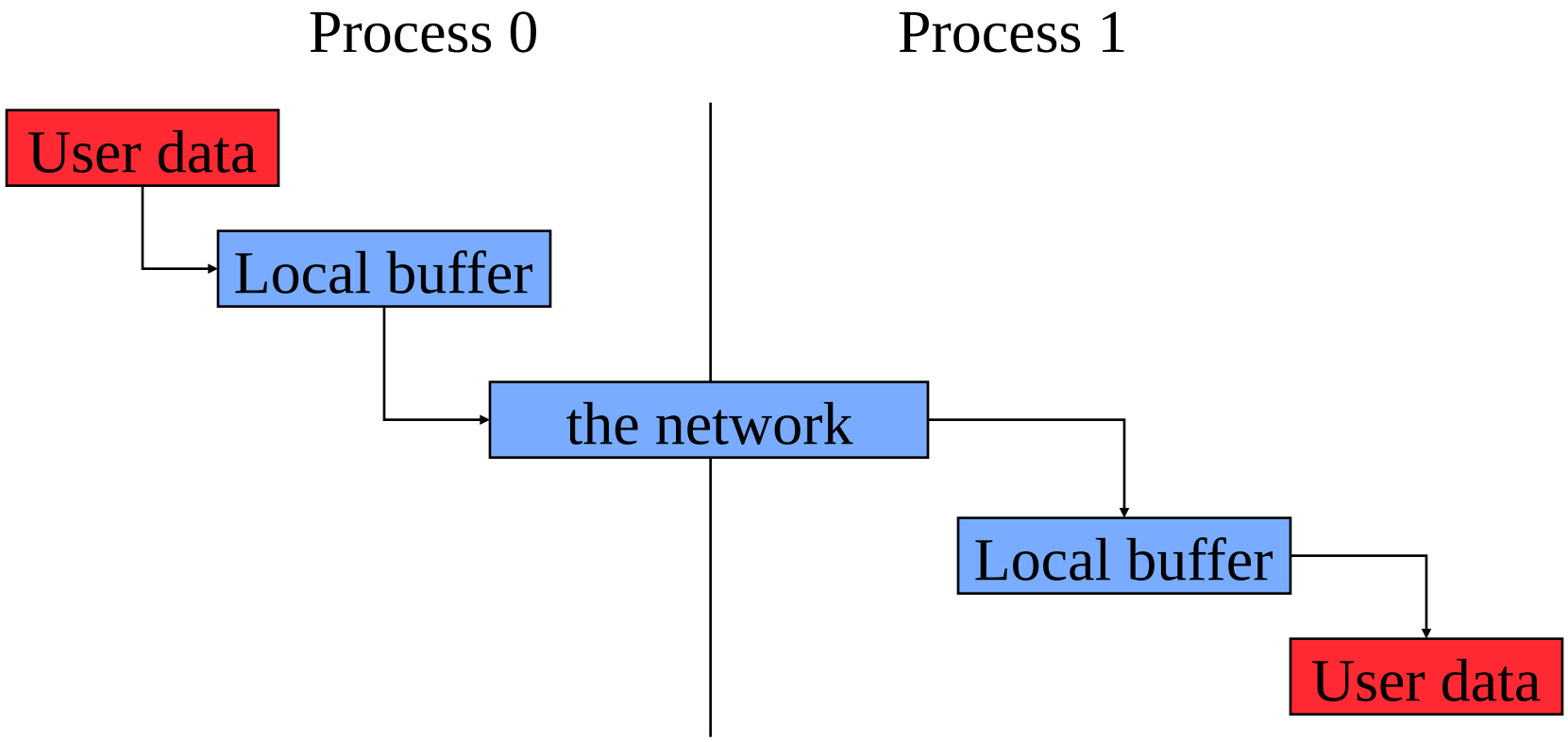
Slide source: Bill Gropp, ANL

MPI Collective Routines

- Many Routines: **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv`**
- **`All`** versions deliver results to all participating processes.
- V versions allow the hunks to have different sizes.
- **`Allreduce`**, **`Reduce`**, **`Reduce_scatter`**, and **`Scan`** take both built-in and user-defined combiner functions.
- MPI-2 adds **`Alltoallw`**, **`Exscan`**, intercommunicator versions of most routines

Buffers

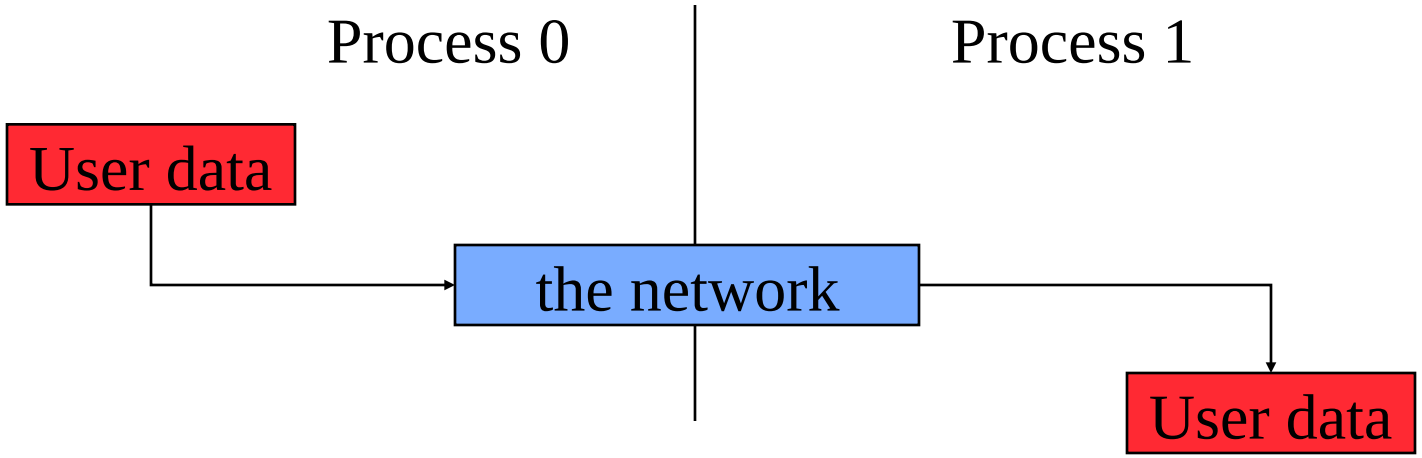
- Message passing has a small set of primitives, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance
- When you send data, where does it go? One possibility is:



Derived from: Bill Gropp, ANL

Avoiding Buffering

- It is better to avoid copies:



This requires that **`MPI_Send`** wait on delivery, or that **`MPI_Send`** return before transfer is complete, and we wait later.

Slide source: Bill Gropp, ANL

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

- This is called “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received

Slide source: Bill Gropp, ANL

Some Solutions to the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
Send(1)	**Recv(0)**
Recv(1)	**Send(0)**

- Supply receive buffer at same time as send:

Process 0	Process 1
Sendrecv(1)	**Sendrecv(0)**

Slide source: Bill Gropp, ANL

More Solutions to the "unsafe" Problem

- Supply own space as buffer for send

Process 0	Process 1
Bsend(1)	**Bsend(0)**
Recv(1)	**Recv(0)**

- Use non-blocking operations:

Process 0	Process 1
Isend(1)	**Isend(0)**
Irecv(1)	**Irecv(0)**
Waitall	**Waitall**

Slide source: Bill Gropp, ANL

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
MPI_Request request;
MPI_Status status;
MPI_Isend(start, count, datatype,
    dest, tag, comm, &request);
MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request);
MPI_Wait(&request, &status);
```

(each request must be Waited on)

- One can also test without waiting:

```
MPI_Test(&request, &flag, &status);
```

Slide source: Bill Gropp, ANL

MPI's Non-blocking Operations (Fortran)

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
integer request
integer status(MPI_STATUS_SIZE)
call MPI_Isend(start, count, datatype,
    dest, tag, comm, request,ierr)
call MPI_Irecv(start, count, datatype,
    dest, tag, comm, request, ierr)
call MPI_Wait(request, status, ierr)
```

(Each request must be waited on)

- One can also test without waiting:

```
call MPI_Test(request, flag, status, ierr)
```

Slide source: Bill Gropp, ANL

MPI's Non-blocking Operations (C++)

- Non-blocking operations return (immediately) "request handles" that can be tested and waited on:

```
MPI::Request request;
MPI::Status  status;
 request = comm.Isend(start, count,
                      datatype, dest, tag);
 request = comm.Irecv(start, count,
                      datatype, dest, tag);
 request.Wait(status);
```

(each request must be Waited on)

- One can also test without waiting:

```
 flag = request.Test( status );
```

Slide source: Bill Gropp, ANL

Other MPI Point-to-Point Features

- It is sometimes desirable to wait on multiple requests:

  `MPI_Waitall(count, array_of_requests, array_of_statuses)`

- Also **`MPI_Waitany`**, **`MPI_Waitsome`**, and test versions
- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**`MPI_Ssend`**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**`MPI_Bsend`**): user supplies a buffer to the system for its use. (User allocates enough memory to avoid deadlock.)
  - Ready mode (**`MPI_Rsend`**): user guarantees that a matching receive has been posted. (Allows access to fast protocols; undefined behavior if matching receive not posted.)

Synchronization

- Global synchronization is available in MPI
  - C: `MPI_Barrier( comm )`
  - Fortran: `MPI_Barrier( comm, ierr )`
  - C++: `comm.Barrier();`
- Blocks until all processes in the group of the communicator **`comm`** call it.
- Almost never required to make a message passing program correct
  - Useful in measuring performance and load balancing

MPI – The de facto standard

MPI has become the de facto standard for parallel computing using message passing

Pros and Cons of standards

- **MPI created finally a standard for applications development in the HPC community → portability**
- **The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation**

Programming Model reflects hardware!

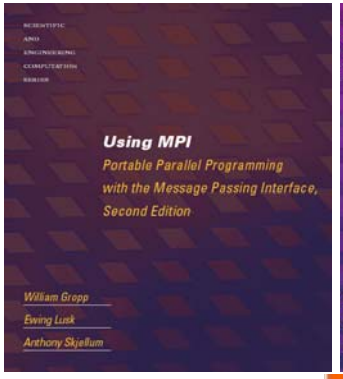
"I am not sure how I will program a Petaflops computer, but I am sure that I will need MPI somewhere" – HDS 2001

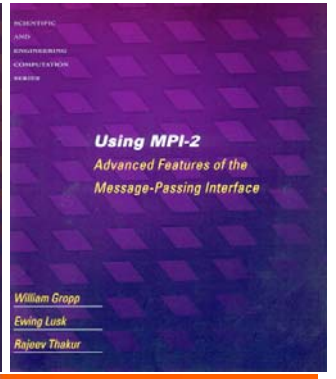
MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Slide source: Bill Gropp, ANL

Books on MPI





- *Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition)*, by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- *MPI: The Complete Reference - Vol 1 The MPI Core,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- *MPI: The Complete Reference - Vol 2 The MPI Extensions*, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.

Slide source: Bill Gropp, ANL

Partitioned Global Address Space Languages

One-Sided Communication

What's Wrong with MPI Everywhere

- We can run 1 MPI process per core
  - This works now (for CMPs) and will work for a while
- How long will it continue working?
  - 4 - 8 cores? Probably. 128 - 1024 cores? Probably not.
  - Depends on performance expectations -- more on this later
- What is the problem?
  - Latency: some copying required by semantics
  - Memory utilization: partitioning data for separate address space requires some replication
    - How big is your per core subgrid? At 10x10x10, over 1/2 of the points are surface points, probably replicated
  - Memory bandwidth: extra state means extra bandwidth
  - Weak scaling: success model for the "cluster era;" will not be for the many core era -- not enough memory per core
  - Heterogeneity: MPI per CUDA thread-block?
- Advantage: no new apps work; modest infrastructure work (multicore-optimized MPI)

Current Implementations of PGAS Languages

- A successful language/library must run everywhere
- UPC
  - **Commercial compilers available on Cray, SGI, HP machines**
  - **Open source compiler from LBNL/UCB (source-to-source)**
  - **Open source gcc-based compiler from Intrepid**
- CAF
  - **Commercial compiler available on Cray machines**
  - **Open source compiler available from Rice**
- Titanium
  - **Open source compiler from UCB runs on most machines**
- **DARPA HPCS Languages**
  - **Cray Chapel, IBM X10, Sun Fortress**
  - **Use PGAS memory abstraction, but have dynamic threading**
  - **Recent additions to parallel language landscape → no mature compilers for clusters yet**

Unified Parallel C (UPC)

Overview and Design Philosophy

- Unified Parallel C (UPC) is:
  - An explicit parallel extension of ANSI C
  - A partitioned global address space language
  - Sometimes called a GAS language
- Similar to the C language philosophy
  - Programmers are clever and careful, and may need to get close to hardware
    - to get performance, but
    - can get in trouble
  - Concise and efficient syntax
- Common and familiar syntax and semantics for parallel C with simple extensions to ANSI C
- Based on ideas in Split-C, AC, and PCP

UPC Execution Model

UPC Execution Model

- Threads working independently in a SPMD fashion
  - Number of threads specified at compile-time or run-time; available as program variable **`THREADS`**
  - **`MYTHREAD`** specifies thread index (**`0..THREADS-1`**)
  - **`upc_barrier`** is a global synchronization: all wait
  - There is a form of parallel loop that we will see later
- There are two compilation modes
  - Static Threads mode:
    - THREADS is specified at compile time by the user
    - The program may use THREADS as a compile-time constant
  - Dynamic threads mode:
    - Compiled code may be run with varying numbers of threads

Hello World in UPC

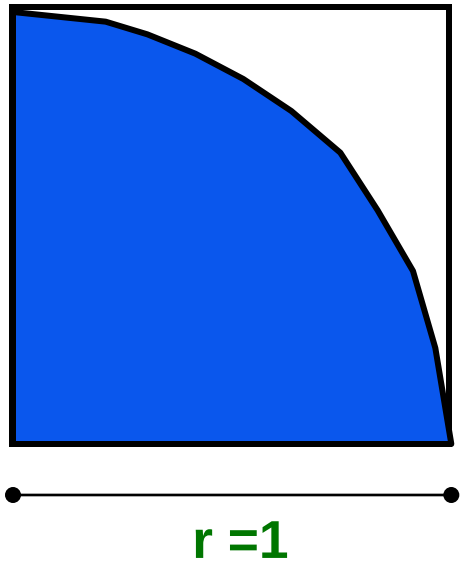
- Any legal C program is also a legal UPC program
- If you compile and run it as UPC with P threads, it will run P copies of the program.
- Using this fact, plus the identifiers from the previous slides, we can parallel hello world:

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

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

Example: Monte Carlo Pi Calculation

- Estimate Pi by throwing darts at a unit square
- Calculate percentage that fall in the unit circle
  - Area of square = $r^2 = 1$
  - Area of circle quadrant = $¼ * \pi r^2 = \pi/4$
- Randomly throw darts at x,y positions
- If $x^2 + y^2 < 1$, then point is inside circle
- Compute ratio:
  - # points inside / # points total
  - π = 4*ratio



Pi in UPC

- Independent estimates of pi:

```
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 use input arguments

Initialize random in math library

Each thread calls “hit” separately

Helper Code for Pi in UPC

- 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);
  }
}
```

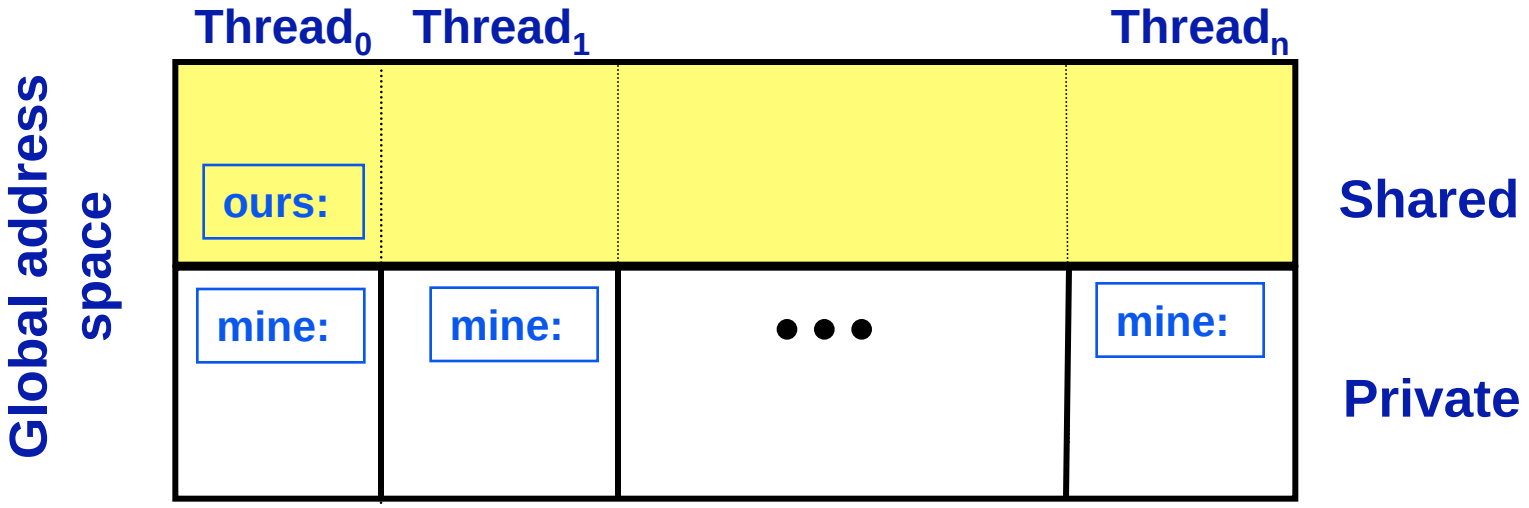
Shared vs. Private Variables

Private vs. Shared Variables in UPC

- Normal C variables and objects are allocated in the private memory space for each thread.
- Shared variables are allocated only once, with thread 0

```
shared int ours;  // use sparingly: performance
int mine;
```

- Shared variables may not have dynamic lifetime: may not occur in a in a function definition, except as static. Why?



Pi in UPC: Shared Memory Style

- Parallel computing of pi, but with a bug

```
shared int hits;
main(int argc, char **argv) {
    int i, my_trials = 0;
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
      hits += hit();
    upc_barrier;
    if (MYTHREAD == 0) {
      printf("PI estimated to %f.", 4.0*hits/trials);
    }
}
```

shared variable to record hits

divide work up evenly

accumulate hits

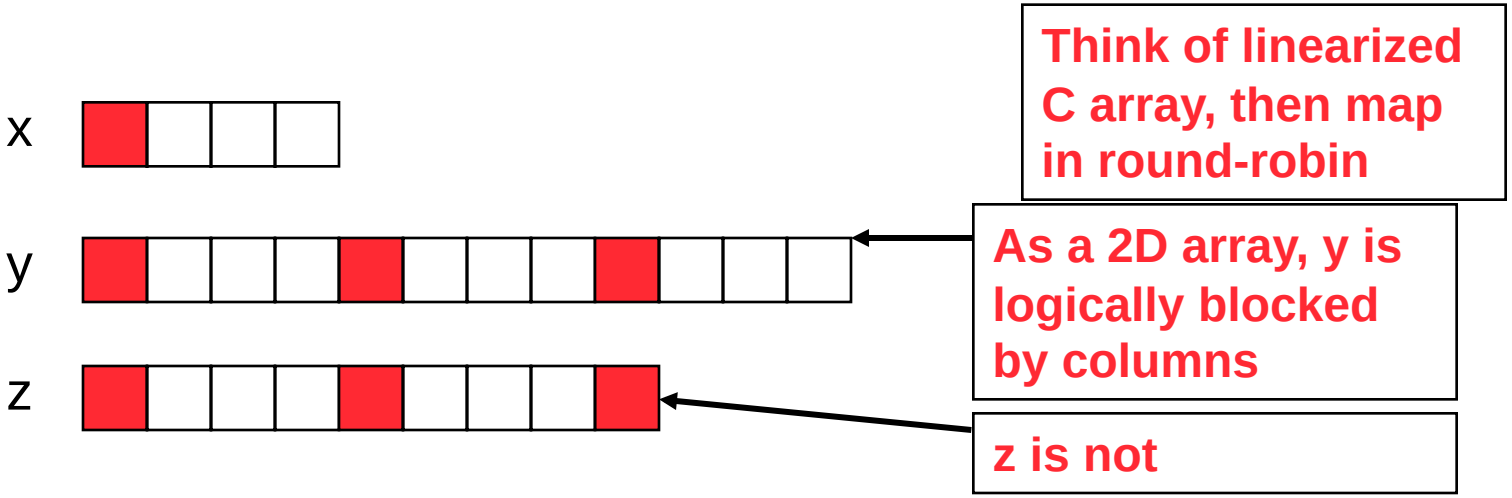
What is the problem with this program?

Shared Arrays Are Cyclic By Default

- Shared scalars always live in thread 0
- Shared arrays are spread over the threads
- Shared array elements are spread across the threads

```
shared int x[THREADS]      /* 1 element per thread */
shared int y[3][THREADS]   /* 3 elements per thread */
shared int z[3][3]         /* 2 or 3 elements per thread */
```

- In the pictures below, assume THREADS = 4
  - Red elts have affinity to thread 0



Pi in UPC: Shared Array Version

- Alternative fix to the race condition
- Have each thread update a separate counter:
  - But do it in a shared array
  - Have one thread compute sum

```
shared int all_hits [THREADS];
main(int argc, char **argv) {
   … declarations an initialization code omitted
   for (i=0; i < my_trials; i++)
      all_hits[MYTHREAD] += hit();
   upc_barrier;
   if (MYTHREAD == 0) {
      for (i=0; i < THREADS; i++) hits += all_hits[i];
      printf("PI estimated to %f.", 4.0*hits/trials);
   }
}
```

all_hits is shared by all processors, just as hits was

update element with local affinity

UPC Synchronization

UPC Global Synchronization

- UPC has two basic forms of barriers:
  - Barrier: block until all other threads arrive

    `upc_barrier`
  - Split-phase barriers

    `upc_notify;` this thread is ready for barrier

    do computation unrelated to barrier

    `upc_wait;` wait for others to be ready
- Optional labels allow for debugging

```
#define MERGE_BARRIER 12
if (MYTHREAD%2 == 0) {
      ...
      upc_barrier MERGE_BARRIER;
} else {
      ...
      upc_barrier MERGE_BARRIER;
}
```

Synchronization - Locks

- UPC Locks are an opaque type:

```
upc_lock_t
```

- Locks must be allocated before use:

```
upc_lock_t *upc_all_lock_alloc(void);
```

allocates 1 lock, pointer to all threads

```
upc_lock_t *upc_global_lock_alloc(void);
```

allocates 1 lock, pointer to one thread

- To use a lock:

```
void upc_lock(upc_lock_t *l)
void upc_unlock(upc_lock_t *l)
```

use at start and end of critical region

- Locks can be freed when not in use

```
void upc_lock_free(upc_lock_t *ptr);
```

Pi in UPC: Shared Memory Style

- Parallel computing of pi, without the bug

```
shared int hits;
main(int argc, char **argv) {
    int i, my_hits, my_trials = 0;
    upc_lock_t *hit_lock = upc_all_lock_alloc();
    int trials = atoi(argv[1]);
    my_trials = (trials + THREADS - 1)/THREADS;
    srand(MYTHREAD*17);
    for (i=0; i < my_trials; i++)
      my_hits += hit();
    upc_lock(hit_lock);
    hits += my_hits;
    upc_unlock(hit_lock);
    upc_barrier;
    if (MYTHREAD == 0)
      printf("PI: %f", 4.0*hits/trials);
}
```

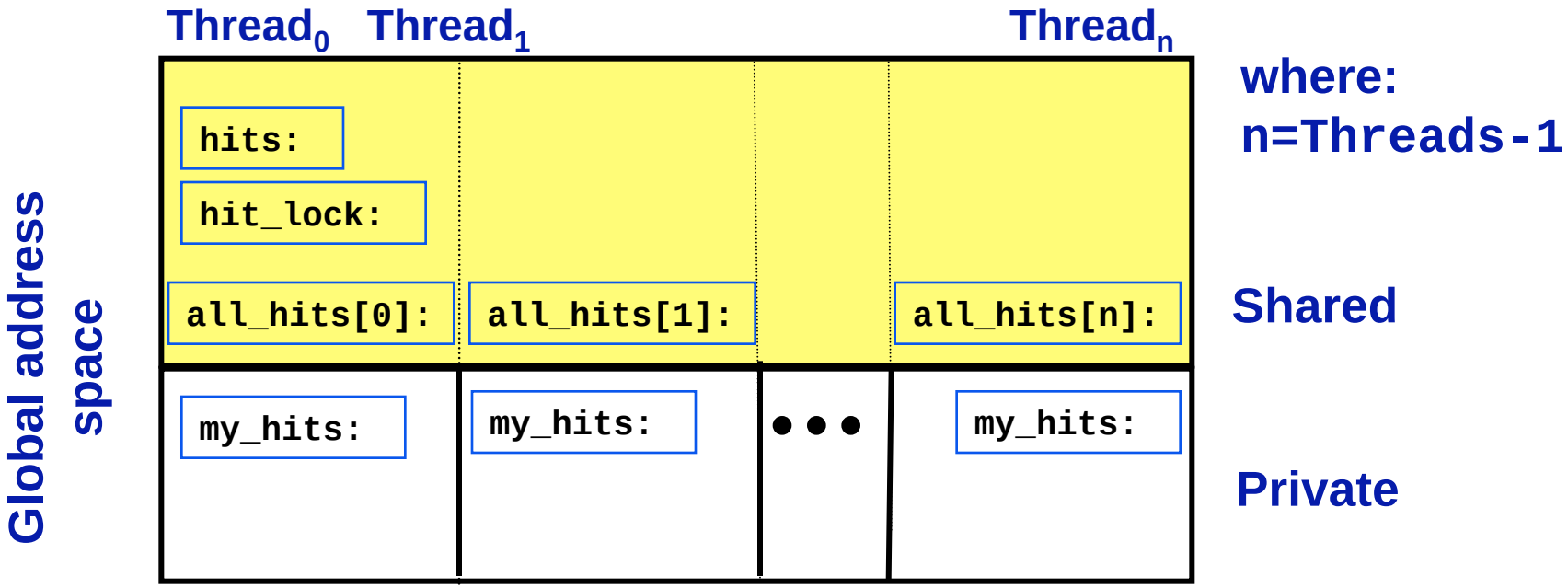
create a lock

accumulate hits locally

accumulate across threads

Recap: Private vs. Shared Variables in UPC

- We saw several kinds of variables in the pi example
  - Private scalars (**`my_hits`**)
  - Shared scalars (**`hits`**)
  - Shared arrays (**`all_hits`**)
  - Shared locks (**`hit_lock`**)



UPC Collectives

UPC Collectives in General

- UPC collectives interface is in the language spec:
  - http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf
- It contains typical functions:
  - Data movement: broadcast, scatter, gather, …
  - Computational: reduce, prefix, …
- General interface has synchronization modes:
  - Avoid over-synchronizing (barrier before/after)
  - Data being collected may be read/written by any thread simultaneously
- Simple interface for scalar values (int, double,…)
  - Berkeley UPC value-based collectives
  - Works with any compiler
  - http://upc.lbl.gov/docs/user/README-collectivev.txt

Pi in UPC: Data Parallel Style

- The previous version of Pi works, but is not scalable:
  - On a large # of threads, the locked region will be a bottleneck
- Use a reduction for better scalability

```
#include <bupc_collectivev.h>
// shared int hits;
main(int argc, char **argv) {
    ...
    for (i=0; i < my_trials; i++)
       my_hits += hit();
    my_hits =          // type, input, thread, op
       bupc_allv_reduce(int, my_hits, 0, UPC_ADD);
    // upc_barrier;
    if (MYTHREAD == 0)
      printf("PI: %f", 4.0*my_hits/trials);
}
```

Berkeley collectives

no shared variables

barrier implied by collective

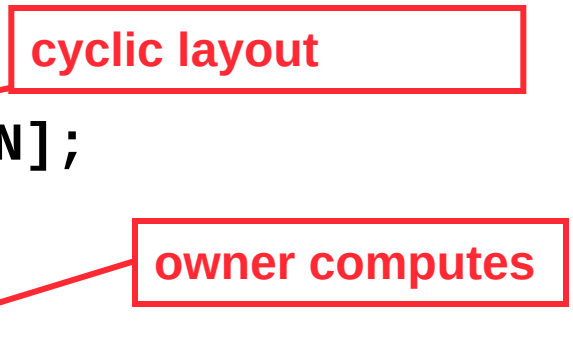
Work Distribution Using `upc_forall`

Example: Vector Addition

- Questions about parallel vector additions:
  - How to layout data (here it is cyclic)
  - Which processor does what (here it is “owner computes”)

```
/* vadd.c */
#include <upc_relaxed.h>
#define N 100*THREADS

shared int v1[N], v2[N], sum[N];
void main() {
    int i;
    for(i=0; i<N; i++)
     if (MYTHREAD == i
%THREADS)
    sum[i]=v1[i]+v2[i];
}
```



upc_forall()

- The idiom in the previous slide is very common
  - Loop over all; work on those owned by this proc
- UPC adds a special type of loop

```
upc_forall(init; test; loop; affinity)
    statement;
```

- Programmer indicates the iterations are independent
  - Undefined if there are dependencies across threads
- Affinity expression indicates which iterations to run on each thread. It may have one of two types:
  - Integer: **affinity%THREADS** is **MYTHREAD**
  - Pointer: **upc_threadof(affinity)** is **MYTHREAD**
- **Syntactic sugar for loop on previous slide**
  - *Some* compilers *may* do better than this, e.g.,

```
for(i=MYTHREAD; i<N; i+=THREADS)
```

  - Rather than having all threads iterate N times:

```
for(i=0; i<N; i++) if (MYTHREAD == i%THREADS)
```

Vector Addition with upc_forall

- The **vadd** example can be rewritten as follows
  - Equivalent code could use "**&sum[i]**" for affinity
  - The code would be correct but slow if the affinity expression were **i+1** rather than **i**.

```
#define N 100*THREADS

  shared int v1[N], v2[N], sum[N];

  void main() {
      int i;
      upc_forall(i=0; i<N; i++;
  i)
                  sum[i]=v1[i]+v2[i];
  }
```

The cyclic data distribution may perform poorly on some machines

Distributed Arrays in UPC

Blocked Layouts in UPC

- If this code were doing nearest neighbor averaging (3pt stencil) the cyclic layout would be the worst possible layout.
- Instead, want a blocked layout
- Vector addition example can be rewritten as follows using a blocked layout

```
#define N 100*THREADS
   shared int [*] v1[N], v2[N], sum[N];

   void main() {
        int i;
        upc_forall(i=0; i<N; i++;
   &sum[i])
                    sum[i]=v1[i]+v2[i];
   }
```

blocked layout

Layouts in General

- All non-array objects have affinity with thread zero.
- Array layouts are controlled by layout specifiers:
  - Empty (cyclic layout)
  - [*] (blocked layout)
  - [0] or [] (indefinite layout, all on 1 thread)
  - [b] or [b1][b2]…[bn] = [b1*b2*…bn] (fixed block size)
- The affinity of an array element is defined in terms of:
  - block size, a compile-time constant
  - and THREADS.
- Element i has affinity with thread

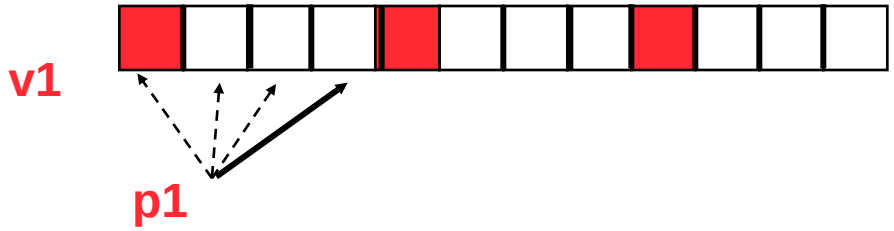
  `(i / block_size) % THREADS`
- In 2D and higher, linearize the elements as in a C representation, and then use above mapping

Pointers to Shared vs. Arrays

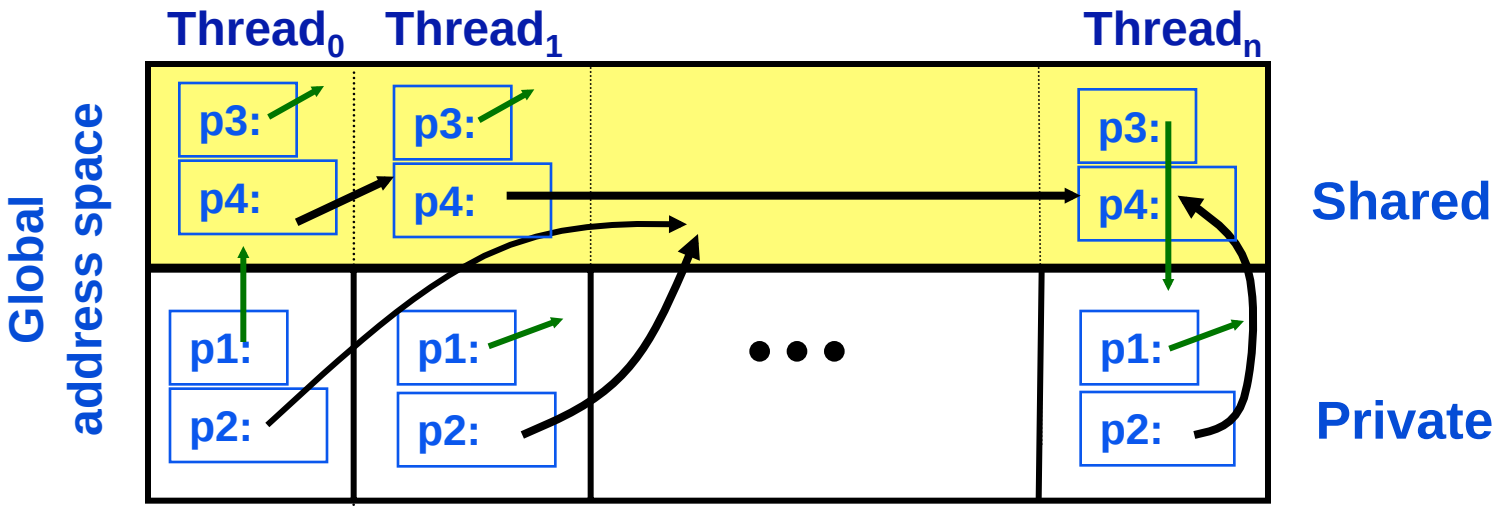
- In the C tradition, arrays can be access through pointers
- Here is the vector addition example using pointers

```
#define N 100*THREADS
shared int v1[N], v2[N], sum[N];
void main() {
  int i;
  shared int *p1, *p2;

  p1=v1; p2=v2;
  for (i=0; i<N; i++, p1++, p2++ )
     if (i %THREADS= = MYTHREAD)
         sum[i]= *p1 + *p2;
  }
```



UPC Pointers




```
int *p1;          /* private pointer to local memory */
shared int *p2;   /* private pointer to shared space */
int *shared p3;   /* shared pointer to local memory */
shared int  *shared p4; /* shared pointer to
                           shared space */
```

Pointers to shared often require more storage and are more costly to dereference; they may refer to local or remote memory.

Dynamic Memory Allocation in UPC

- Dynamic memory allocation of shared memory is available in UPC
- Non-collective (called independently)

```
shared void *upc_global_alloc(size_t nblocks,
                              size_t nbytes);
```

nblocks : number of blocks

nbytes : block size

- Collective (called together; all threads get same pointer)

```
shared void *upc_all_alloc(size_t nblocks,
                           size_t nbytes);
```

- Freeing dynamically allocated memory in shared space

```
void upc_free(shared void *ptr);
```

Performance of UPC

PGAS Languages have Performance Advantages

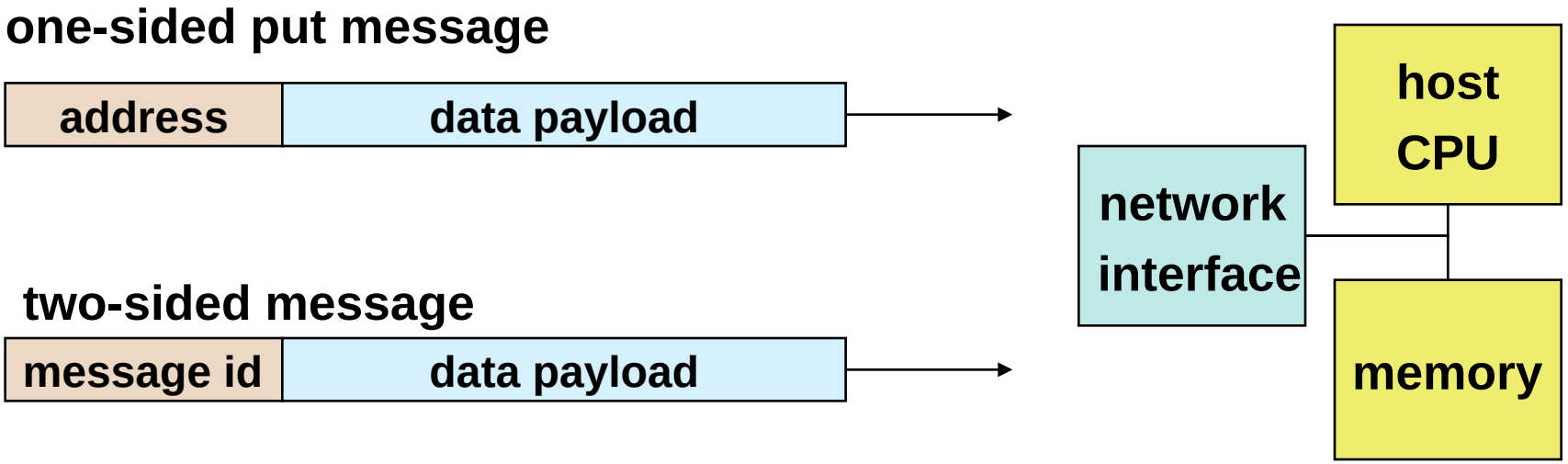
Strategy for acceptance of a new language

- Make it run faster than anything else

Keys to high performance

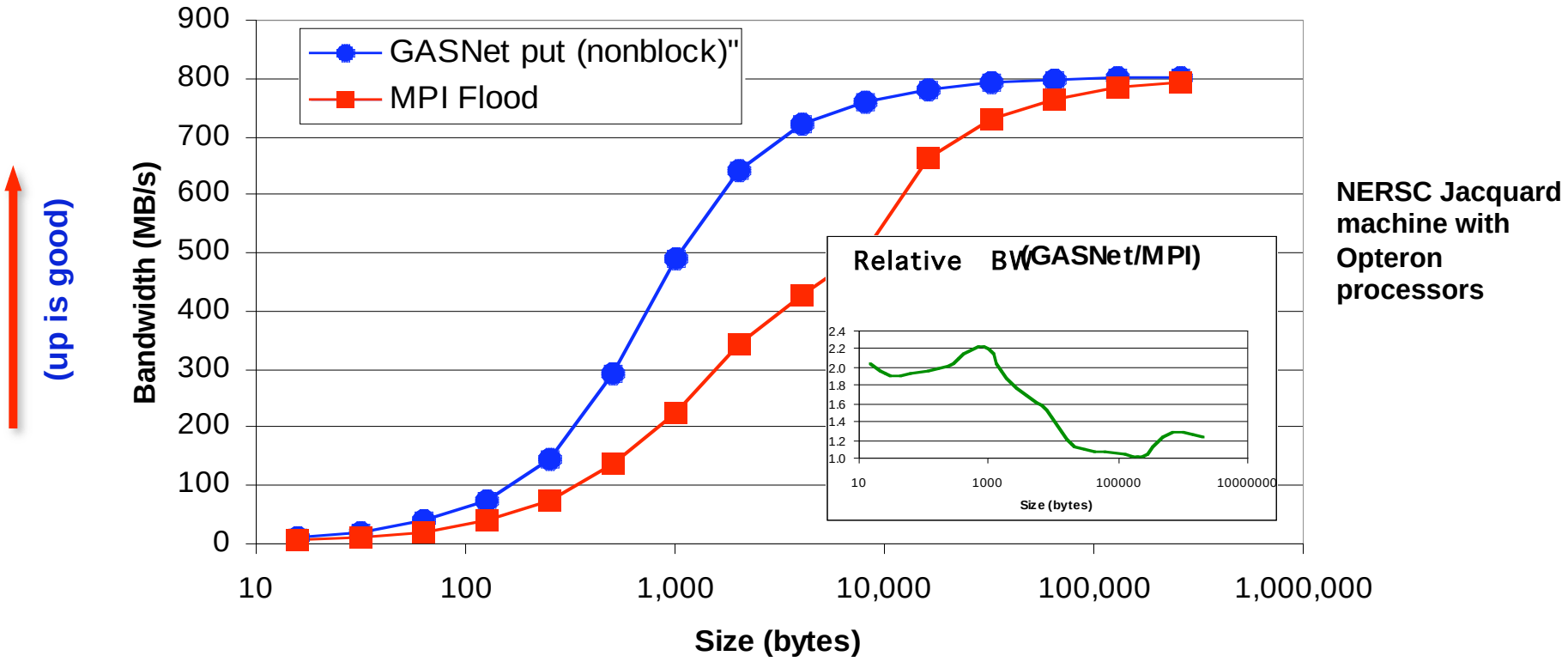
- Parallelism:
  - Scaling the number of processors
- Maximize single node performance
  - Generate friendly code or use tuned libraries (BLAS, FFTW, etc.)
- Avoid (unnecessary) communication cost
  - Latency, bandwidth, overhead
  - Berkeley UPC and Titanium use GASNet communication layer
- Avoid unnecessary delays due to dependencies
  - Load balance; Pipeline algorithmic dependencies

One-Sided vs Two-Sided



- A one-sided put/get message can be handled directly by a network interface with RDMA support
  - Avoid interrupting the CPU or storing data from CPU (preposts)
- A two-sided messages needs to be matched with a receive to identify memory address to put data
  - Offloaded to Network Interface in networks like Quadrics
  - Need to download match tables to interface (from host)
  - Ordering requirements on messages can also hinder bandwidth

One-Sided vs. Two-Sided: Practice

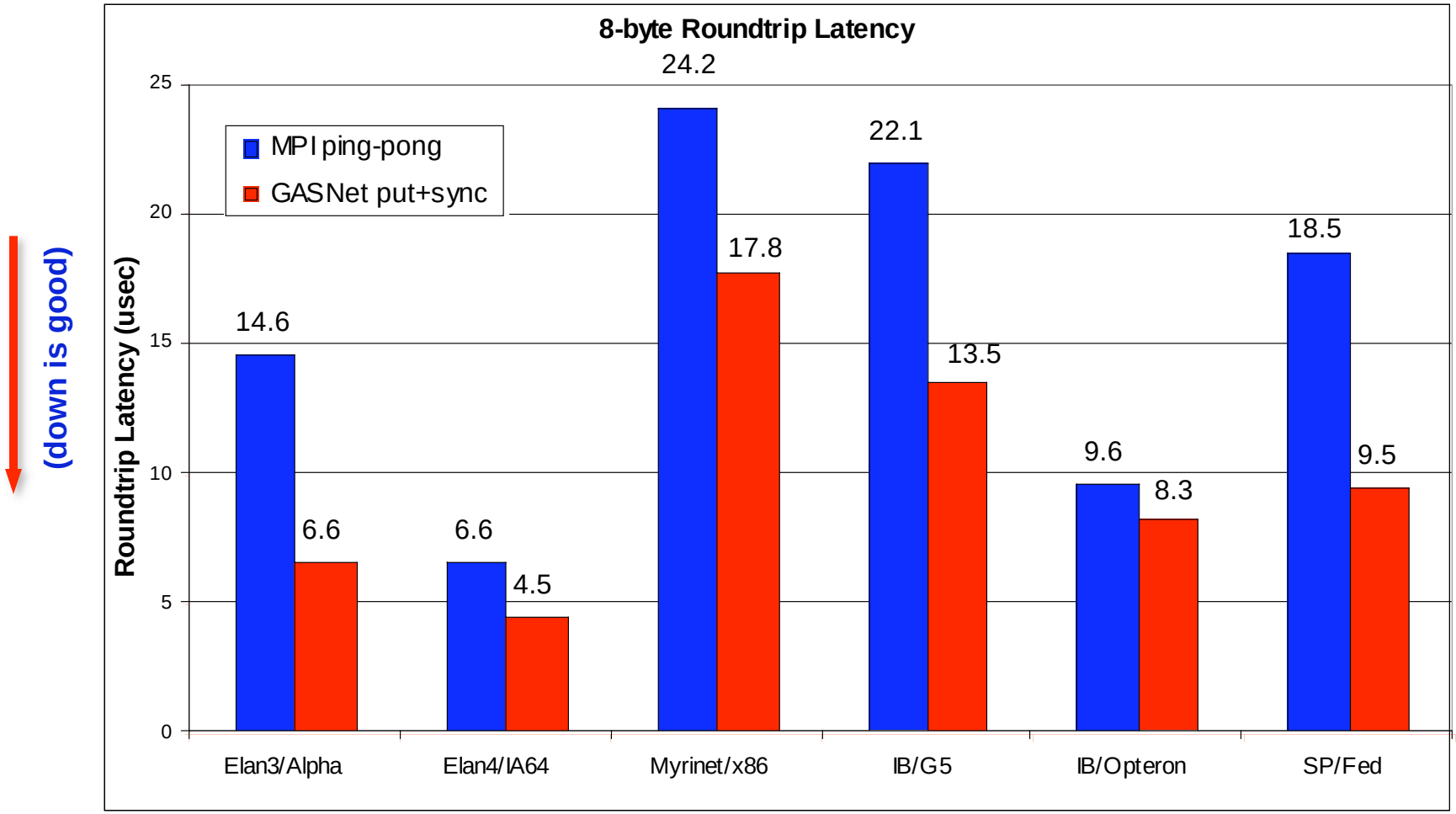


NERSC Jacquard machine with Opteron processors

- InfiniBand: GASNet vapi-conduit and OSU MVAPICH 0.9.5
- Half power point (N ½) differs by *one order of magnitude*
- This is not a criticism of the implementation!

Joint work with Paul Hargrove and Dan Bonachea

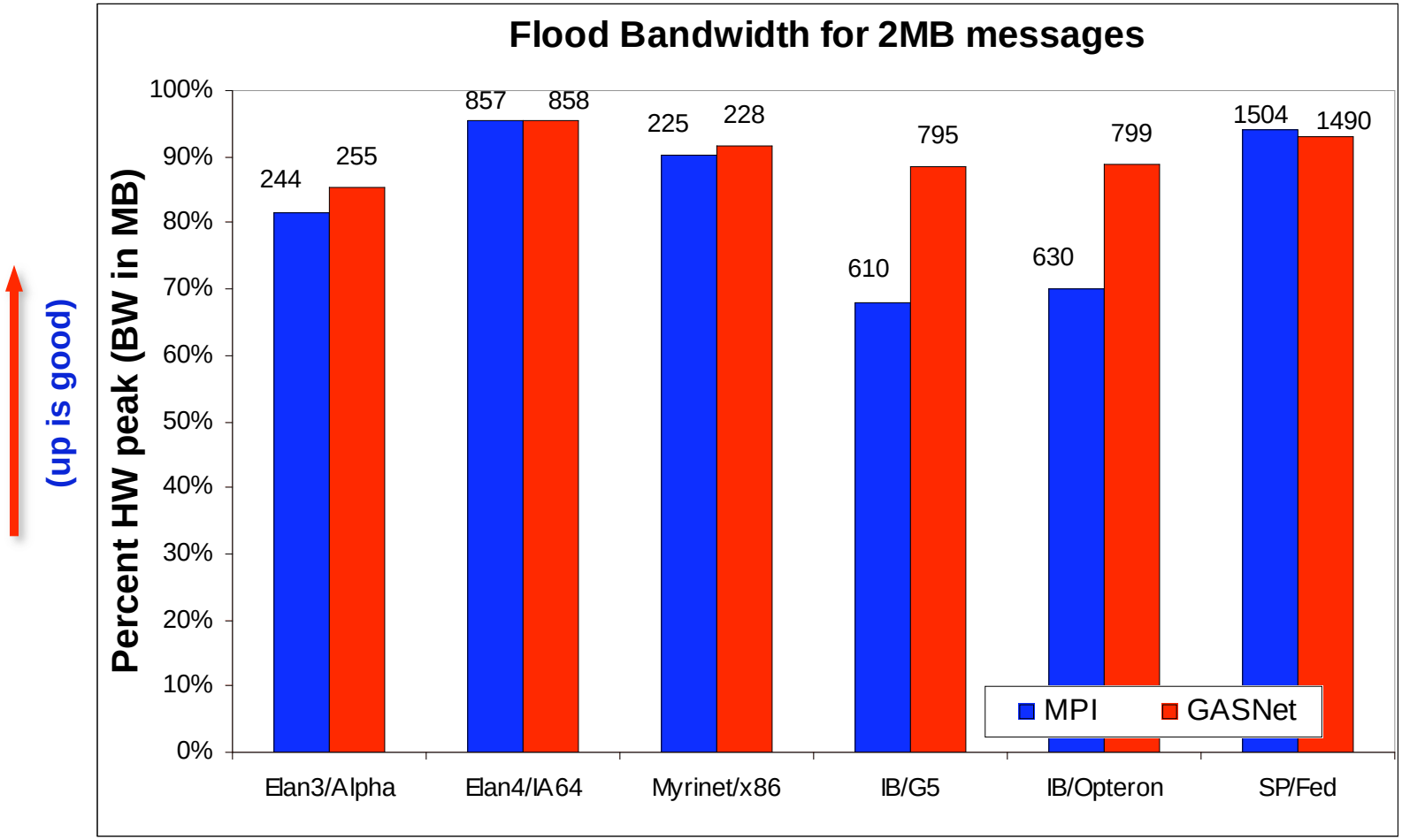
GASNet: Portability *and* High-Performance



GASNet better for latency across machines

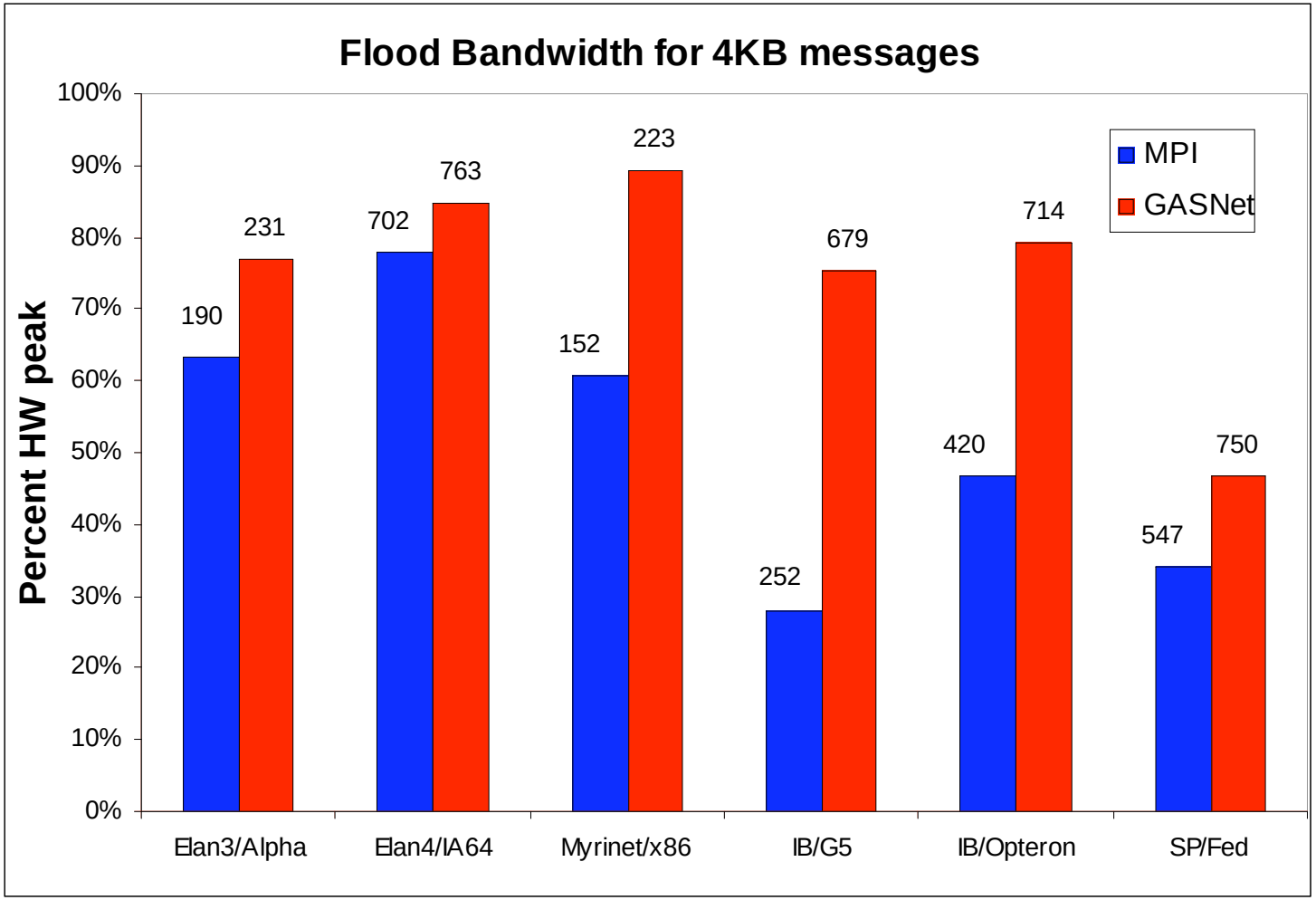
Joint work with UPC Group; GASNet design by Dan Bonachea

GASNet: Portability *and* High-Performance



GASNet at least as high (comparable) for large messages

GASNet: Portability *and* High-Performance



(up is good)

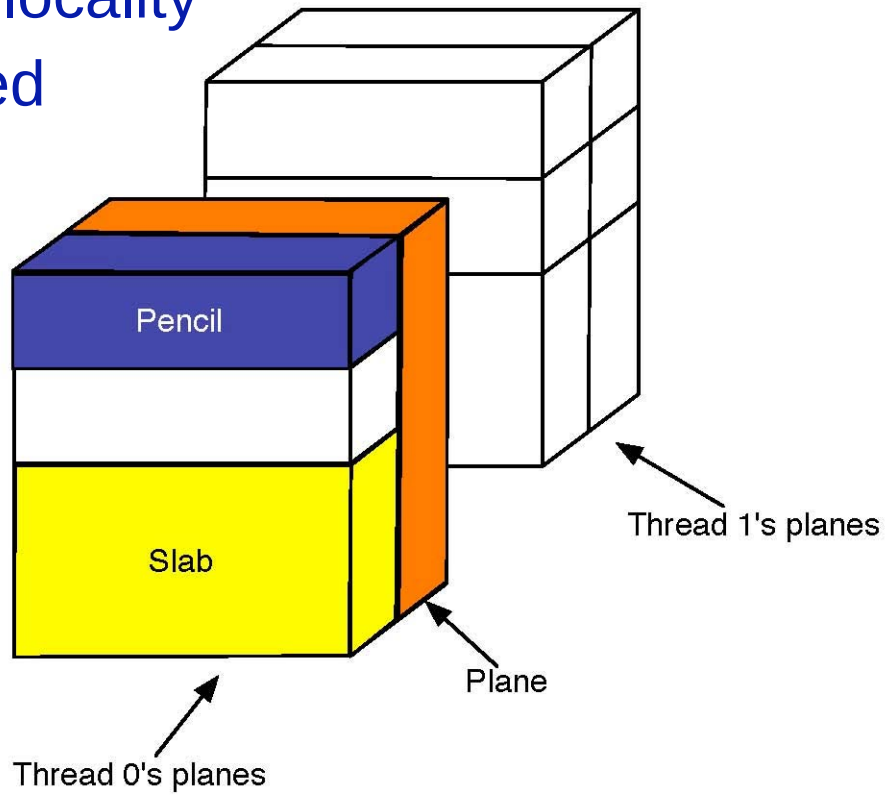
GASNet excels at mid-range sizes: important for overlap

Joint work with UPC Group; GASNet design by Dan Bonachea

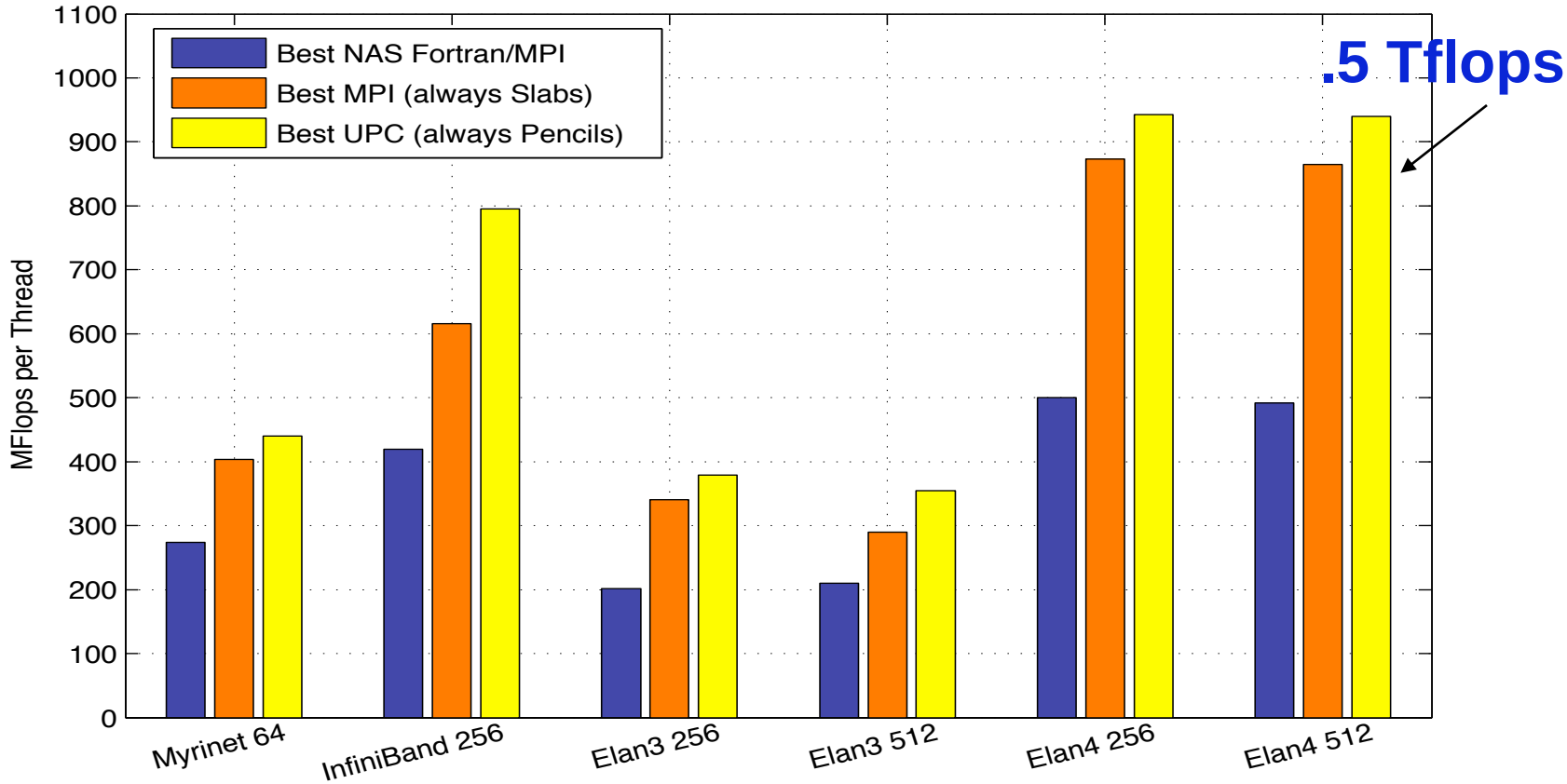
Case Study: NAS FT in UPC

- Perform FFT on a 3D Grid
  - 1D FFTs in each dimension, 3 phases
  - Transpose after first 2 for locality
  - Bisection bandwidth-limited
    - Problem as #procs grows

- Three approaches:
  - **Exchange:**
    - wait for 2nd dim FFTs to finish, send 1 message per processor pair
  - **Slab:**
    - wait for chunk of rows destined for 1 proc, send when ready
  - **Pencil:**
    - send each row as it completes



NAS FT Variants Performance Summary



-
-

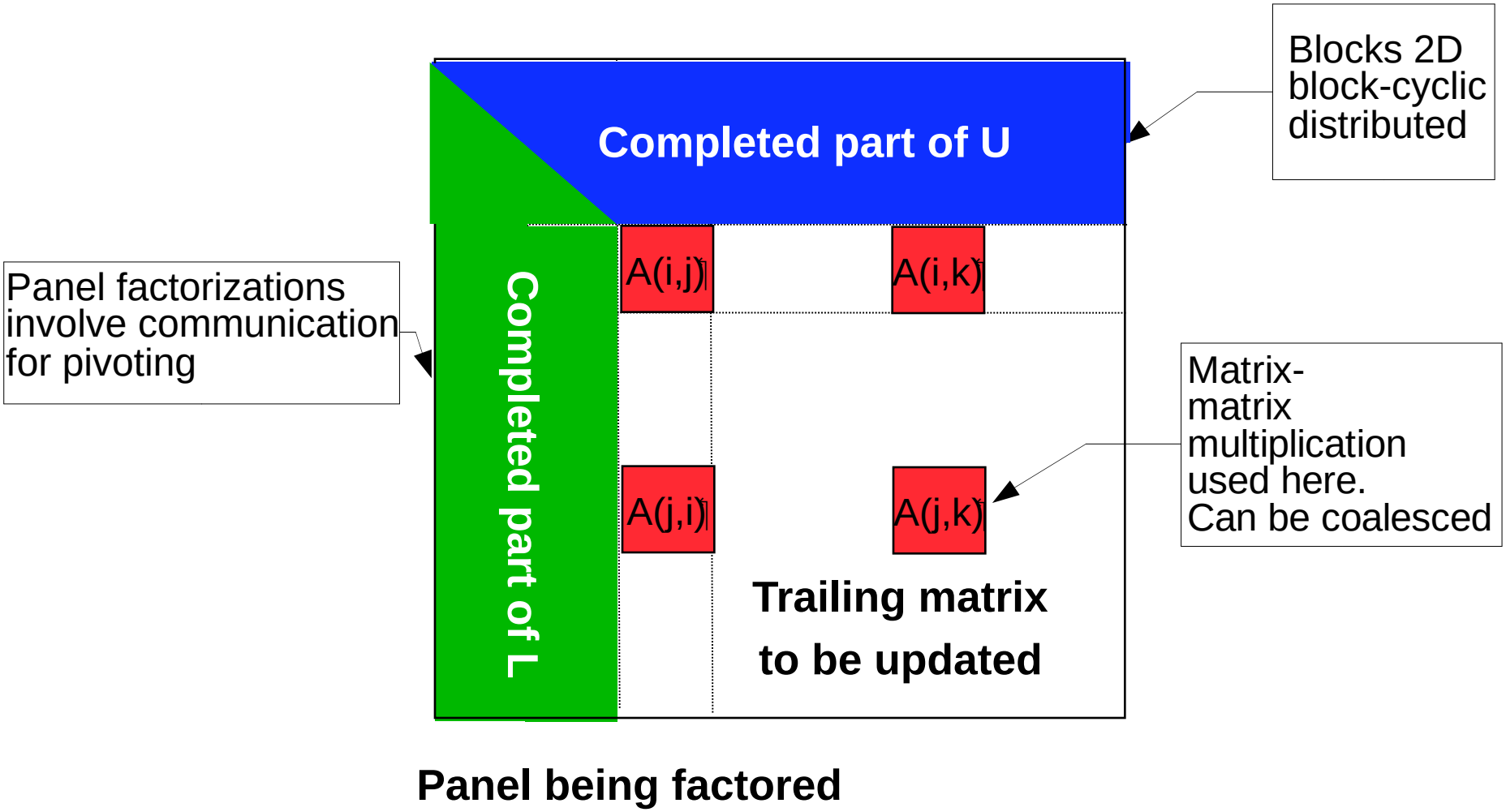
Joint work with Chris Bell, Rajesh Nishtala, Dan Bonachea

Beyond the SPMD Model: Dynamic Threads

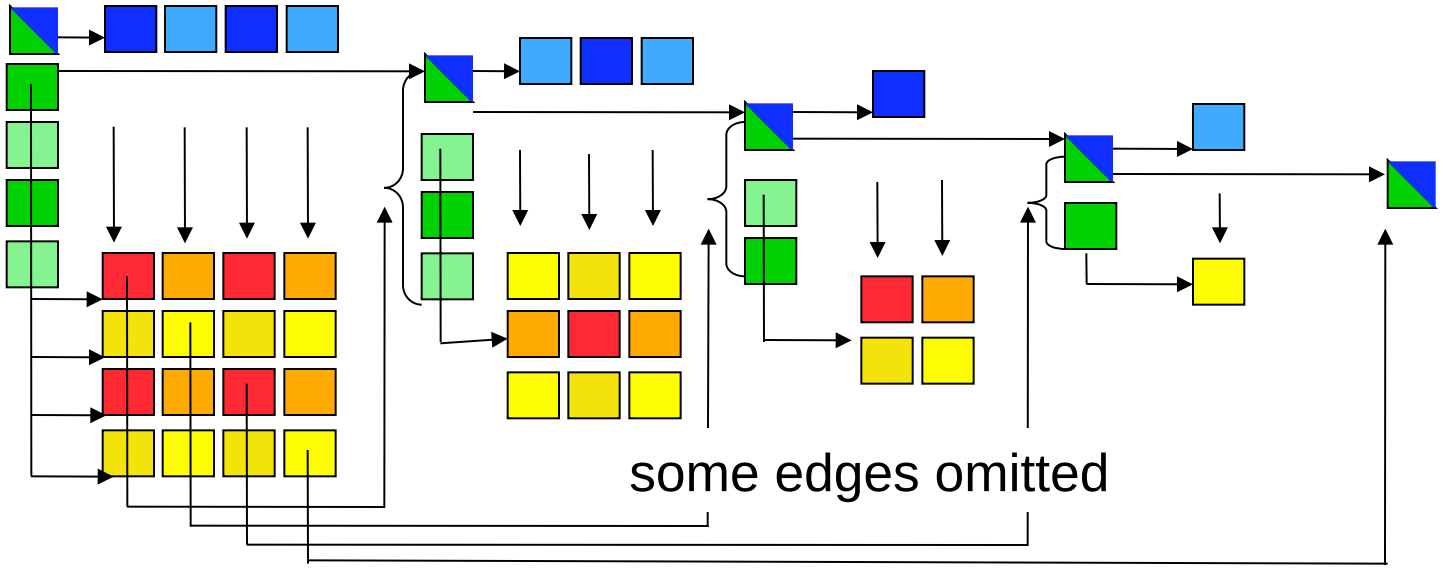
- UPC uses a static threads (SPMD) programming model
  - No dynamic load balancing built-in, although some examples (Delaunay mesh generation) of building it on top
  - Berkeley UPC model extends basic memory semantics (remote read/write) with active messages
  - AM have limited functionality (no messages except acks) to avoid deadlock in the network
- A more dynamic runtime would have many uses
  - Application load imbalance, OS noise, fault tolerance
- Two extremes are well-studied
  - Dynamic load balancing (e.g., random stealing) without locality
  - Static parallelism (with threads = processors) with locality
- Can we combine both in a general-purpose way?

Joint work with Parry Husbands

The Parallel Case

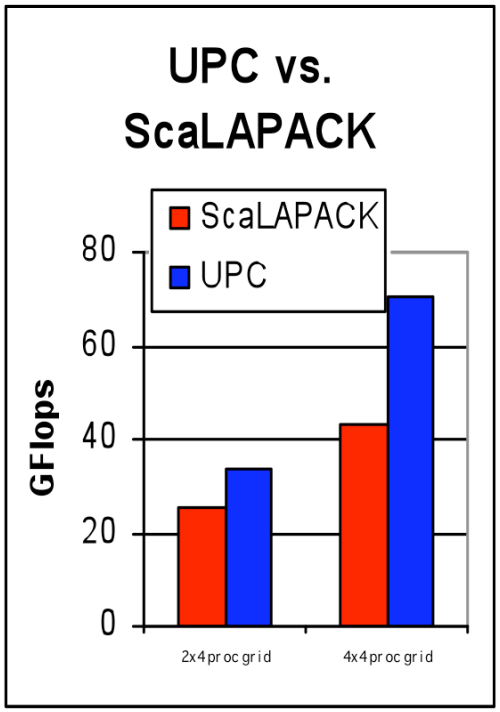


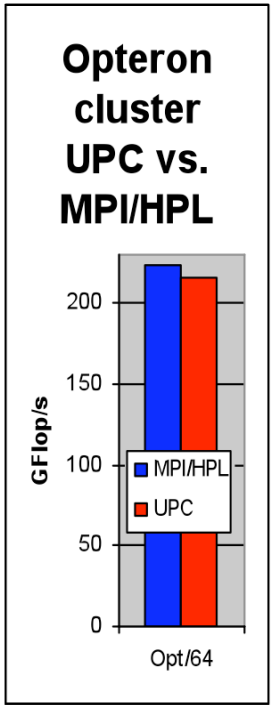
Parallel Tasks in LU

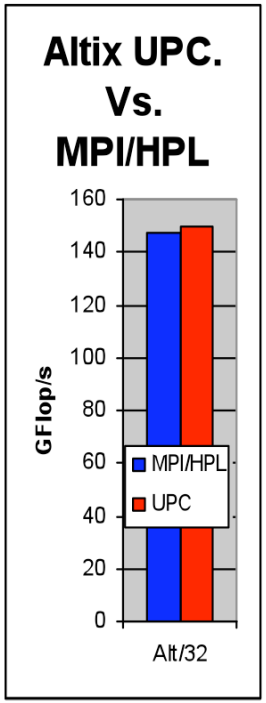


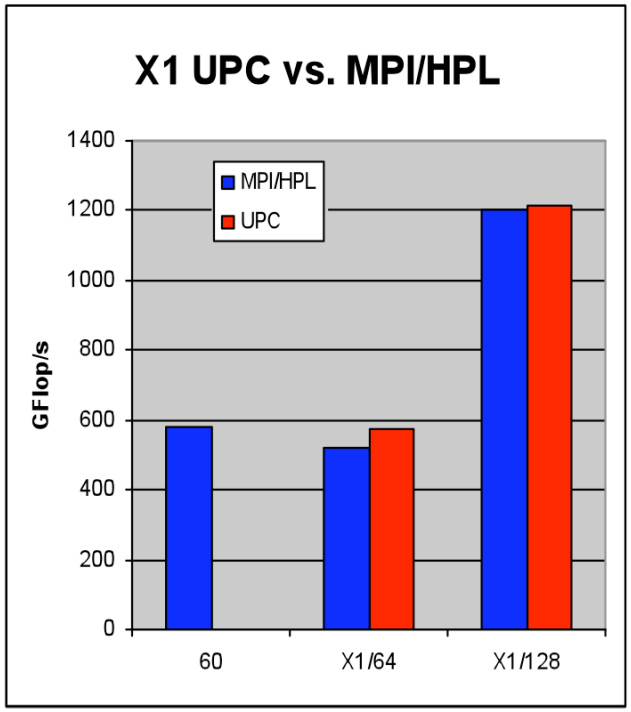
- Implementation uses 3 levels of threading:
  - UPC threads (SPMD), user-level non-preemptive threads, BLAS threads
- Theoretical and practical problem: Memory deadlock
  - Not enough memory for all tasks at once. (Each update needs two temporary blocks, a green and blue, to run.)
  - If updates are scheduled too soon, you will run out of memory
  - If updates are scheduled too late, critical path will be delayed.

UPC HP Linpack Performance





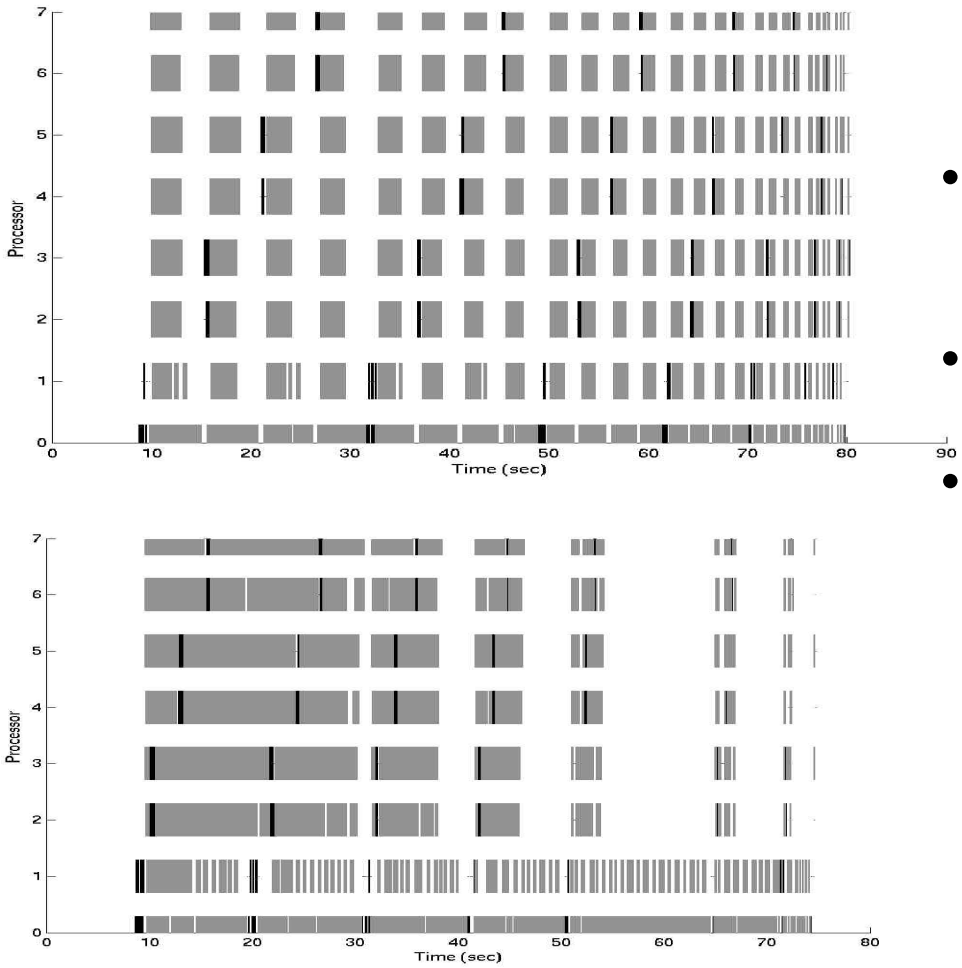




- Faster than ScaLAPACK due to less synchronization
- Comparable to MPI HPL (numbers from HPCC database)
- Large scaling of UPC code on Itanium/Quadrics (Thunder)
  - 2.2 TFlops on 512p and 4.4 TFlops on 1024p

Joint work with Parry Husbands

Utilization Comparison



- Synchronous (above) vs. asynchronous (below) schedule
- SGI Altix Itanium 2 1.4GHz, n=12,800, process grid = 2x4, block size = 400
- Grey blocks = matrix multiplication
- Black blocks = panel factorization

Summary and Discussion

- Message Passing
  - MPI is the de facto programming model for large-scale machines
  - Was developed as a standardization of “known” ideas (but not without controversy)
  - MPI 3.0 standards effort is underway now: you can join!
    - Looking at one-sided communication again
  - Race conditions are relatively rare
- Partitioned Global Address Space Language
  - Offer a compromise on performance and ease of programming
  - Match both shared and distributed memory
  - Demonstrated scalability (like MPI), portability (through GASNet + C)
  - UPC is an example, others include Co-Array Fortran, Titanium (Java)
  - The DARPA HPCS languages: X10, Chapel, Fortress
- Productivity
  - In the eye of the programmer
  - Trade-off: races vs packing/unpacking code

UPC Group (Past and Present)

- Filip Blagojevic
- Dan Bonachea
- Paul Hargrove (Runtime Lead)
- Steve Hofmeyer
- Costin Iancu (Compiler Lead)
- Seung-Jai Min
- Rajesh Nishtala
- Kathy Yelick (Project Lead)
- Yili Zheng

http://upc.lbl.gov

Compiler, runtime, GASNet available here.

Former:

- Christian Bell
- Michael Welcome
- Parry Husbands