ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

User Experiments with PGAS Languages, or

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

User Experiments with PGAS Languages, or It's the Performance, Stupid!

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

User Experiments with PGAS Languages, or

It’s the Performance, Stupid!

Will Sawyer, Sergei Isakov, Adrian Tineo

Overarching goals of our group’s work

Overarching goals of our group's work

Use *scientifically relevant* mini-apps from communities to:

Overarching goals of our group's work

Use *scientifically relevant* mini-apps from communities to:

- Evaluate emerging architectures
  - AMD Interlagos
  - Intel Sandybridge
  - IBM BG/Q, GPUs, Intel MIC, if possible

Overarching goals of our group’s work

Use *scientifically relevant* mini-apps from communities to:

- Evaluate emerging architectures
  - AMD Interlagos
  - Intel Sandybridge
  - IBM BG/Q, GPUs, Intel MIC, if possible
- Evaluate programming paradigms
  - MPI + OpenMP hybrid programming
  - MPI-2 one-sided communication
  - SHMEM
  - PGAS languages (CAF, UPC)
  - OpenACC, CUDA, OpenCL, if possible

Overarching goals of our group’s work

Use *scientifically relevant* mini-apps from communities to:

- Evaluate emerging architectures
  - AMD Interlagos
  - Intel Sandybridge
  - IBM BG/Q, GPUs, Intel MIC, if possible
- Evaluate programming paradigms
  - MPI + OpenMP hybrid programming
  - MPI-2 one-sided communication
  - SHMEM
  - PGAS languages (CAF, UPC)
  - OpenACC, CUDA, OpenCL, if possible
- Compare performance across platforms
  - out-of-the-box performance
  - evaluate optimization effort
  - socket-for-socket, node-for-node, energy-to-solution comparisons

Important concepts and acronyms

Important concepts and acronyms

- PGAS: Partitioned Global Address Space

Important concepts and acronyms

- PGAS: Partitioned Global Address Space
- UPC: Unified Parallel C

Important concepts and acronyms

- PGAS: Partitioned Global Address Space
- UPC: Unified Parallel C
- CAF: Co-Array Fortran

Important concepts and acronyms

- PGAS: Partitioned Global Address Space
- UPC: Unified Parallel C
- CAF: Co-Array Fortran
- Titanium: PGAS Java dialect

Important concepts and acronyms

- PGAS: Partitioned Global Address Space
- UPC: Unified Parallel C
- CAF: Co-Array Fortran
- Titanium: PGAS Java dialect
- MPI: Message-Passing Interface

Important concepts and acronyms

- PGAS: Partitioned Global Address Space
- UPC: Unified Parallel C
- CAF: Co-Array Fortran
- Titanium: PGAS Java dialect
- MPI: Message-Passing Interface
- SHMEM: Shared Memory API (SGI)

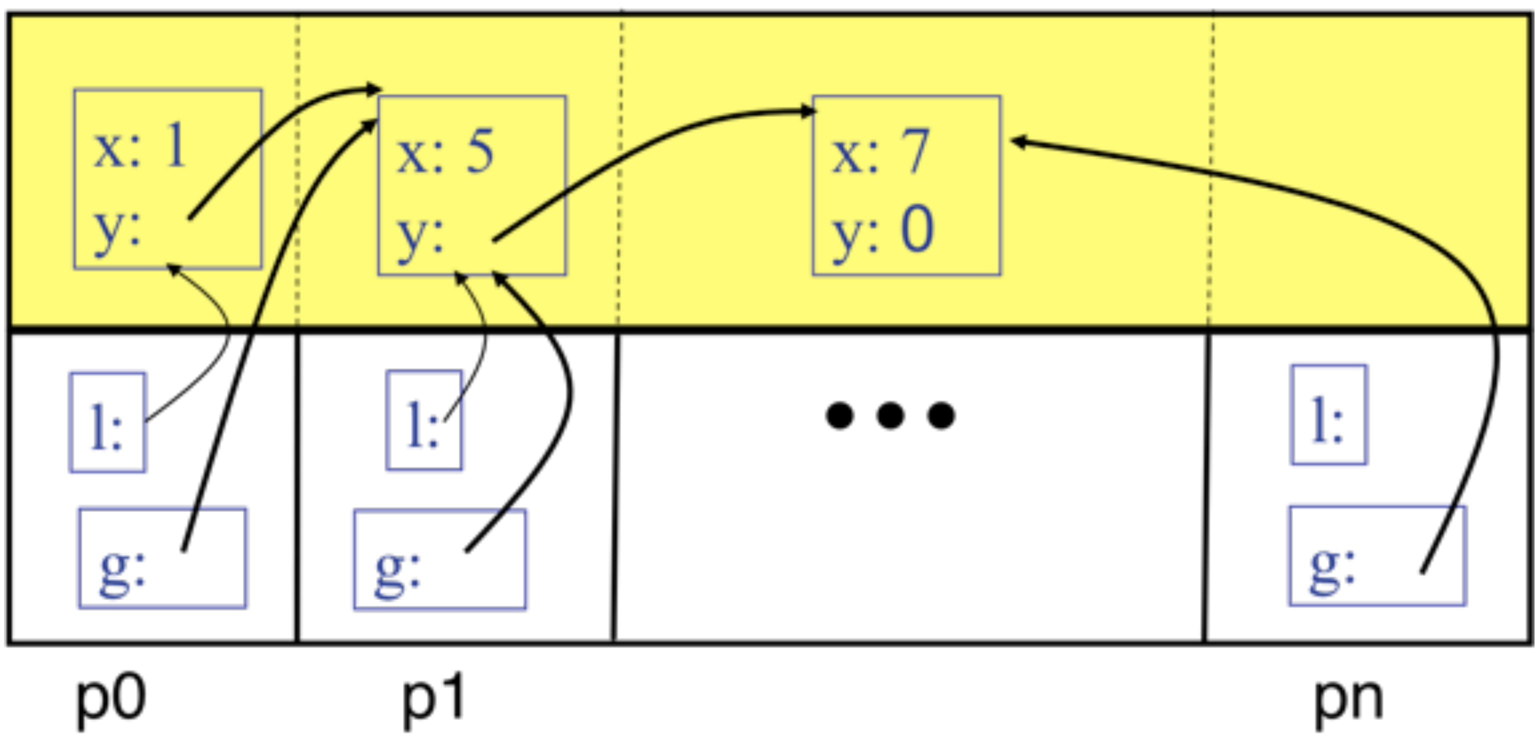
Partitioned Global Address Space

Partitioned Global Address Space

- *Global address space:* any thread/process may directly read/write data allocated by any other
- *Partitioned:* data is designated as local or global; programmer controls layout

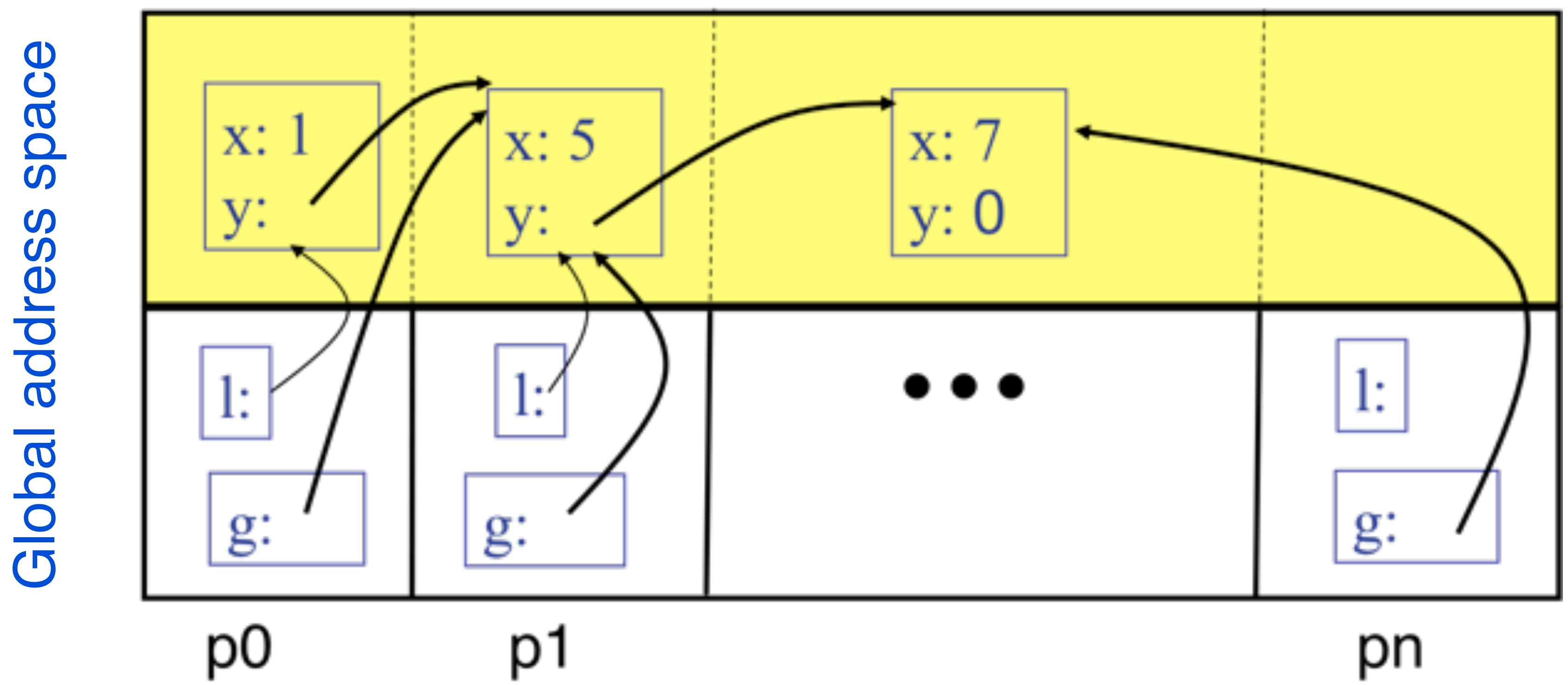
Partitioned Global Address Space

- *Global address space:* any thread/process may directly read/write data allocated by any other
- *Partitioned:* data is designated as local or global; programmer controls layout



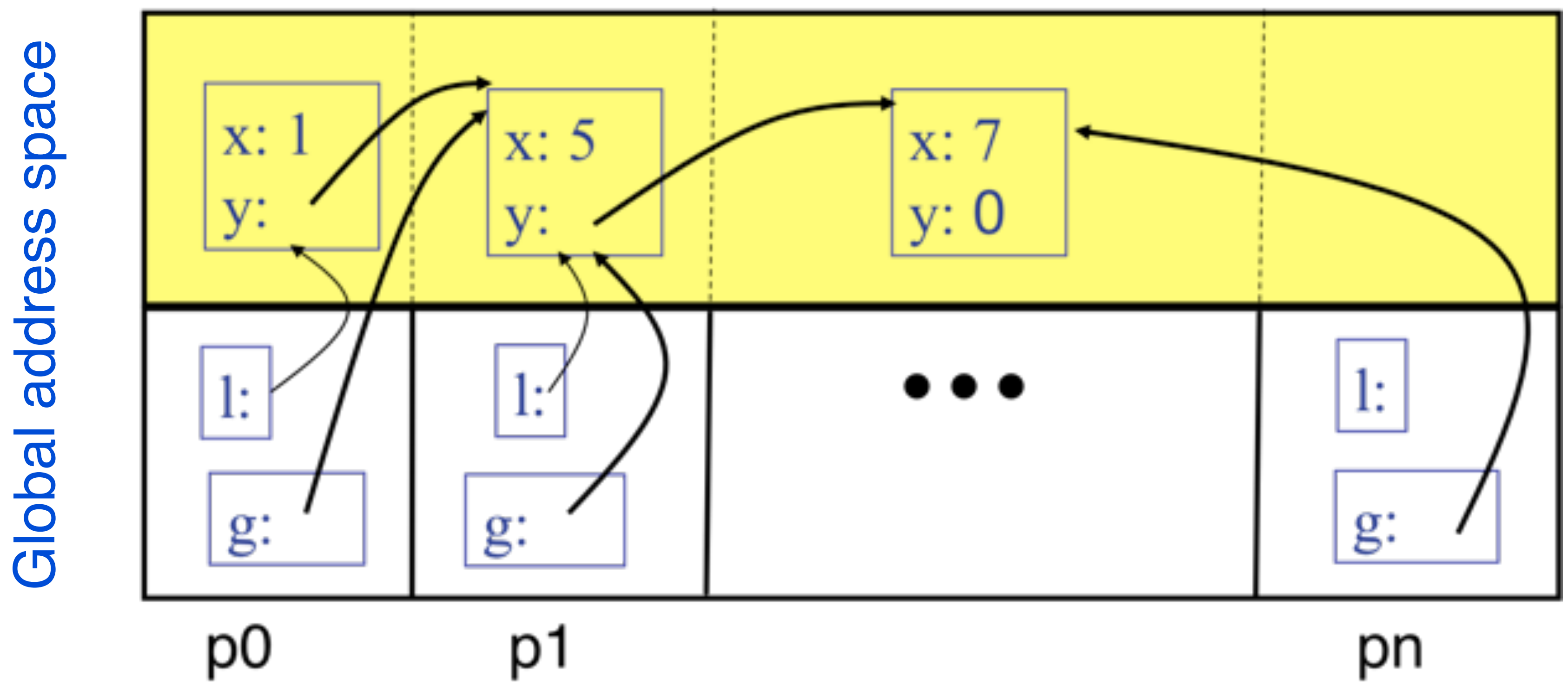
Partitioned Global Address Space

- *Global address space:* any thread/process may directly read/write data allocated by any other
- *Partitioned:* data is designated as local or global; programmer controls layout



Partitioned Global Address Space

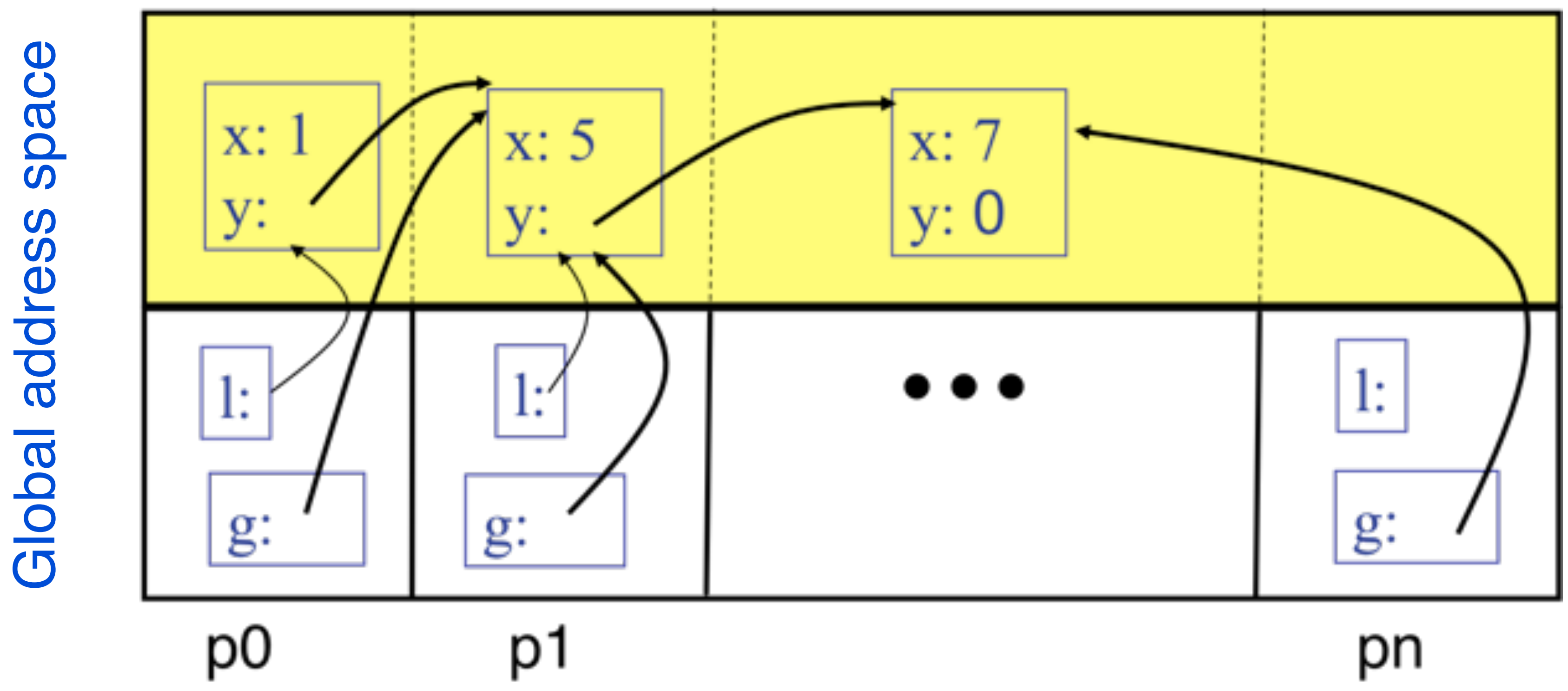
- *Global address space:* any thread/process may directly read/write data allocated by any other
- *Partitioned:* data is designated as local or global; programmer controls layout



By default:
- object heaps are shared
- program stacks are private

Partitioned Global Address Space

- *Global address space:* any thread/process may directly read/write data allocated by any other
- *Partitioned:* data is designated as local or global; programmer controls layout



By default:
- object heaps are shared
- program stacks are private

3 Current languages: UPC, CAF, and Titanium

Potential strengths of a PGAS language

Potential strengths of a PGAS language

- Interprocess communication intrinsic to language
  - Explicit support for distributed data structures (private and shared data)
  - Conceptually the parallel formulation can be more elegant

Potential strengths of a PGAS language

- Interprocess communication intrinsic to language
  - Explicit support for distributed data structures (private and shared data)
  - Conceptually the parallel formulation can be more elegant
- One-sided communication
  - Values are either ‘put’ or ‘got’ from remote images
  - Support for bulk messages, synchronization
  - Could be implemented with message-passing library or through RDMA (remote direct memory access)

Potential strengths of a PGAS language

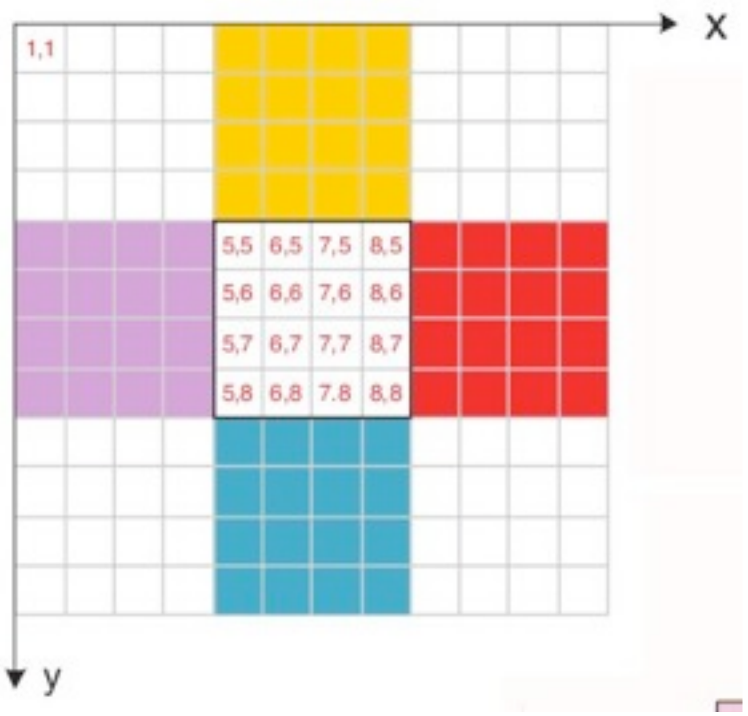
- Interprocess communication intrinsic to language
  - Explicit support for distributed data structures (private and shared data)
  - Conceptually the parallel formulation can be more elegant
- One-sided communication
  - Values are either 'put' or 'got' from remote images
  - Support for bulk messages, synchronization
  - Could be implemented with message-passing library or through RDMA (remote direct memory access)
- PGAS hardware support available
  - Cray Gemini (XE6) interconnect supports RDMA

Potential strengths of a PGAS language

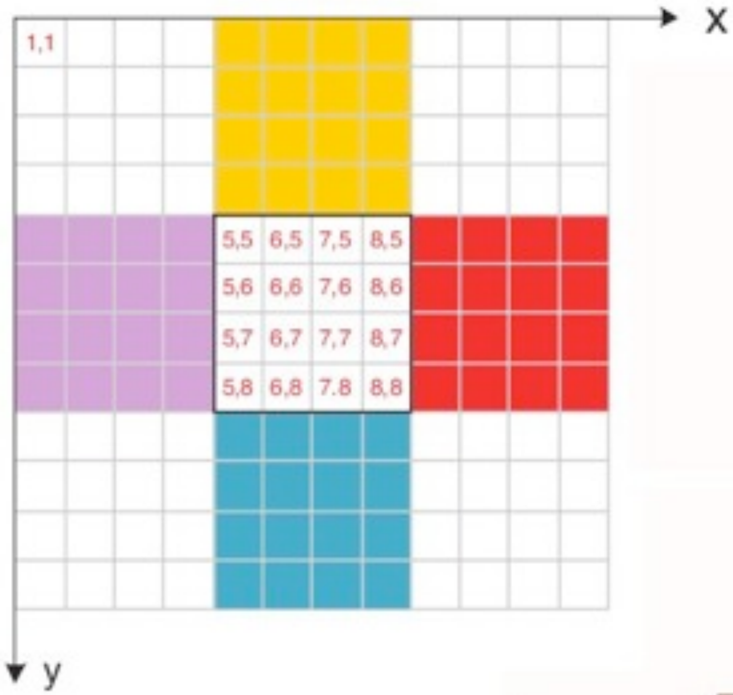
- Interprocess communication intrinsic to language
  - Explicit support for distributed data structures (private and shared data)
  - Conceptually the parallel formulation can be more elegant
- One-sided communication
  - Values are either ‘put’ or ‘got’ from remote images
  - Support for bulk messages, synchronization
  - Could be implemented with message-passing library or through RDMA (remote direct memory access)
- PGAS hardware support available
  - Cray Gemini (XE6) interconnect supports RDMA
- Potential interoperability with existing C/Fortran/Java code

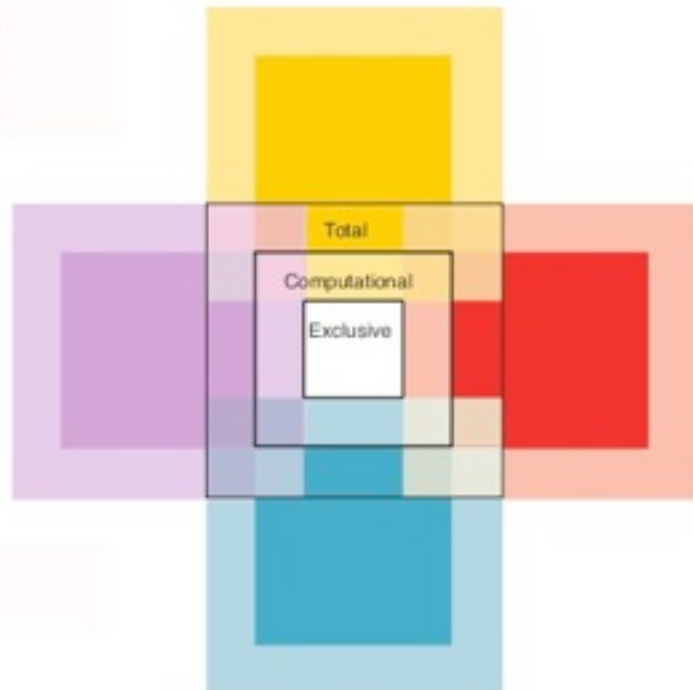
Problem 1: Halo Exchange

Problem 1: Halo Exchange

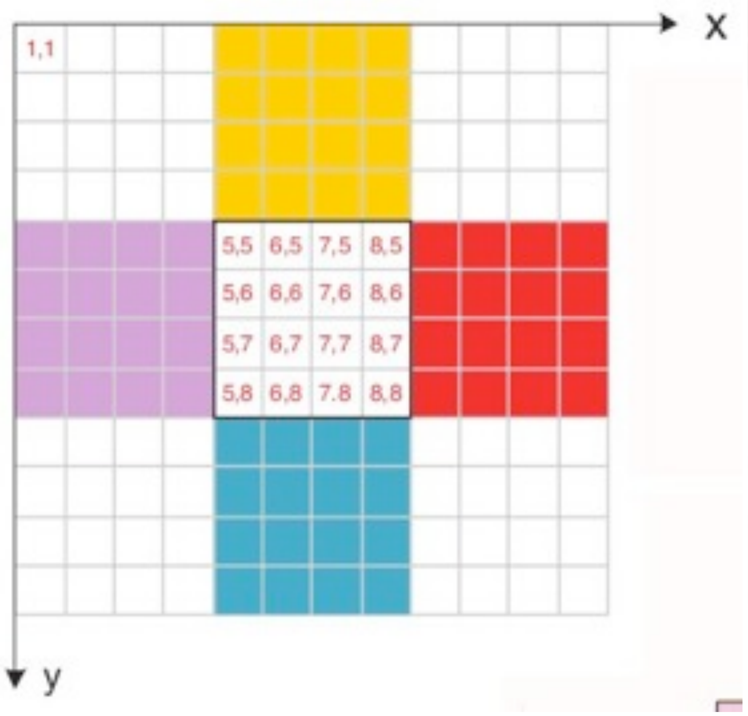


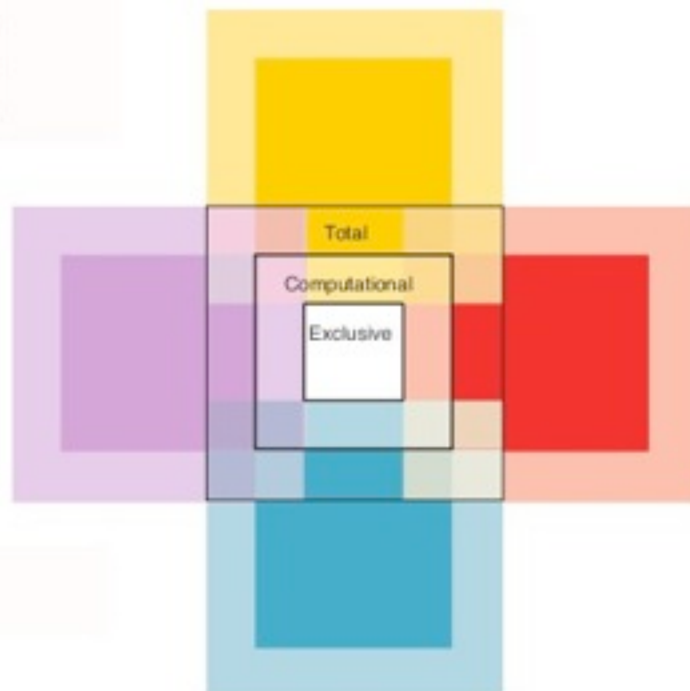
Problem 1: Halo Exchange

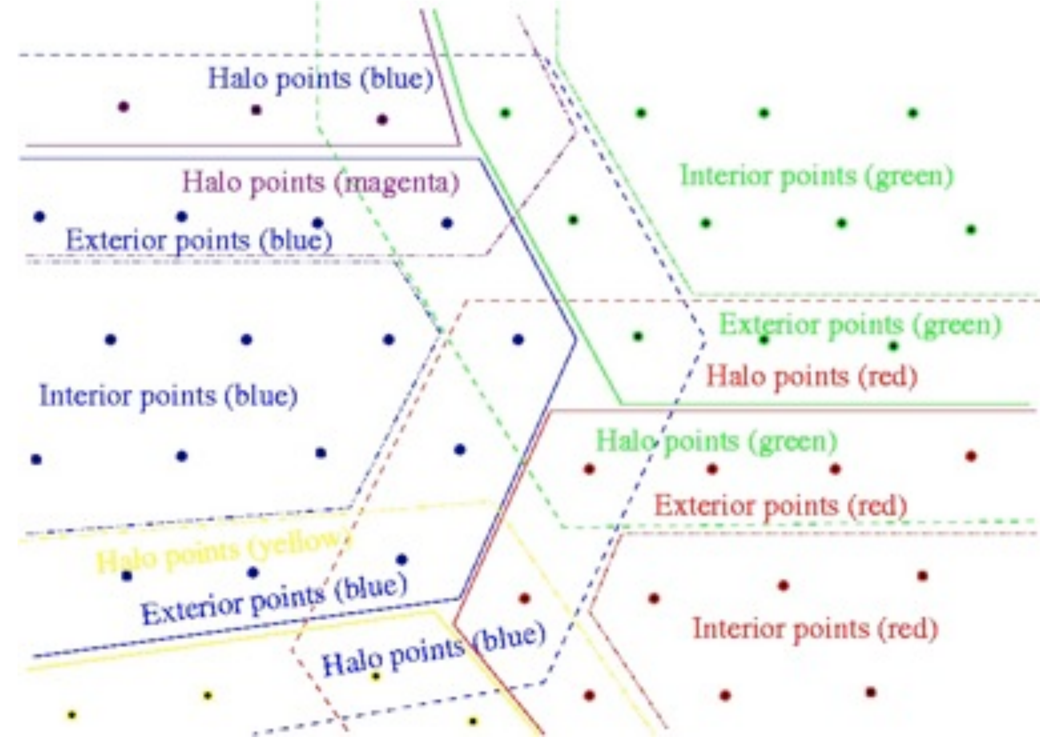




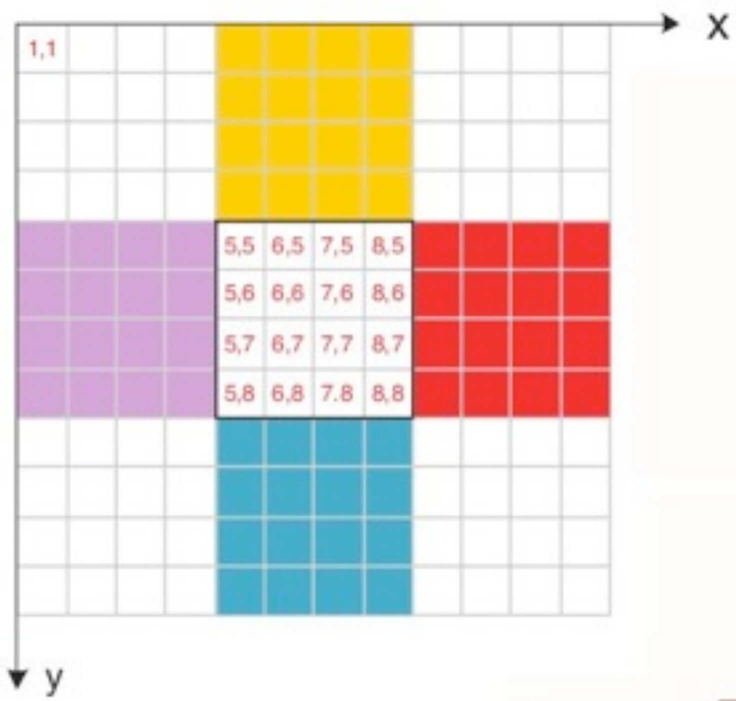
Problem 1: Halo Exchange

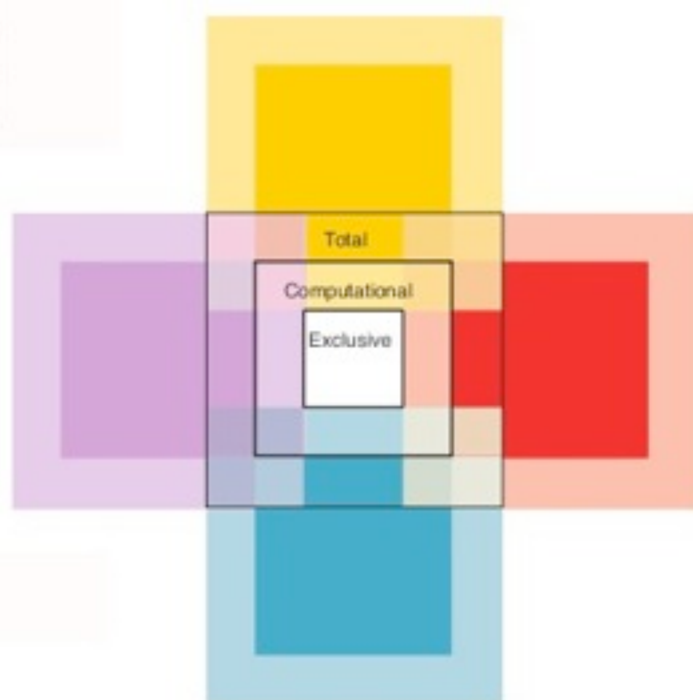


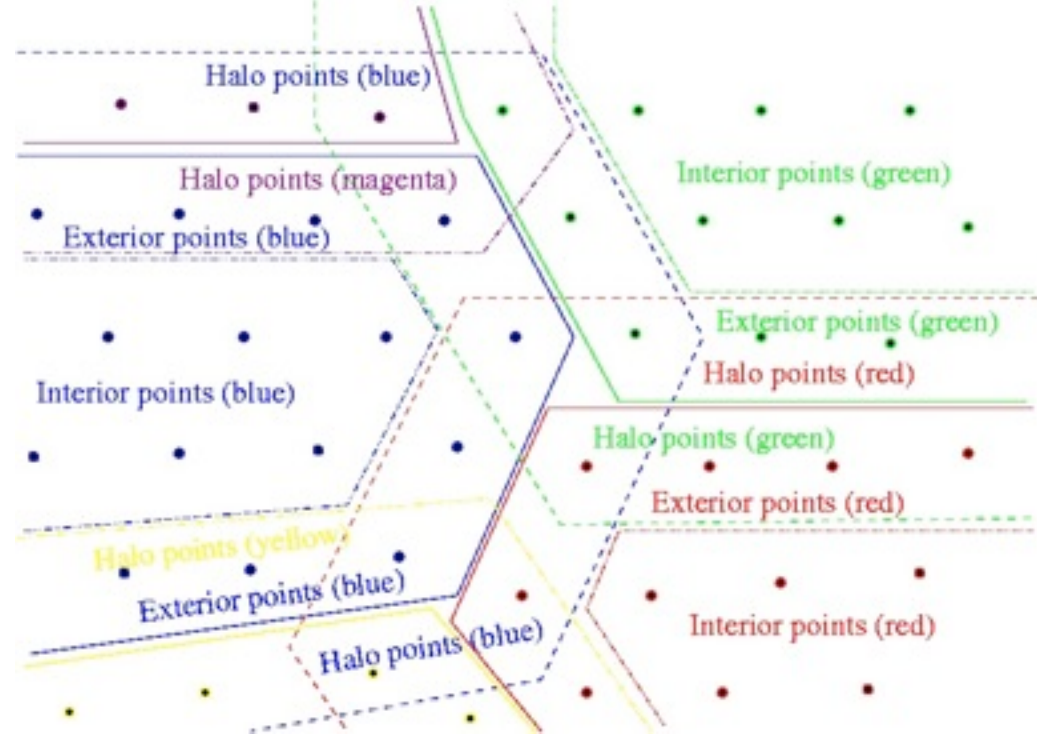


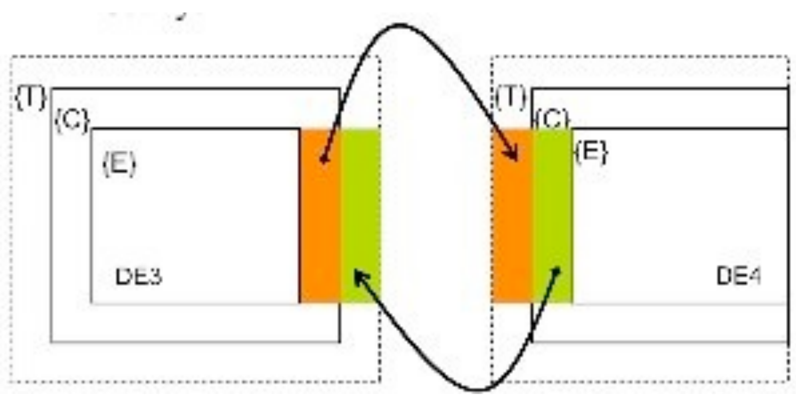


Problem 1: Halo Exchange









Potential Performance Gains with Co-Array Fortran

Potential Performance Gains with Co-Array Fortran

The Performance Evolution of the Parallel Ocean Program on the Cray X1 *

P. H. Worley †
Oak Ridge National Laboratory

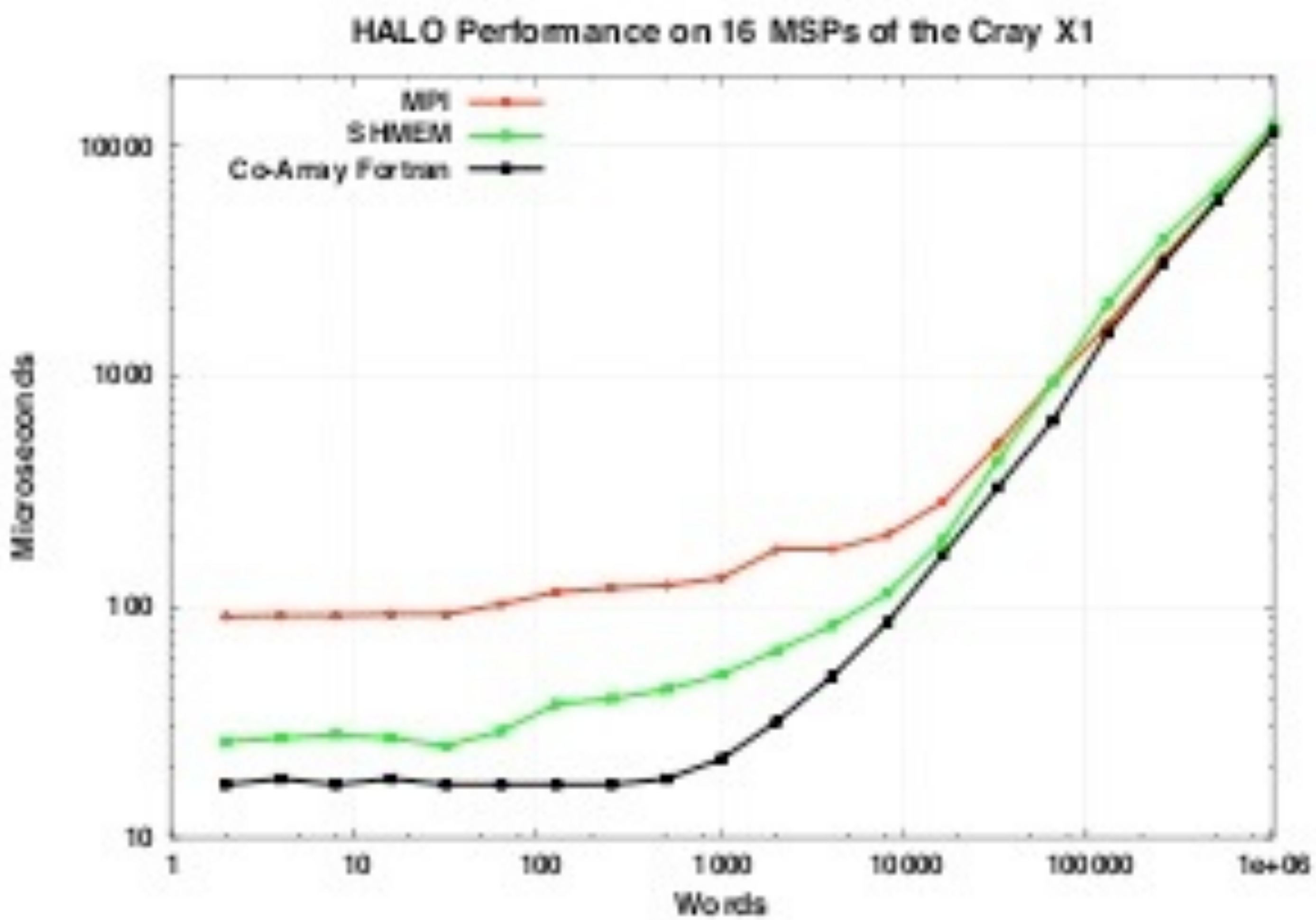
J. Levesque ‡
Cray Inc.

Potential Performance Gains with Co-Array Fortran

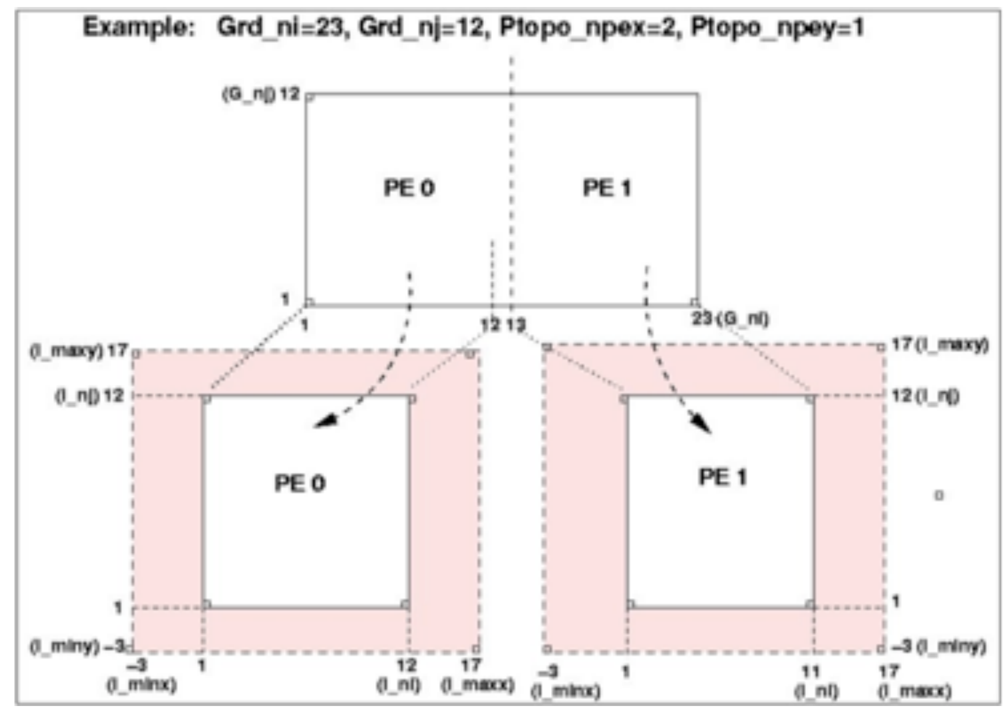
The Performance Evolution of the Parallel Ocean Program on the Cray X1 *

P. H. Worley †
Oak Ridge National Laboratory

J. Levesque ‡
Cray Inc.



Halo Exchange “Stencil 2D” Benchmark

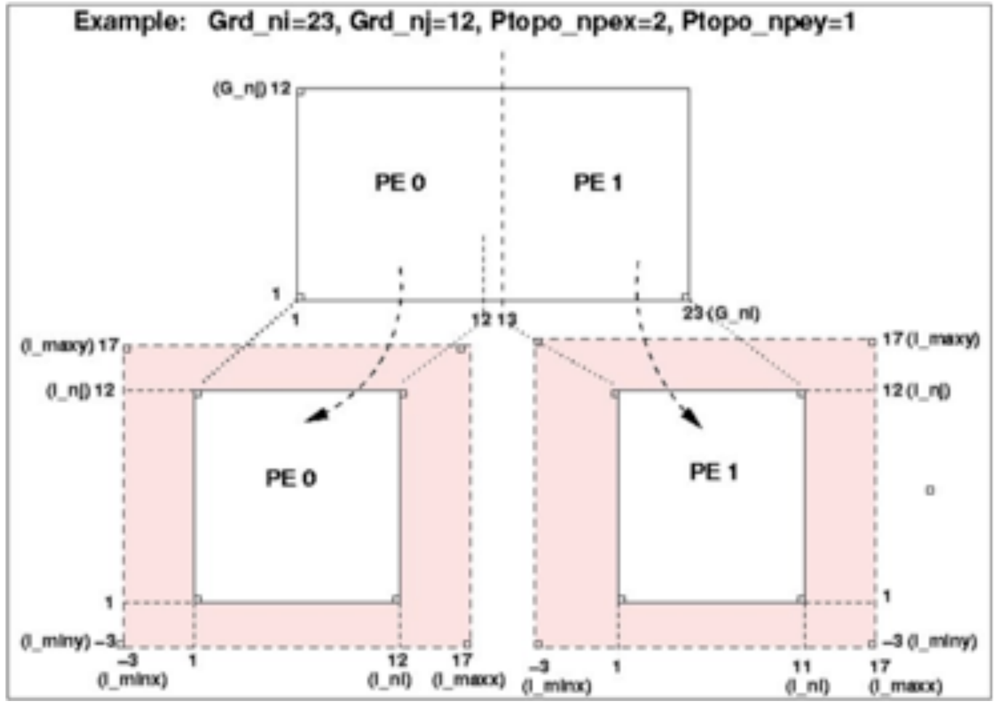


Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Halo Exchange “Stencil 2D” Benchmark

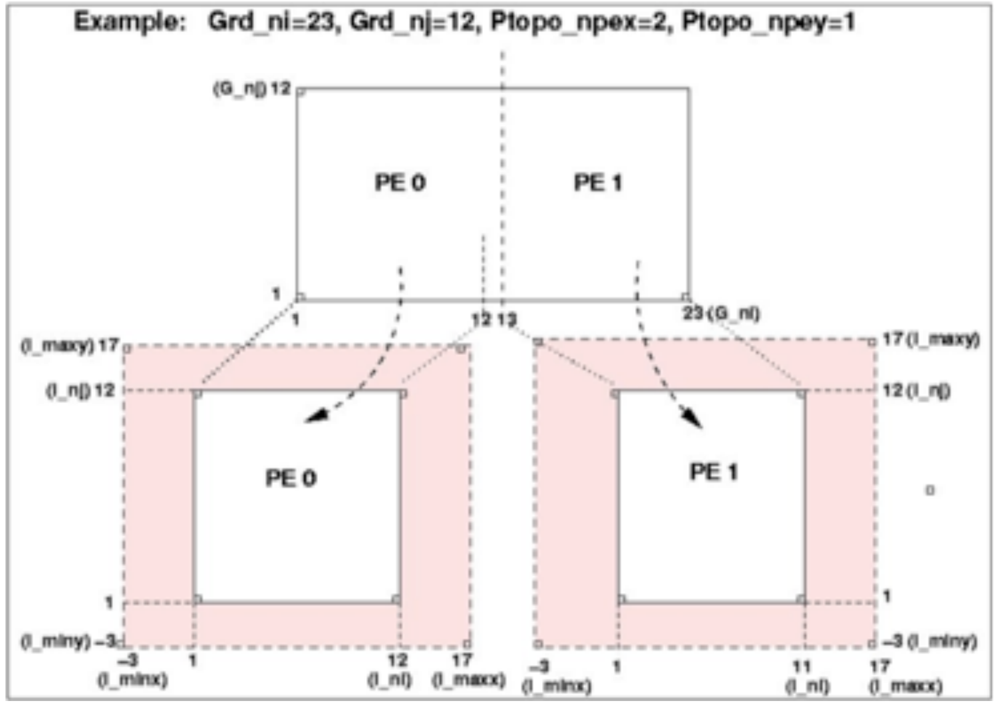
Halo exchange and stencil operation over a square domain distributed over a 2-D virtual process topology



Halo Exchange “Stencil 2D” Benchmark

Halo exchange and stencil operation over a square domain distributed over a 2-D virtual process topology

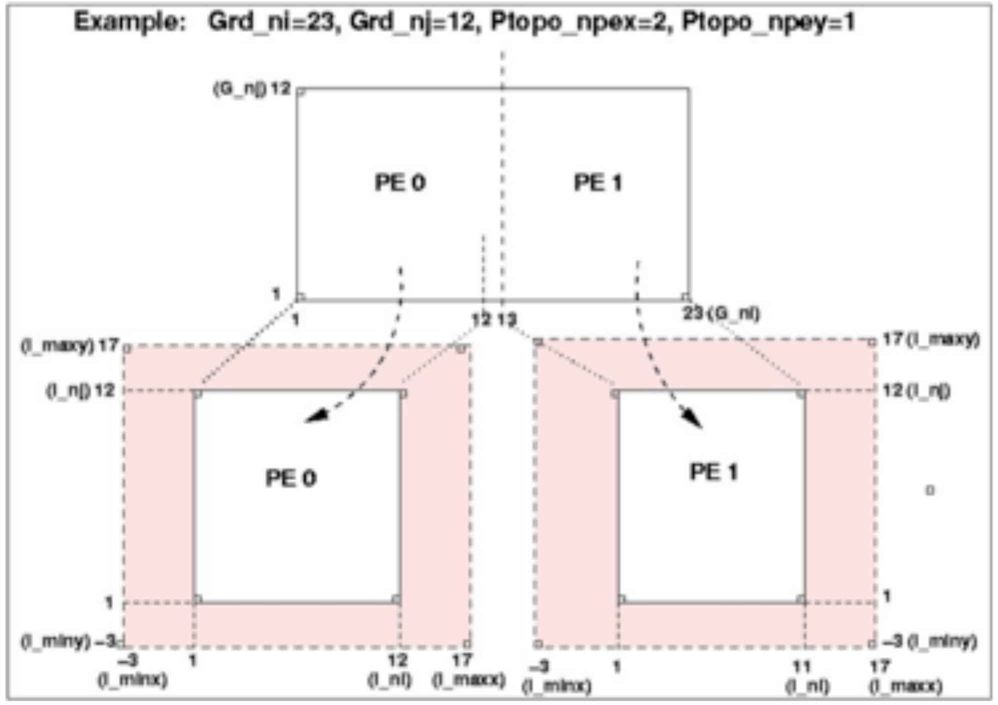
- Arbitrary halo ‘radius’ (number of halo cells in a given dimension, e.g. 3)



Halo Exchange “Stencil 2D” Benchmark

Halo exchange and stencil operation over a square domain distributed over a 2-D virtual process topology

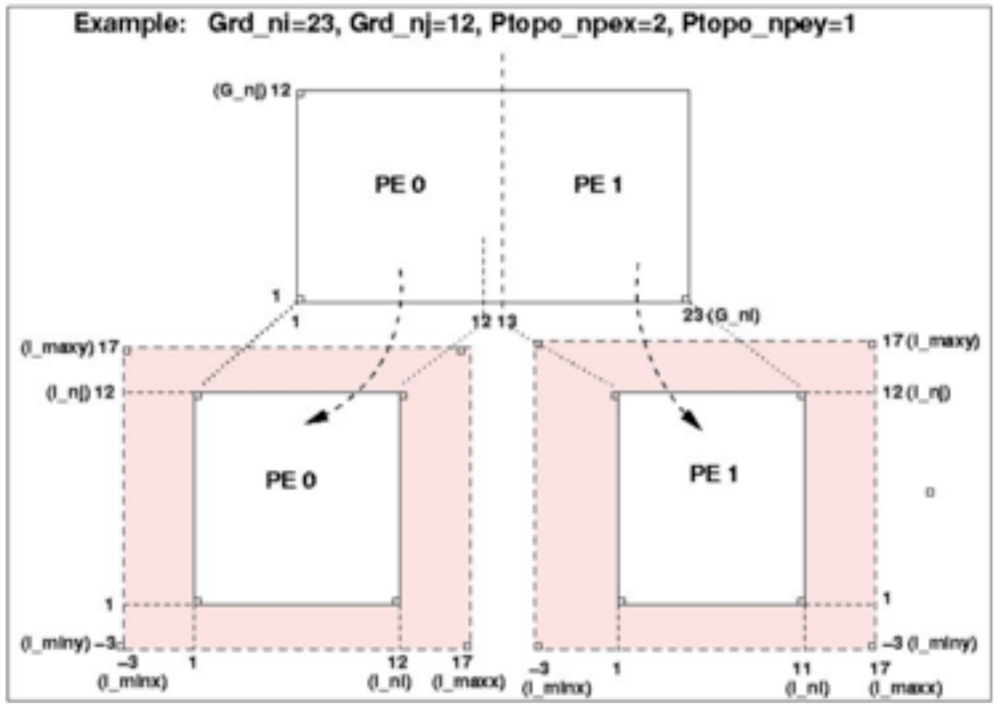
- Arbitrary halo ‘radius’ (number of halo cells in a given dimension, e.g. 3)
- MPI implementations:
  - Trivial: post all 8 MPI_Isend and Irecv
  - Sendrecv: MPI_Sendrecv between PE pairs
  - Halo: MPI_Isend/Irecv between PE pairs



Halo Exchange “Stencil 2D” Benchmark

Halo exchange and stencil operation over a square domain distributed over a 2-D virtual process topology

- Arbitrary halo ‘radius’ (number of halo cells in a given dimension, e.g. 3)
- MPI implementations:
  - Trivial: post all 8 MPI_Isend and Irecv
  - Sendrecv: MPI_Sendrecv between PE pairs
  - Halo: MPI_Isend/Irecv between PE pairs
- CAF implementations:
  - Trivial: simple copies to remote images
  - Put: reciprocal puts between image pairs
  - Get: reciprocal gets between image pairs
  - Get0: all images do inner region first, then all do block region (fine grain, no sync.)
  - Get1: half of images do inner region first,half do block region first (fine grain, no sync.)



Example code: Trivial CAF

Example code: Trivial CAF

```
     real, allocatable, save :: V(:,:)[:,:]
  :
 allocate( V(1-halo:m+halo,1-halo:n+halo)[p,*] )
  :
 WW = myP-1 ; if (WW<1) WW = p
 EE = myP+1 ; if (EE>p) EE = 1
 SS = myQ-1 ; if (SS<1) SS = q
 NN = myQ+1 ; if (NN>q) NN = 1
   :

 V(1:m,1:n)                        = dom(1:m,1:n)               !  computational region

 V(1-halo:0, 1:n)[EE,myQ]          = dom(m-halo+1:m,1:n)        !  to East
 V(m+1:m+halo, 1:n)[WW,myQ]        = dom(1:halo,1:n)            !  to West
 V(1:m,1-halo:0)[myP,NN]           = dom(1:m,n-halo+1:n)        !  to North
 V(1:m,n+1:n+halo)[myP,SS]         = dom(1:m,1:halo)            !  to South
 V(1-halo:0,1-halo:0)[EE,NN]       = dom(m-halo+1:m,n-halo+1:n) !  to North-East
 V(m+1:m+halo,1-halo:0)[WW,NN]     = dom(1:halo,n-halo+1:n)     !  to North-West
 V(1-halo:0,n+1:n+halo)[EE,SS]     = dom(m-halo+1:m,1:halo)     !  to South-East
 V(m+1:m+halo,n+1:n+halo)[WW,SS]   = dom(1:halo,1:halo)         !  to South-West

 sync all
!
! Now run a stencil filter over the computational region (the region unaffected by halo values)
!
 do j=1,n
   do i=1,m
     sum = 0.
     do l=-halo,halo
       do k=-halo,halo
         sum = sum + stencil(k,l)*V(i+k,j+l)
       enddo
     enddo
     dom(i,j) = sum
   enddo
 enddo
```

Example code: CAF Put

Example code: CAF Put

```
      :
  V(1:m,1:n)                       = dom(1:m,1:n)               !  internal region

  V(1-halo:0, 1:n)[EE,myQ]         = dom(m-halo+1:m,1:n)        !  to East
   :
  V(m+1:m+halo,n+1:n+halo)[WW,SS]  = dom(1:halo,1:halo)         !  to South-West

! NO GLOBAL SYNCHRONIZATION HERE
! Perform filter over exclusive region only
  do j=1+halo,n-halo
    do i=1+halo,m-halo
      sum = 0.
      do l=-halo,halo
        do k=-halo,halo
          sum = sum + stencil(k,l)*V(i+k,j+l)
        enddo
      enddo
      dom(i,j) = sum
    enddo
  enddo

! Pair-wise handshake synchronization
  do mode=0,1
    if ( mod(myP,2) == mode ) then
      sync images( (myQ-1)*p+WW )    ! West

      do j=1+halo,n-halo
        do i=1,halo
! Apply filter
            dom(i,j) = sum
        enddo
      enddo
    else
      sync images( (myQ-1)*p+EE )    ! East
      do j=1+halo,n-halo
        do i=m-halo+1,m
         :
```

Stencil 2D Results on XT5, XE6, X2; Halo = 1

Stencil 2D Results on XT5, XE6, X2; Halo = 1

Using a fixed size virtual PE topology, vary
the size of the local square

Stencil 2D Results on XT5, XE6, X2; Halo = 1

Using a fixed size virtual PE topology, vary the size of the local square

- XT5: CAF puts/gets implemented through message-passing library

Stencil 2D Results on XT5, XE6, X2; Halo = 1

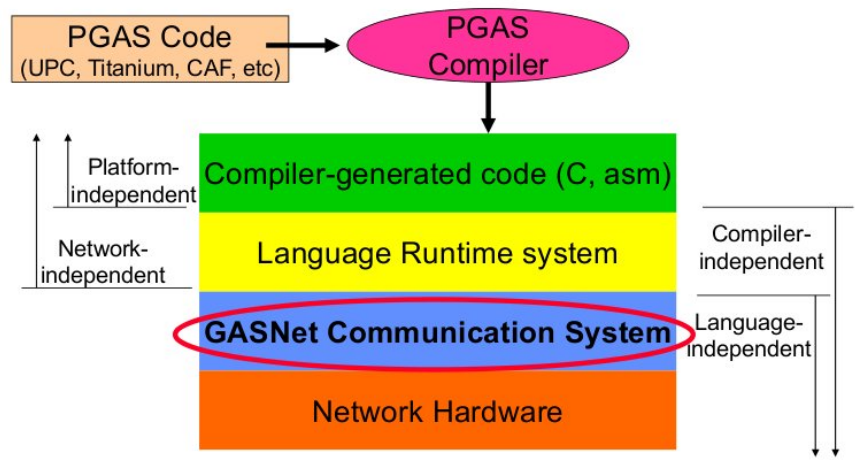
Using a fixed size virtual PE topology, vary the size of the local square

- XT5: CAF puts/gets implemented through message-passing library
- XE6, X2: RMA-enabled hardware support for PGAS, but still must pass through software stack

Stencil 2D Results on XT5, XE6, X2; Halo = 1

Using a fixed size virtual PE topology, vary the size of the local square

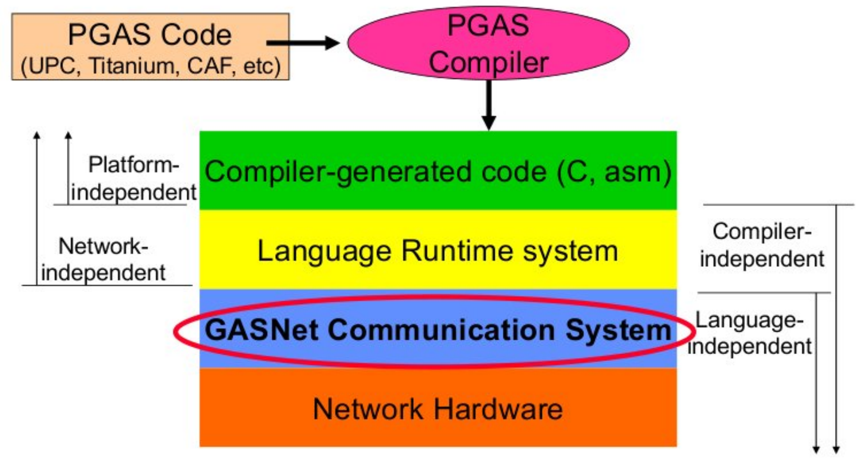
- XT5: CAF puts/gets implemented through message-passing library
- XE6, X2: RMA-enabled hardware support for PGAS, but still must pass through software stack

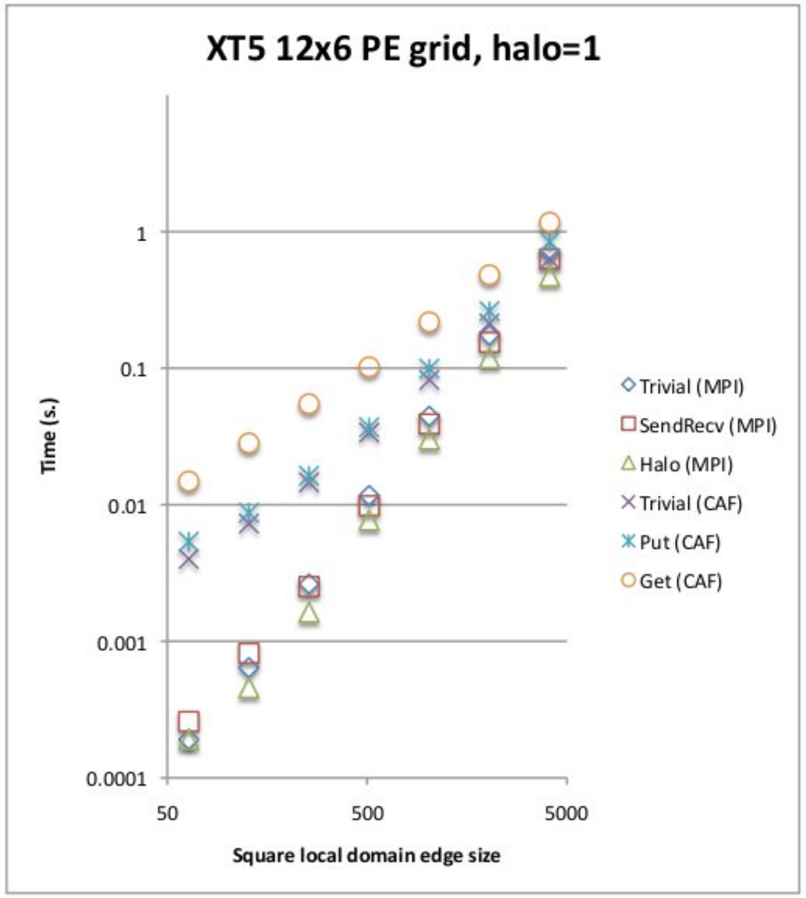


Stencil 2D Results on XT5, XE6, X2; Halo = 1

Using a fixed size virtual PE topology, vary the size of the local square

- XT5: CAF puts/gets implemented through message-passing library
- XE6, X2: RMA-enabled hardware support for PGAS, but still must pass through software stack

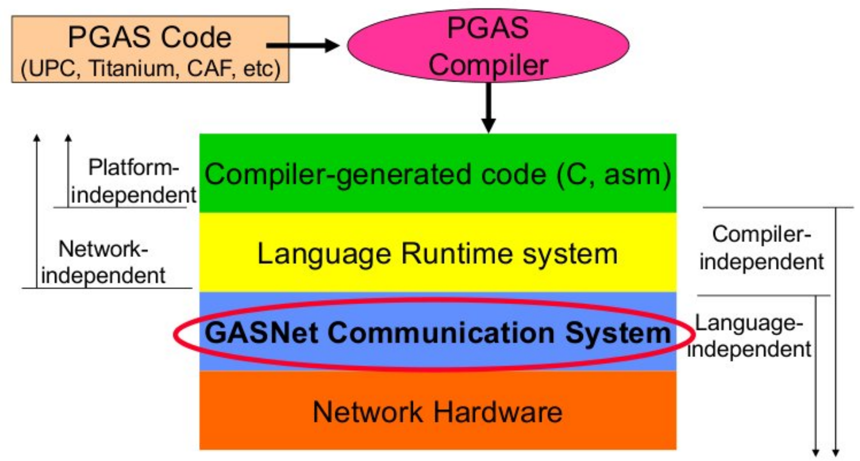


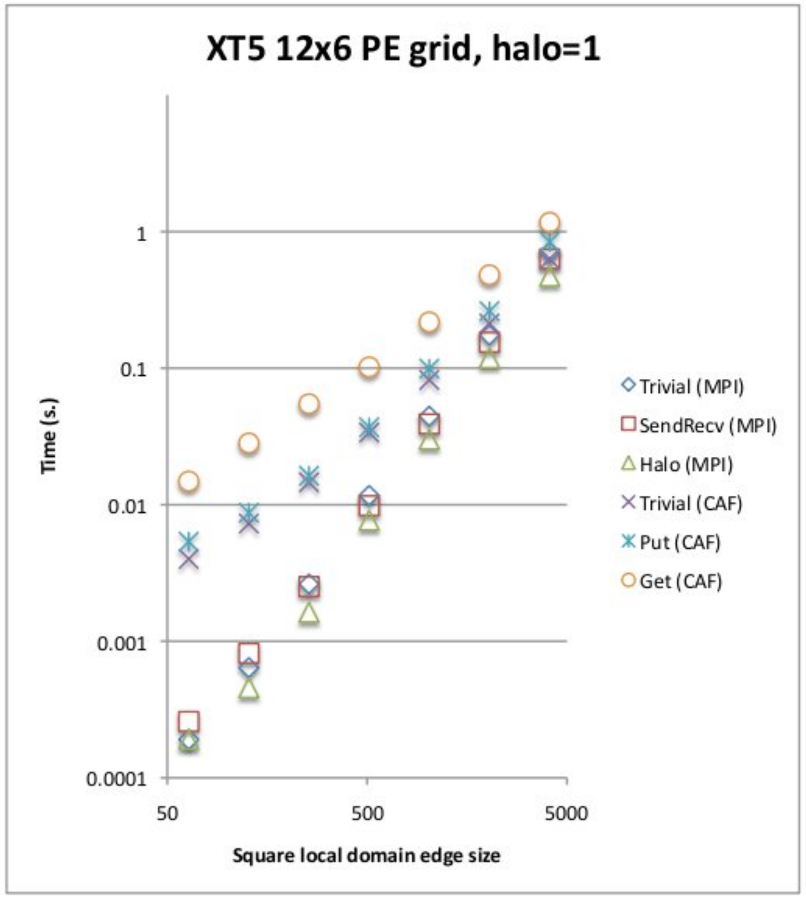


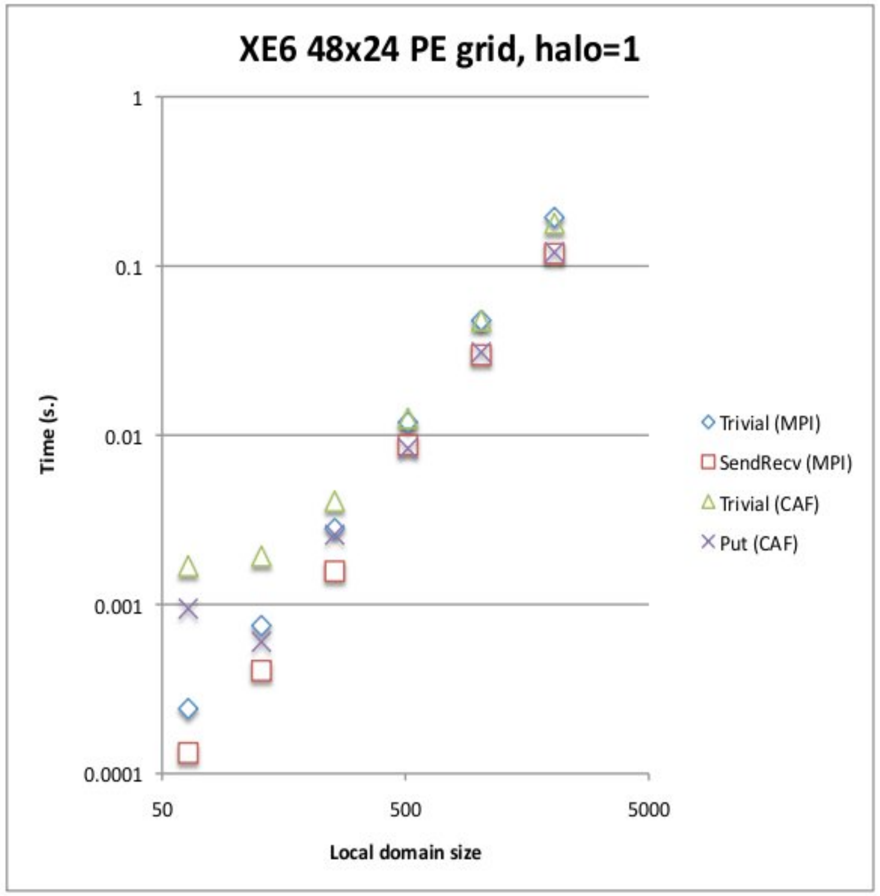
Stencil 2D Results on XT5, XE6, X2; Halo = 1

Using a fixed size virtual PE topology, vary the size of the local square

- XT5: CAF puts/gets implemented through message-passing library
- XE6, X2: RMA-enabled hardware support for PGAS, but still must pass through software stack



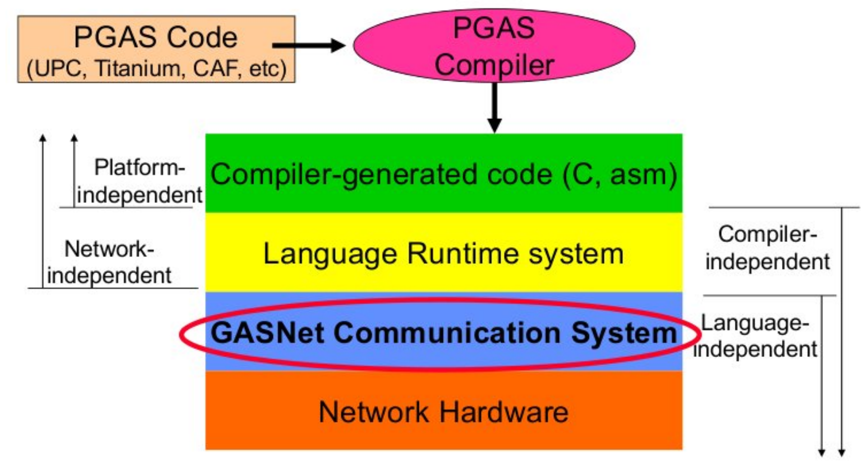


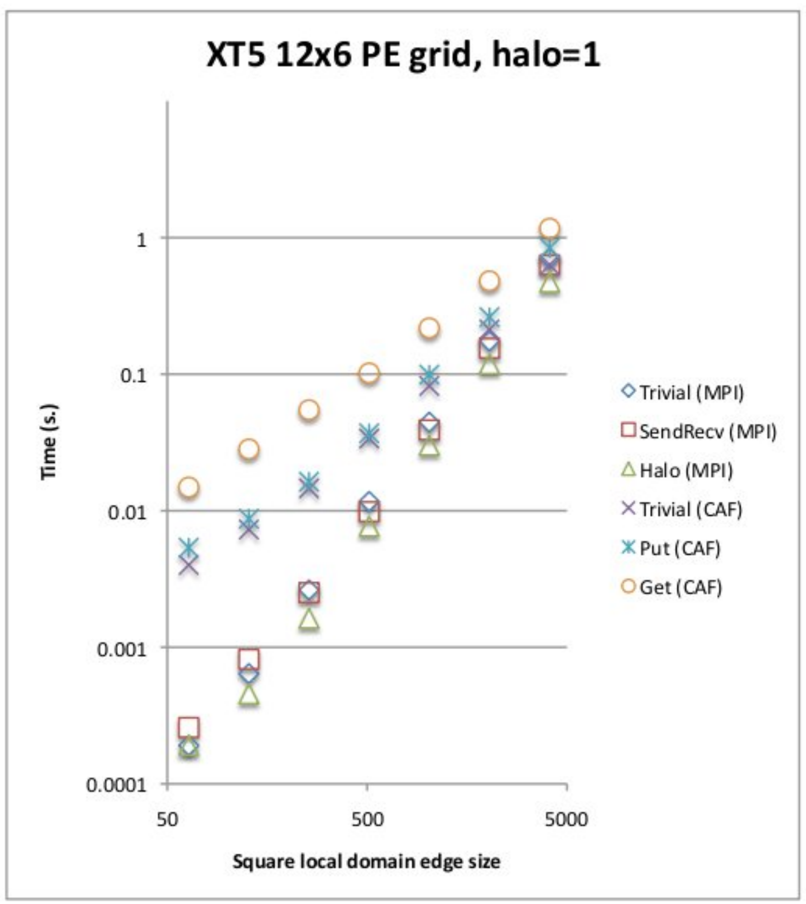


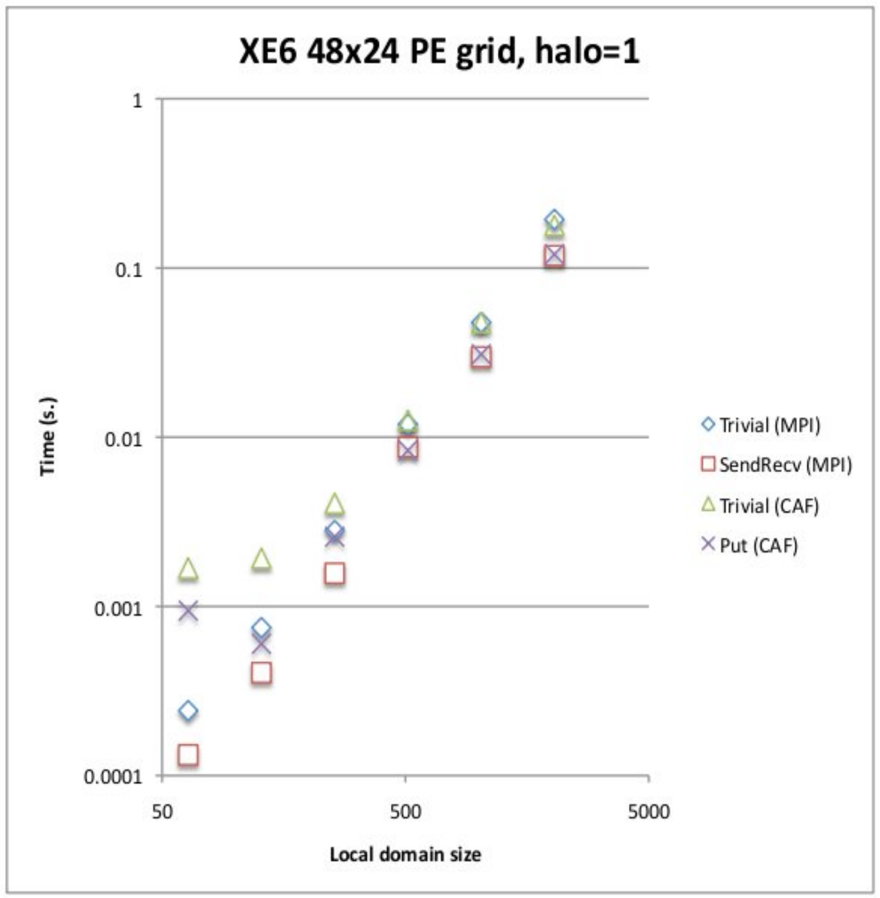
Stencil 2D Results on XT5, XE6, X2; Halo = 1

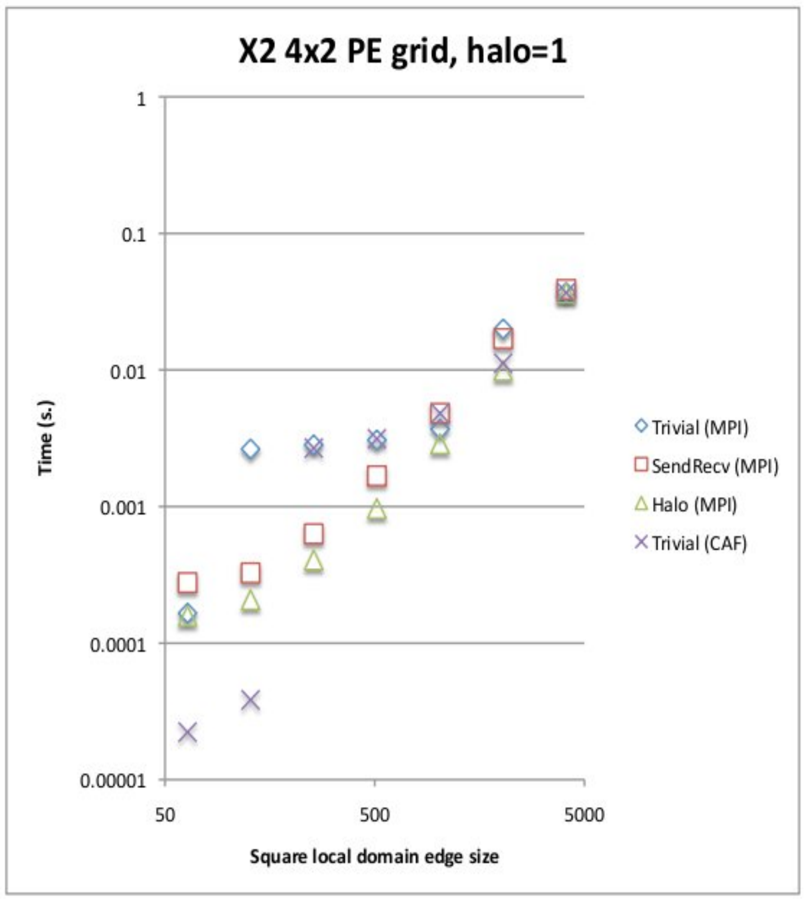
Using a fixed size virtual PE topology, vary the size of the local square

- XT5: CAF puts/gets implemented through message-passing library
- XE6, X2: RMA-enabled hardware support for PGAS, but still must pass through software stack







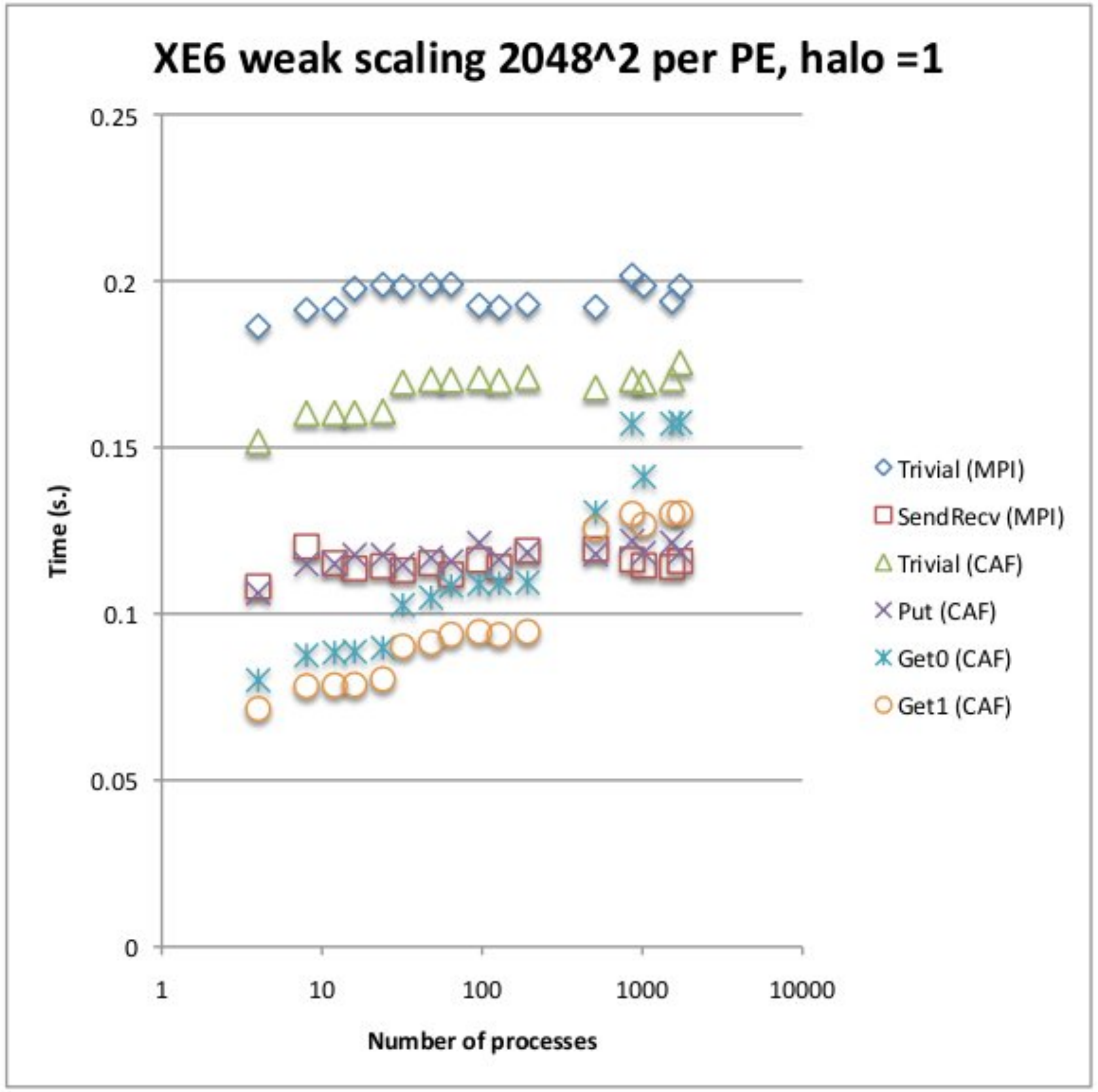


Stencil 2D Weak Scaling on XE6

Fixed local dimension, vary the PE virtual topology (take the optimal configuration)

Stencil 2D Weak Scaling on XE6

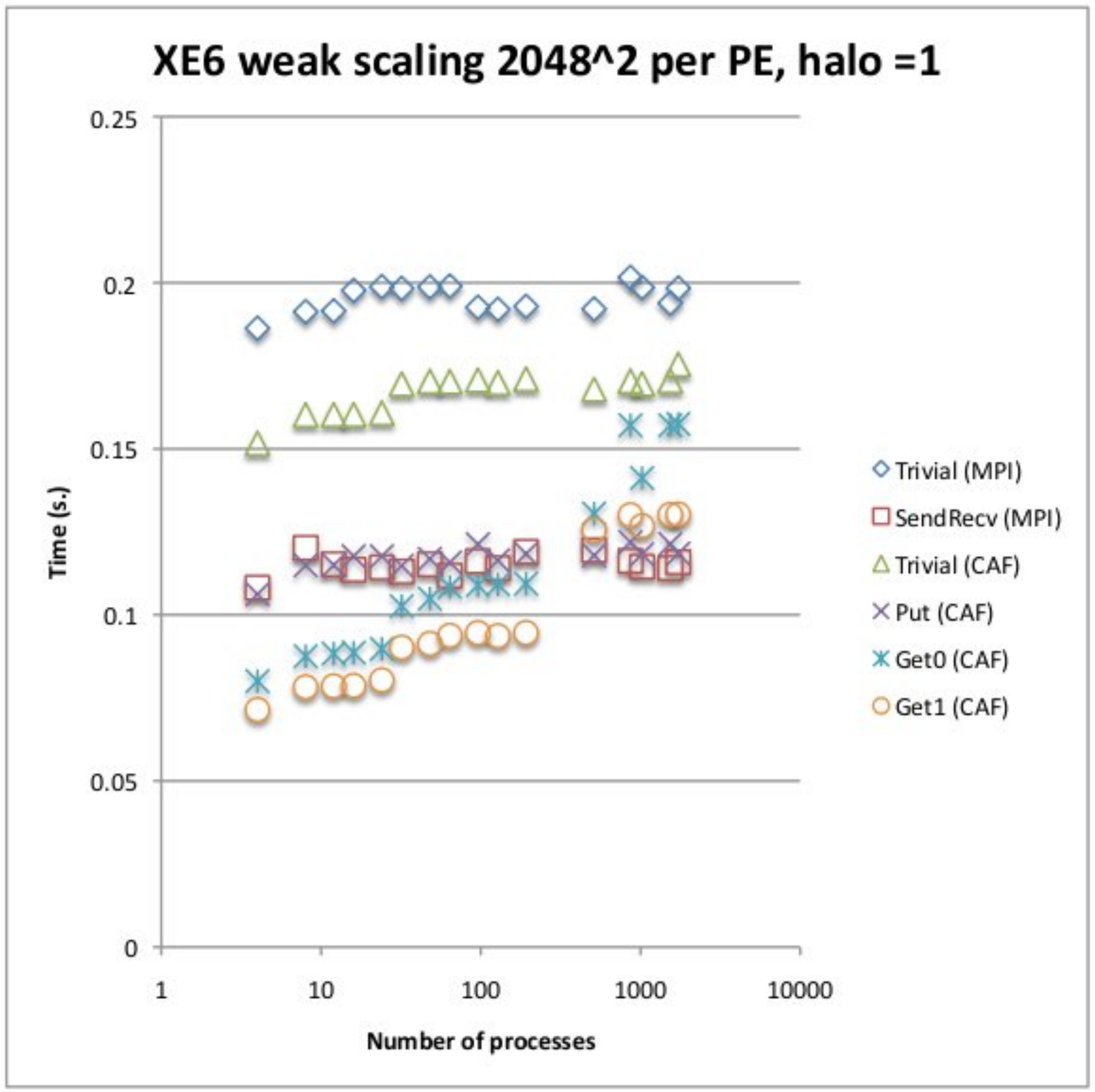
Fixed local dimension, vary the PE virtual topology (take the optimal configuration)

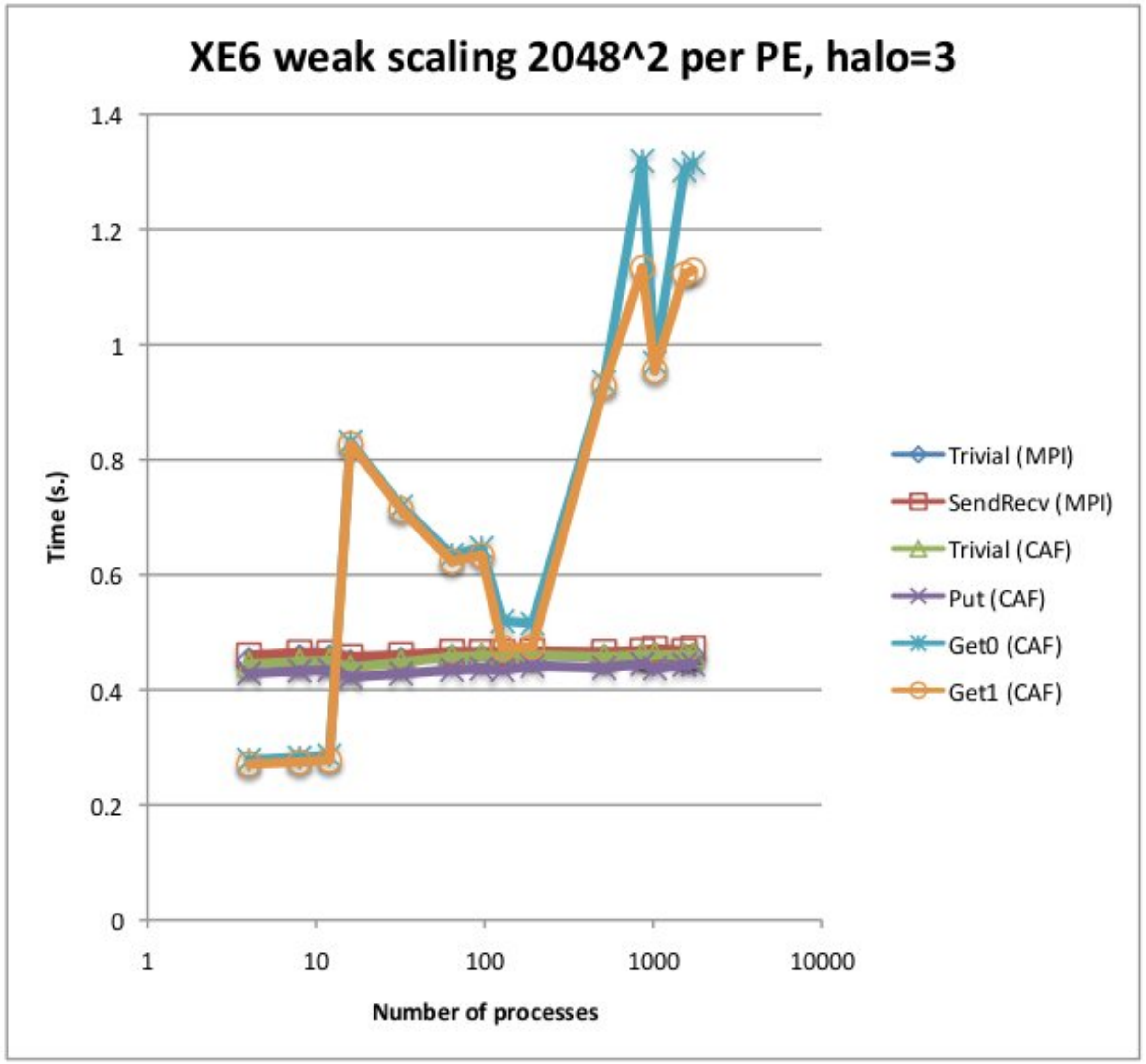


Friday, May 25, 2012

Stencil 2D Weak Scaling on XE6

Fixed local dimension, vary the PE virtual topology (take the optimal configuration)





Problem 2: Exact Diagonalization

$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

Problem 2: Exact Diagonalization

$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

Problem 2: Exact Diagonalization

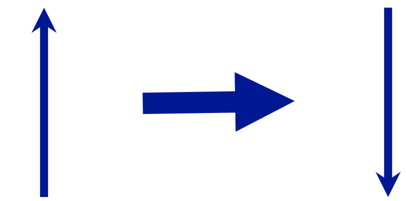
$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

Problem 2: Exact Diagonalization

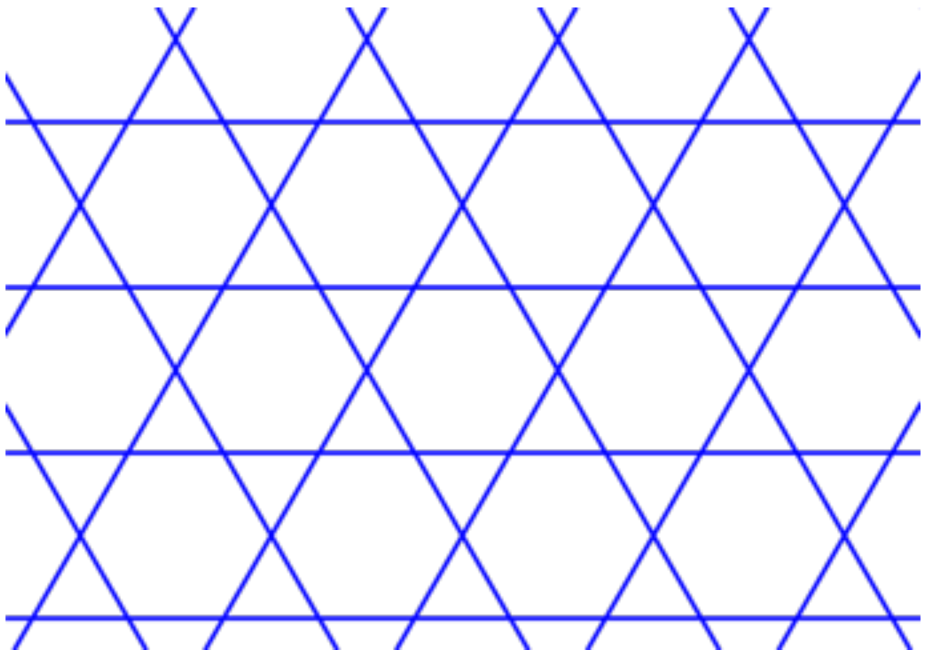
$\uparrow \;\rightarrow\; \downarrow$

$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

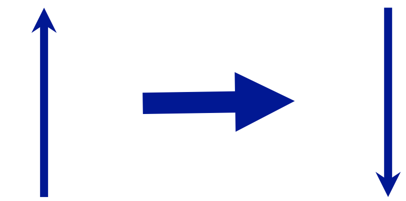
Problem 2: Exact Diagonalization



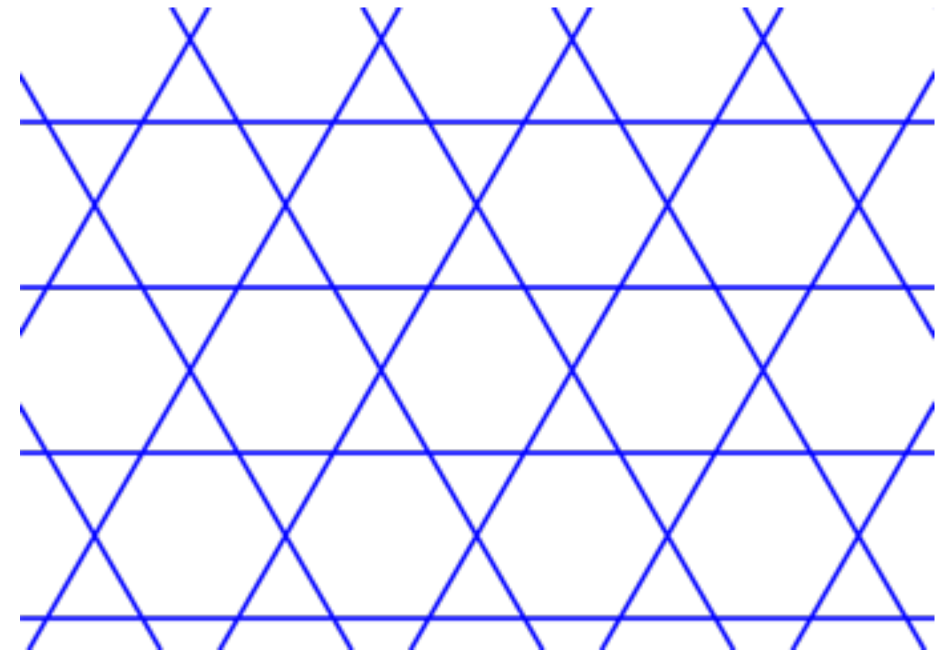
$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$



Problem 2: Exact Diagonalization



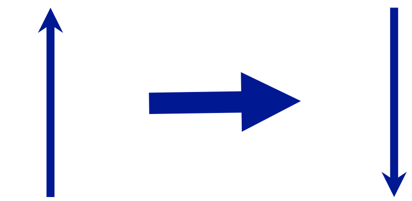
$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$



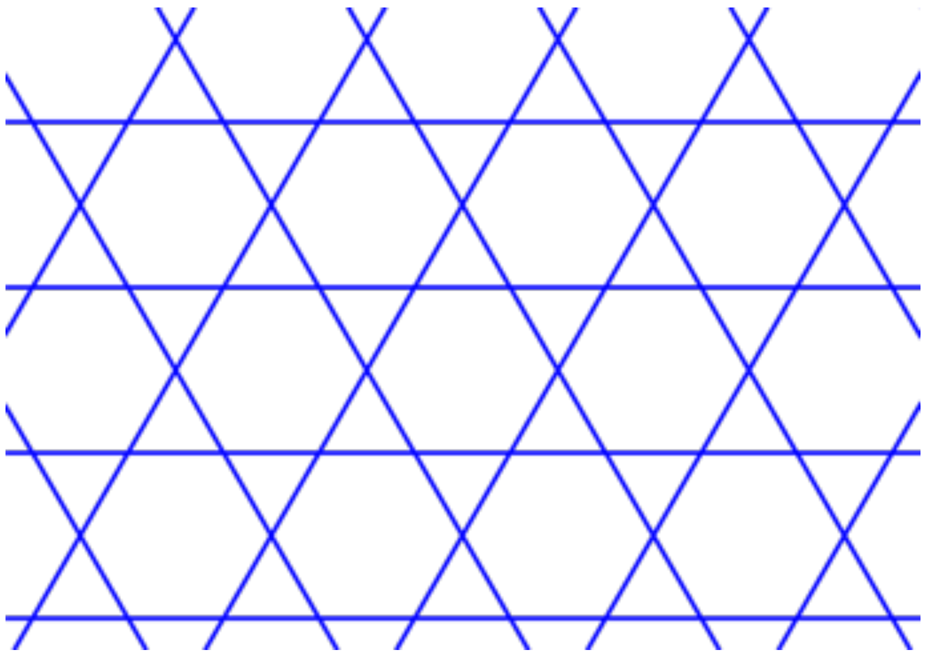
Algorithm 1 *Lanczos iteration*

1: $v_1 \leftarrow$ random vector with norm 1
2: $v_0 \leftarrow 0$
3: $\beta_1 \leftarrow 0$
4: **for** $j = 1, \ldots, r$ **do**
5: $\quad w_j \leftarrow Hv_j - \beta_j v_{j-1}$
6: $\quad \alpha_j \leftarrow (w_j, v_j)$
7: $\quad \beta_{j+1} \leftarrow \|w_j\|$
8: $\quad v_{j+1} \leftarrow w_j / \beta_{j+1}$
9: **end for**

Problem 2: Exact Diagonalization



$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$



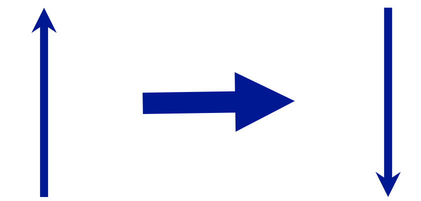
Algorithm 1 *Lanczos iteration*

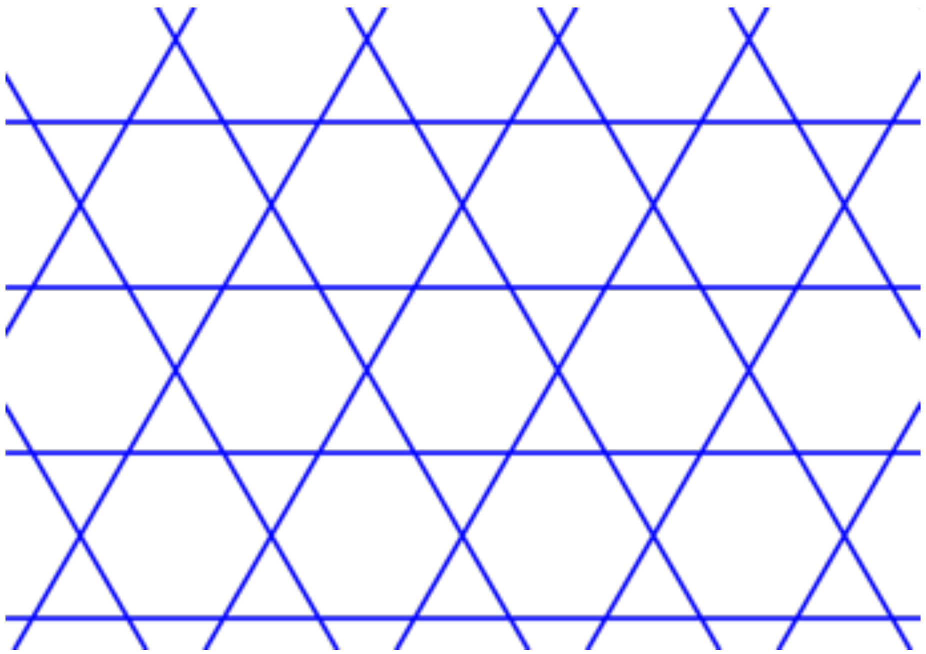
```
v_1 ← random vector with norm 1
v_0 ← 0
β_1 ← 0
for j = 1, ..., r do
    w_j ← H v_j − β_j v_{j−1}
    α_j ← (w_j, v_j)
    β_{j+1} ← ||w_j||
    v_{j+1} ← w_j / β_{j+1}
end for
```

$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Problem 2: Exact Diagonalization

$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$



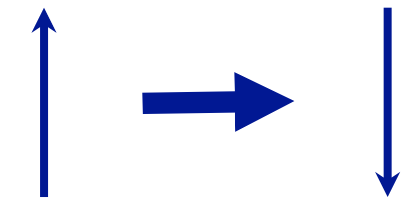


Algorithm 1 *Lanczos iteration*

1: $v_1 \leftarrow$ random vector with norm 1
2: $v_0 \leftarrow 0$
3: $\beta_1 \leftarrow 0$
4: **for** $j = 1, \ldots, r$ **do**
5: $\quad w_j \leftarrow H v_j - \beta_j v_{j-1}$
6: $\quad \alpha_j \leftarrow (w_j, v_j)$
7: $\quad \beta_{j+1} \leftarrow \|w_j\|$
8: $\quad v_{j+1} \leftarrow w_j / \beta_{j+1}$
9: **end for**

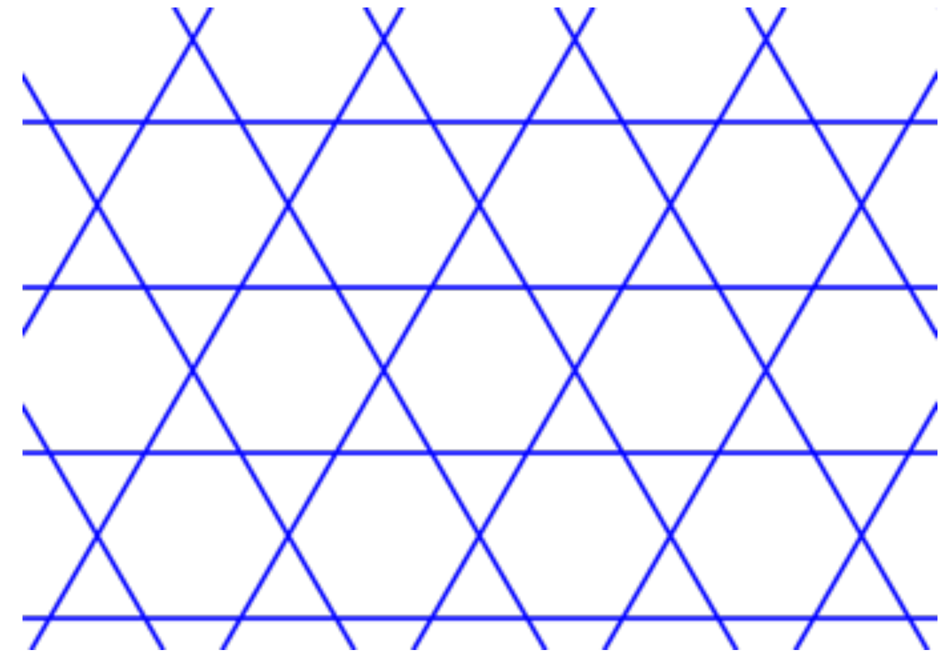
$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Problem 2: Exact Diagonalization



$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

- Any lattice with n sites, 2^n states

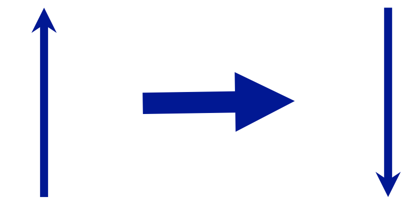


Algorithm 1 *Lanczos iteration*

$v_1 \leftarrow$ random vector with norm 1
$v_0 \leftarrow 0$
$\beta_1 \leftarrow 0$
for $j = 1, \ldots, r$ **do**
  $w_j \leftarrow Hv_j - \beta_j v_{j-1}$
  $\alpha_j \leftarrow (w_j, v_j)$
  $\beta_{j+1} \leftarrow \|w_j\|$
  $v_{j+1} \leftarrow w_j / \beta_{j+1}$
end for

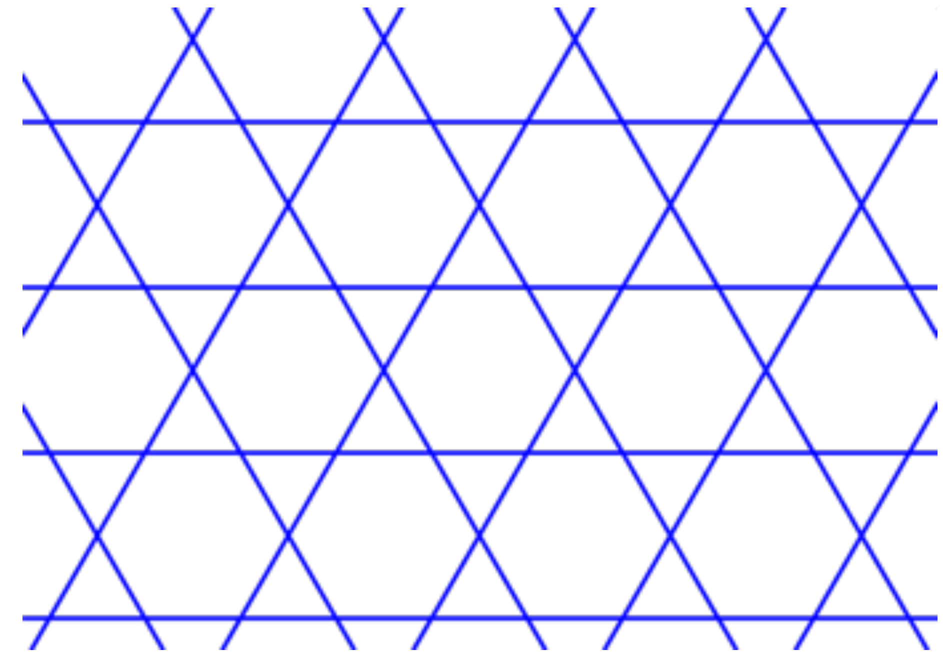
$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Problem 2: Exact Diagonalization



$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

- Any lattice with n sites, 2^n states
- Lanczos eigensolver



Algorithm 1 *Lanczos iteration*

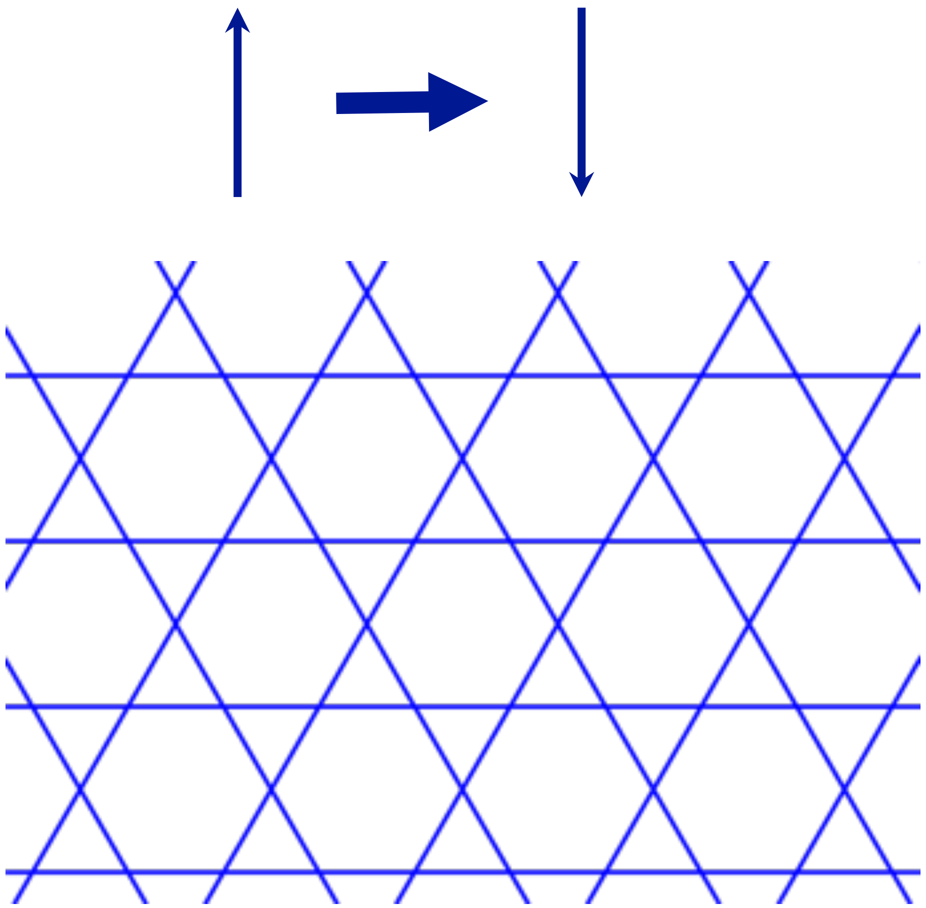
```
v_1 ← random vector with norm 1
v_0 ← 0
β_1 ← 0
for j = 1, ..., r do
    w_j ← H v_j − β_j v_{j−1}
    α_j ← (w_j, v_j)
    β_{j+1} ← ||w_j||
    v_{j+1} ← w_j / β_{j+1}
end for
```

$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Problem 2: Exact Diagonalization

$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

- Any lattice with n sites, 2^n states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec

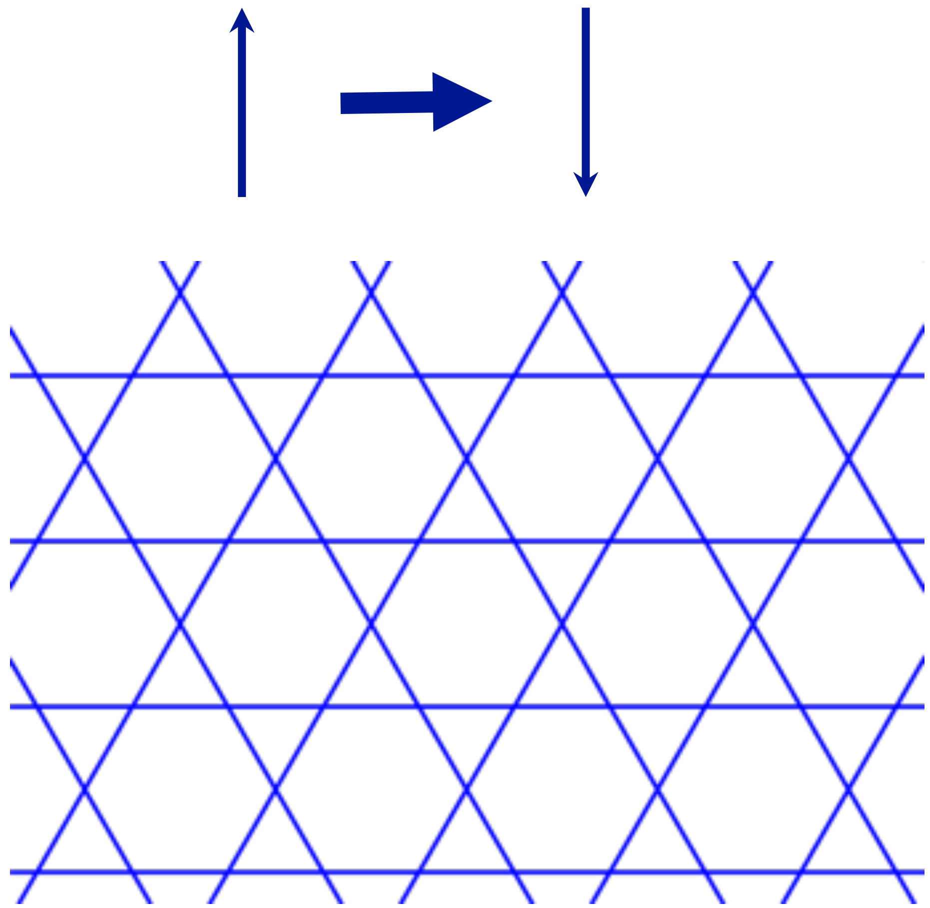


Algorithm 1 *Lanczos iteration*

1: $v_1 \leftarrow$ random vector with norm 1
2: $v_0 \leftarrow 0$
3: $\beta_1 \leftarrow 0$
4: **for** $j = 1, \ldots, r$ **do**
5: $\quad w_j \leftarrow H v_j - \beta_j v_{j-1}$
6: $\quad \alpha_j \leftarrow (w_j, v_j)$
7: $\quad \beta_{j+1} \leftarrow \|w_j\|$
8: $\quad v_{j+1} \leftarrow w_j / \beta_{j+1}$
9: **end for**

$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Problem 2: Exact Diagonalization



$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

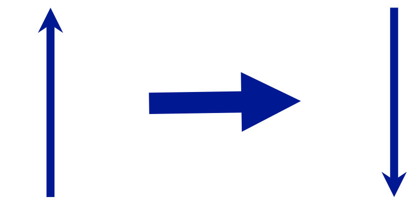
- Any lattice with n sites, 2^n states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec
- Operator has integer operations

Algorithm 1 *Lanczos iteration*

1: $v_1 \leftarrow$ random vector with norm 1
2: $v_0 \leftarrow 0$
3: $\beta_1 \leftarrow 0$
4: **for** $j = 1, \ldots, r$ **do**
5: $\quad w_j \leftarrow H v_j - \beta_j v_{j-1}$
6: $\quad \alpha_j \leftarrow (w_j, v_j)$
7: $\quad \beta_{j+1} \leftarrow \|w_j\|$
8: $\quad v_{j+1} \leftarrow w_j / \beta_{j+1}$
9: **end for**

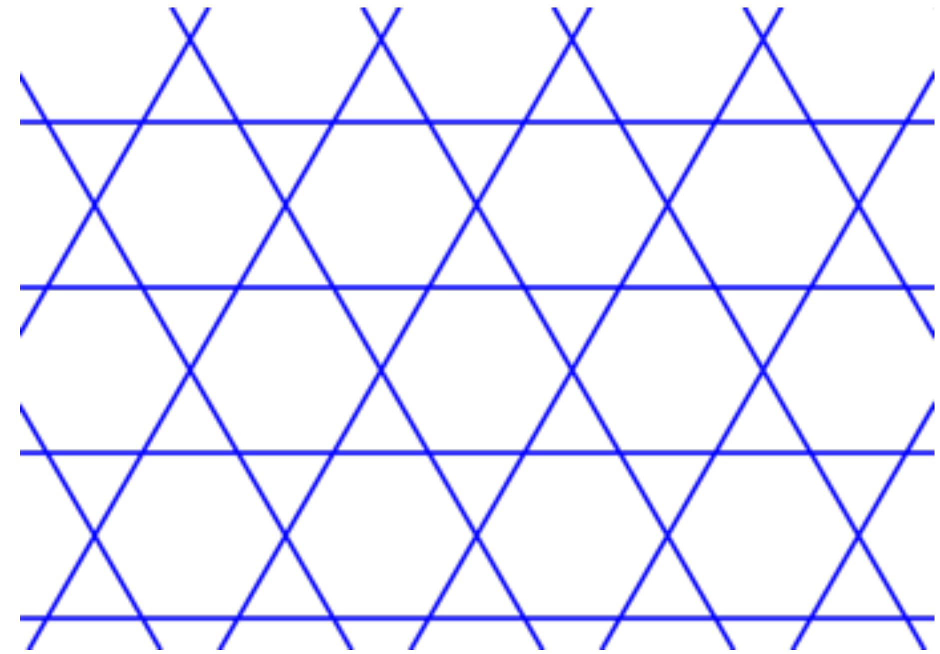
$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Problem 2: Exact Diagonalization



$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

- Any lattice with n sites, 2^n states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec
- Operator has integer operations
- Very irregular sparsity, but

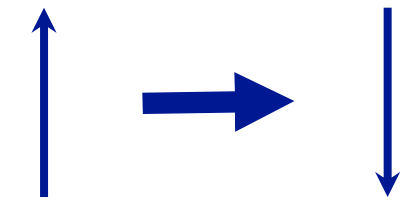


Algorithm 1 *Lanczos iteration*

$v_1 \leftarrow$ random vector with norm 1
$v_0 \leftarrow 0$
$\beta_1 \leftarrow 0$
for $j = 1, \ldots, r$ **do**
    $w_j \leftarrow Hv_j - \beta_j v_{j-1}$
    $\alpha_j \leftarrow (w_j, v_j)$
    $\beta_{j+1} \leftarrow \|w_j\|$
    $v_{j+1} \leftarrow w_j / \beta_{j+1}$
end for

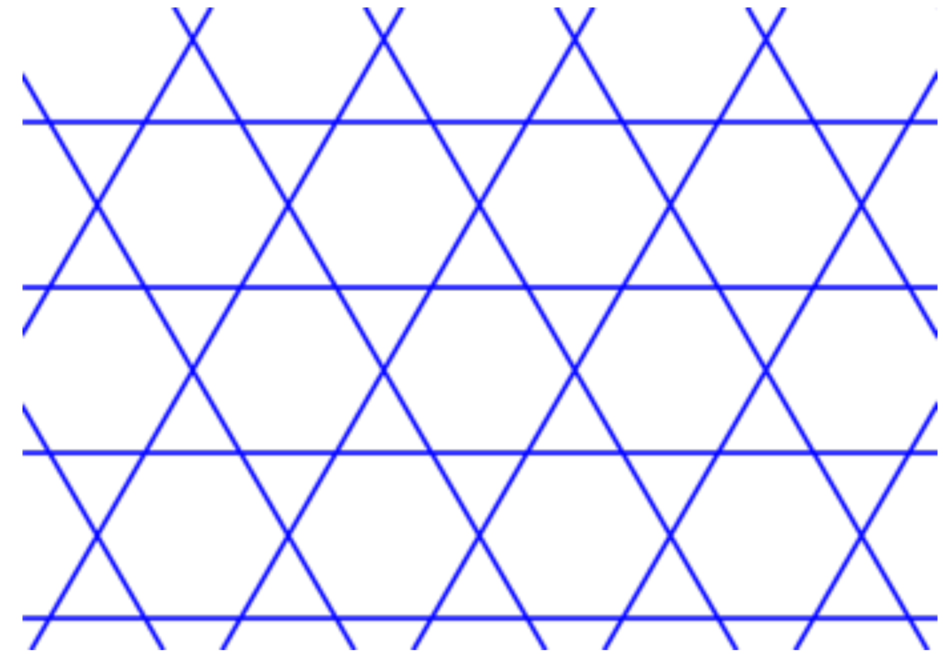
$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Problem 2: Exact Diagonalization



$$H = J \sum S_i^z S_j^z + \Gamma \sum S_i^x$$

- Any lattice with n sites, 2^n states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec
- Operator has integer operations
- Very irregular sparsity, but
- Limited number of process neighbors *(new to this work)*

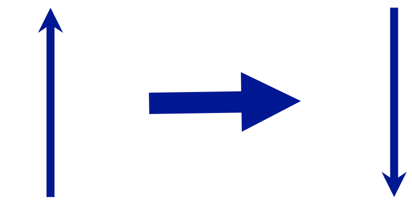


Algorithm 1 *Lanczos iteration*

```
v_1 ← random vector with norm 1
v_0 ← 0
β_1 ← 0
for j = 1, ..., r do
    w_j ← H v_j − β_j v_{j−1}
    α_j ← (w_j, v_j)
    β_{j+1} ← ‖w_j‖
    v_{j+1} ← w_j / β_{j+1}
end for
```

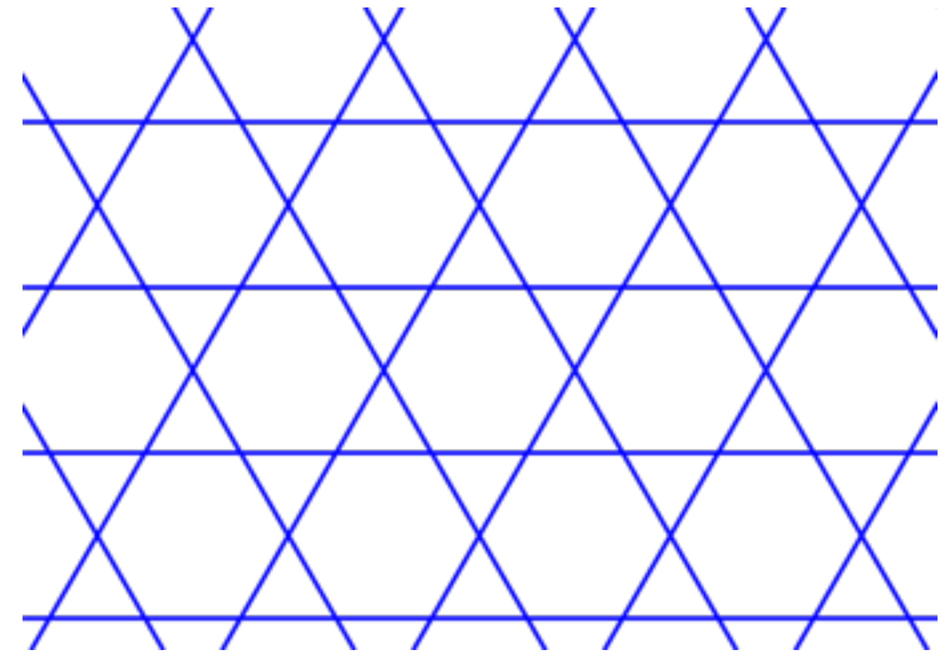
$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Problem 2: Exact Diagonalization



$$H = J\sum S_i^z S_j^z + \Gamma \sum S_i^x$$

- Any lattice with n sites, 2^n states
- Lanczos eigensolver
- Large, sparse symmetric mat-vec
- Operator has integer operations
- Very irregular sparsity, but
- Limited number of process neighbors *(new to this work)*
- Symmetries considered in some models: smaller complexity at cost of more communication



Algorithm 1 *Lanczos iteration*

$v_1 \leftarrow$ random vector with norm 1
$v_0 \leftarrow 0$
$\beta_1 \leftarrow 0$
for $j = 1, \dots, r$ **do**
  $w_j \leftarrow Hv_j - \beta_j v_{j-1}$
  $\alpha_j \leftarrow (w_j, v_j)$
  $\beta_{j+1} \leftarrow \|w_j\|$
  $v_{j+1} \leftarrow w_j / \beta_{j+1}$
end for

$$T_H = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ \beta_2 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_r \\ & & \beta_r & \alpha_r \end{bmatrix}$$

Benchmark code: simplest "SPIN" model

Benchmark code: simplest "SPIN" model

Loop for MAX_ITER
- **Reduction B** (MPI_Reduce)
- **L3** (local work, normalize v1)
- **Loop over rounds of msgs**
  - **MSG_NB** (MPI_Isend,upc_memput(_nbi))
  - **L4** (work on local matrix, only 1st iteration)
  - **SYNC** (no-op, upc_fence)
  - **L7** (manage msg reception and do remote work)
- **L8** (local work, A norm)
- **Reduction A** (MPI_Reduce)
- **L9** (local work, B norm)

Benchmark code: simplest “SPIN” model

Loop for MAX_ITER
- **Reduction B** (MPI_Reduce)
- **L3** (local work, normalize v1)
- **Loop over rounds of msgs**
  - **MSG_NB** (MPI_Isend,upc_memput(_nbi))
  - **L4** (work on local matrix, only 1st iteration)
  - **SYNC** (no-op, upc_fence)
  - **L7** (manage msg reception and do remote work)
- **L8** (local work, A norm)
- **Reduction A** (MPI_Reduce)
- **L9** (local work, B norm)

Loops:
- **L3: Initialize array**
- **L4: Local mat-vec**
- **L6/7: Off process mat-vec**
- **L8: Alpha calculation**
- **L9: Beta calculation**

Benchmark code: simplest “SPIN” model

Loop for MAX_ITER
- **Reduction B** (MPI_Reduce)
- **L3** (local work, normalize v1)
- **Loop over rounds of msgs**
  - **MSG_NB** (MPI_Isend,upc_memput(_nbi))
  - **L4** (work on local matrix, only 1st iteration)
  - **SYNC** (no-op, upc_fence)
  - **L7** (manage msg reception and do remote work)
- **L8** (local work, A norm)
- **Reduction A** (MPI_Reduce)
- **L9** (local work, B norm)

Loops:
- L3: Initialize array
- L4: Local mat-vec
- L6/7: Off process mat-vec
- L8: Alpha calculation
- L9: Beta calculation

Code structure

```
void execute(ed){
  L1: for (i=0; i<2^(n-m); i++)
        b += v1[i]*v1[i]
  L2: for(iter=0; iter<max_iters; iter++)
        MPI_Barrier
        MPI_Allreduce of b
    L3: for (i=0; i<2^(n-m); i++)
          v1[i] *= f1(iter,b)
    MPI_Isend/MPI_Irecv to vv1 using handler1
    // local contribution, no comm
    L4: for (i=0; i<2^(n-m); i++)
          v2[i] = a(i)*v1[i]
          L5: for (k=0; k<n-m; k++)
                v2[i] += g*v1[f2(i,k)]
    // remote contribution, comm
    L6: for (k=n-m; k<n; k++) // m iterations
          MPI_Isend/MPI_Irecv to vv2 using handler2
          MPI_Wait on handler1
          L7: for (i=0; i<2^(n-m); i++)
                v2[i] += g*vv1[i]
          swap(vv1,vv2) // Pointer swap, no memcpy
          handler1 = handler2
    L8: for(i=0; i<2^(n-m); i++)
           a += v1[i]*v2[i]
        MPI_Barrier
        MPI_Allreduce of a
    L9: for(i=0; i<2^(n-m); i++)
          v2[i] = f3(i,iter,a)
          b += v2[i]*v2[i]
        swap (v0,v1,v2)
}
```

L3 L4 L6 L8 L9 execute

SPIN single core/socket/node comparisons

- Loop-based OMP directives: performance worse than MPI-only
- Task-based OpenMP/MPI implementation by Fourestey/Stringfellow

SPIN single core/socket/node comparisons

- Loop-based OMP directives: performance worse than MPI-only
- Task-based OpenMP/MPI implementation by Fourestey/Stringfellow

System name	Rivera	Castor	Sandy
Processor	AMD 6274	Intel E5-2680	Intel X5650
Nickname	Interlagos	Westmere	Sandybridge
Cores/Socket	16	6	8
Sockets/Node	2	2	2
Hyperthreading	no	unenabled	yes (2)
Compiler	Open64	Intel	Intel
Core time (s.)	754 (1T)	280 (1T)	227 (1T)
Socket time (s.)	74 (15T)	51 (6T)	29 (16T)
Node time (s.)	38 (31T)	26 (12T)	15 (32T)

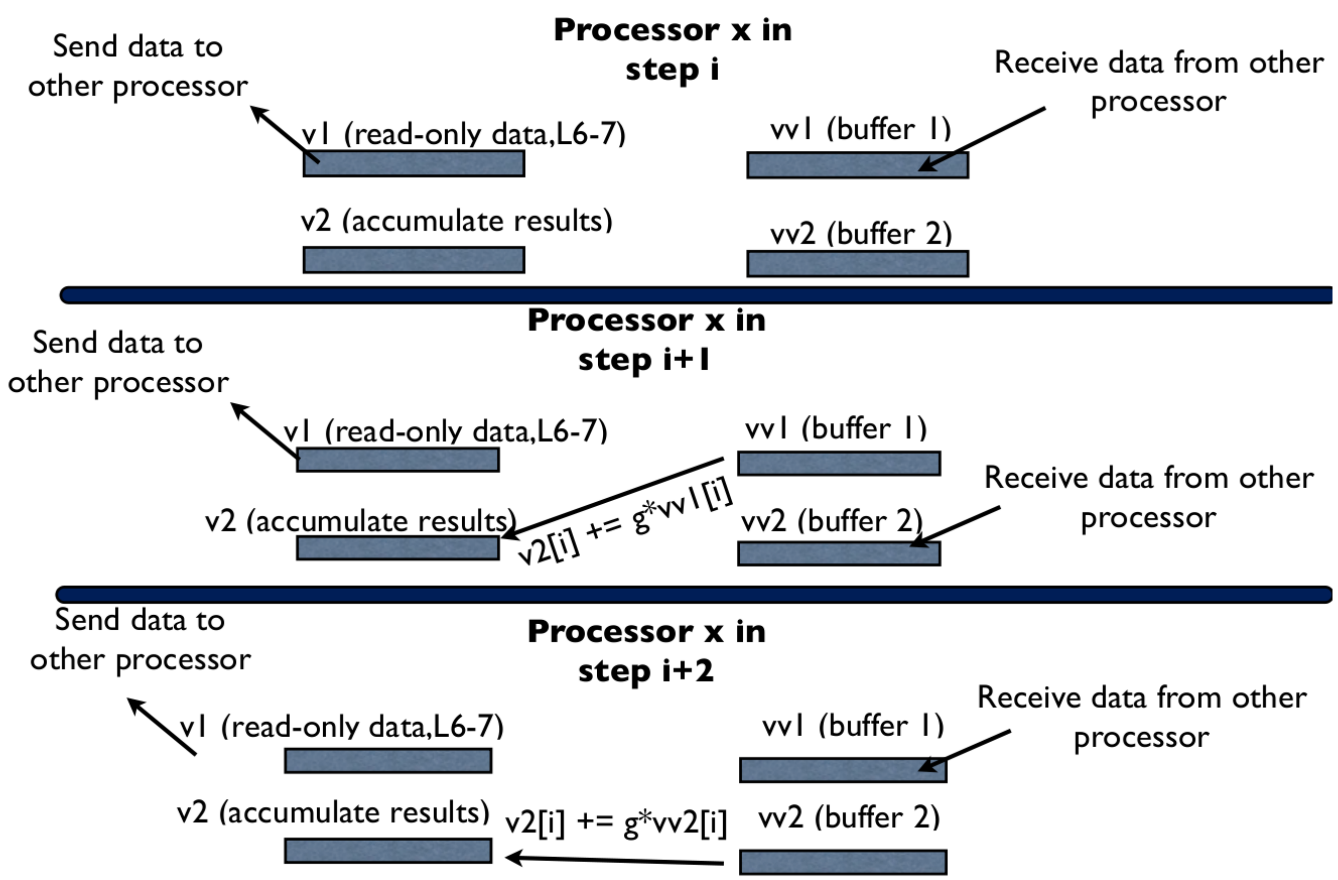
Multi-buffering concept

Multi-buffering concept

Double buffering

Multi-buffering concept

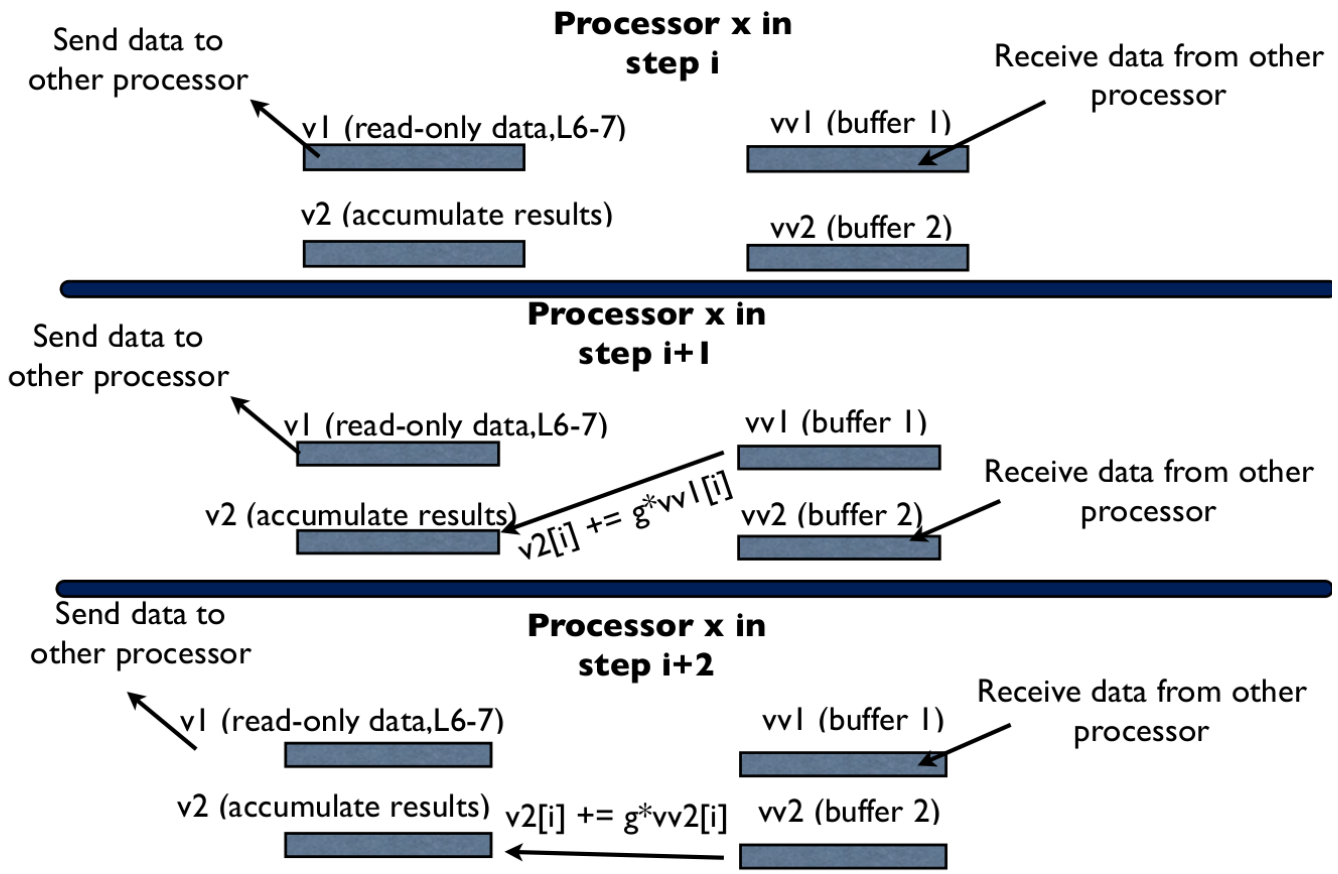
Double buffering



Multi-buffering concept

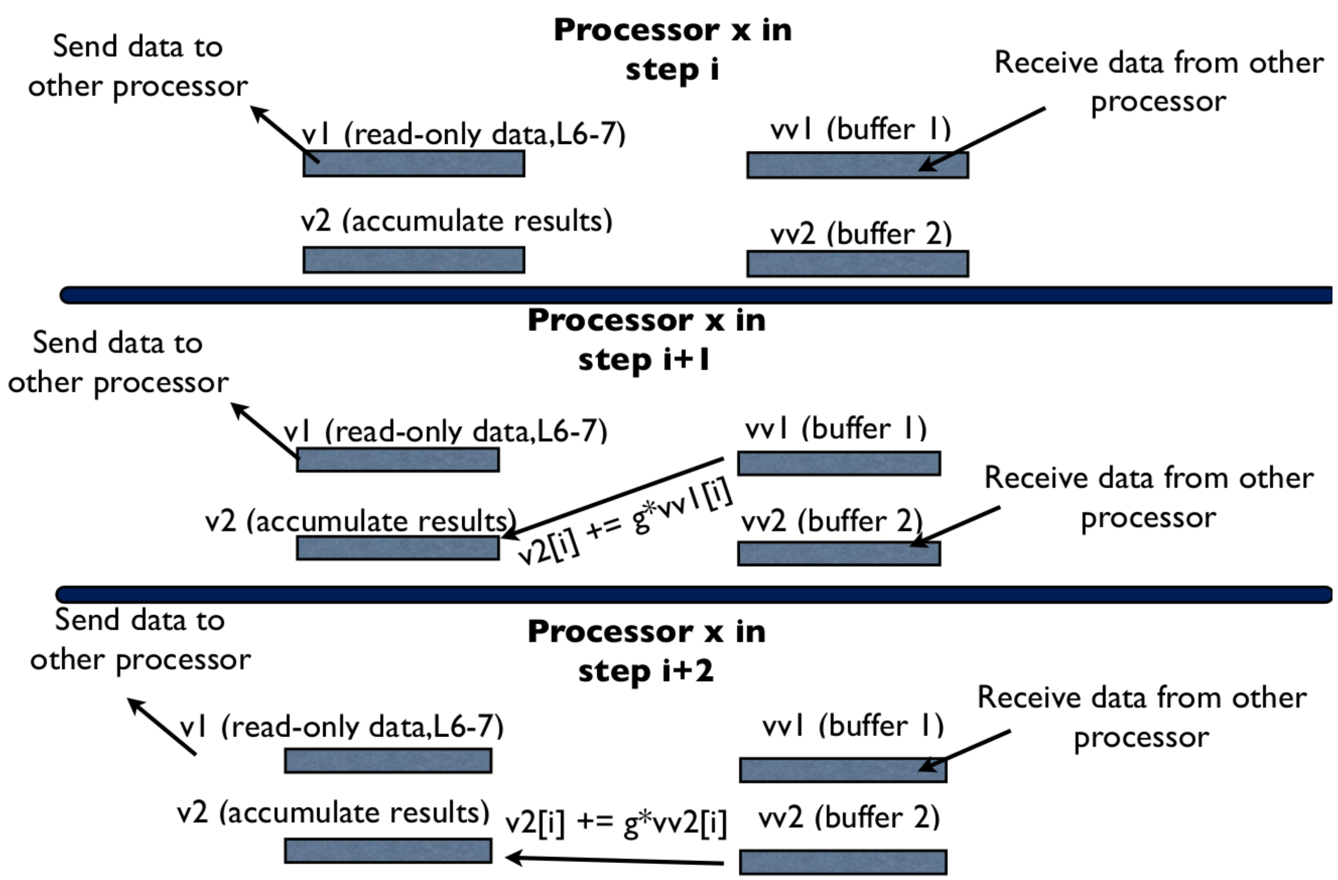
Double buffering

Multi-buffering



Multi-buffering concept

Double buffering



Multi-buffering

REFERENCE — 1 private buffer per pe

- Local work
- Loop
  - 2-sided MPI_Isend/Irecv (single round)
  - MPI_Wait
  - Remote work (in order)

OPTIMIZED — k shared buffers per pe (limited by on-node mem)

- Loop
  - 1-sided non-blocking put (round of k msgs)
  - Local work (if any)
  - Sync (e.g. barrier, fence, or notification flags)
  - Remote work (out of order)

Sequential Implementation

Sequential Implementation

```
struct ed_s { ...
        double *v0, *v1, *v2;       /* vectors */
        double *swap;               /* for swapping vectors */
};
              :
for (iter = 0; iter < ed->max_iter; ++iter) {
                 :
                /* matrix vector multiplication */
                for (s = 0; s < ed->nlstates; ++s ) {
                        /* diagonal part */
                        ed->v2[s] = diag(s, ed->n, ed->j) * ed->v1[s];
                        /* offdiagonal part */
                        for (k = 0; k < ed->n; ++k) {
                                s1 = flip_state(s, k);
                                ed->v2[s] += ed->gamma * ed->v1[s1];
                        }
                }
                 :
                /* Calculate alpha */
                /* Calculate beta */
        }
}
```

UPC “Elegant” Implementation

UPC “Elegant” Implementation

```
struct ed_s { ...
        shared double *v0, *v1, *v2;        /* vectors */
        shared double *swap;                /* for swapping vectors */
};
              :
for (iter = 0; iter < ed->max_iter; ++iter) {
                :
                upc_barrier(0);
                /* matrix vector multiplication */
                upc_forall (s = 0; s < ed->nlstates; ++s; &(ed->v1[s]) ) {
                        /* diagonal part */
                        ed->v2[s] = diag(s, ed->n, ed->j) * ed->v1[s];
                        /* offdiagonal part */
                        for (k = 0; k < ed->n; ++k) {
                                s1 = flip_state(s, k);
                                ed->v2[s] += ed->gamma * ed->v1[s1];
                        }
                }
                :
                /* Calculate alpha */
                /* Calculate beta */
        }
}
```

Inelegant UPC versions

Inelegant UPC versions

Inelegant 1

```
shared[NBLOCK] double vtmp[THREADS*NBLOCK];
  :
for (i = 0; i < NBLOCK; ++i) vtmp[i+MYTHREAD*NBLOCK] = ed->v1[i];
upc_barrier(1);
for (i = 0; i < NBLOCK; ++i) ed->vv1[i] = vtmp[i+(ed->from_nbs[0]*NBLOCK)];
  :
upc_barrier(2);
```

Inelegant 2

```
shared[NBLOCK] double vtmp[THREADS*NBLOCK];
  :
upc_memput( &vtmp[MYTHREAD*NBLOCK], ed->v1, NBLOCK*sizeof(double) );
upc_barrier(1);
upc_memget( ed->vv1, &vtmp[ed->from_nbs[0]*NBLOCK], NBLOCK*sizeof(double) );
  :
upc_barrier(2);
```

UPC *Inelegant3:* use double buffers and upc_put

UPC *Inelegant3:* use double buffers and upc_put

```
shared[NBLOCK] double vtmp1[THREADS*NBLOCK];
shared[NBLOCK] double vtmp2[THREADS*NBLOCK];
:
upc_memput( &vtmp1[ed->to_nbs[0]*NBLOCK], ed->v1, NBLOCK*sizeof(double) );
upc_barrier(1);
:
 if ( mode == 0 ) {
   upc_memput( &vtmp2[ed->to_nbs[neighb]*NBLOCK], ed->v1, NBLOCK*sizeof(double) );
 } else {
   upc_memput( &vtmp1[ed->to_nbs[neighb]*NBLOCK], ed->v1, NBLOCK*sizeof(double) );
 }
  :
 if ( mode == 0 ) {
   for (i = 0; i < ed->nlstates; ++i) { ed->v2[i] += ed->gamma * vtmp1[i+MYTHREAD*NBLOCK]; }
   mode = 1;
 } else {
   for (i = 0; i < ed->nlstates; ++i) { ed->v2[i] += ed->gamma * vtmp2[i+MYTHREAD*NBLOCK]; }
   mode = 0;
 }
 upc_barrier(2);
```

Other message passing paradigms

Other message passing paradigms

MPI-2: One-sided PUT

Other message passing paradigms

MPI-2: One-sided PUT

```
MPI_Put(ed->v1, ed->nlstates, MPI_DOUBLE, ed->to_nbs[0], 0, ed->nlstates, MPI_DOUBLE, win1);
MPI_Win_fence( 0, win1);
```

Other message passing paradigms

MPI-2: One-sided PUT

```
MPI_Put(ed->v1, ed->nlstates, MPI_DOUBLE, ed->to_nbs[0], 0, ed->nlstates, MPI_DOUBLE, win1);
MPI_Win_fence( 0, win1);
```

SHMEM: non-blocking PUT

Other message passing paradigms

MPI-2: One-sided PUT

```
MPI_Put(ed->v1, ed->nlstates, MPI_DOUBLE, ed->to_nbs[0], 0, ed->nlstates, MPI_DOUBLE, win1);
MPI_Win_fence( 0, win1);
```

SHMEM: non-blocking PUT

```
vtmp1 = (double *) shmalloc(ed->nlstates*sizeof(double));
  :
shmem_barrier_all();
shmem_double_put_nb(vtmp1, ed->v1, ed->nlstates, ed->from_nbs[neighb], NULL);
```

Other message passing paradigms

MPI-2: One-sided PUT

```
MPI_Put(ed->v1, ed->nlstates, MPI_DOUBLE, ed->to_nbs[0], 0, ed->nlstates, MPI_DOUBLE, win1);
MPI_Win_fence( 0, win1);
```

SHMEM: non-blocking PUT

```
vtmp1 = (double *) shmalloc(ed->nlstates*sizeof(double));
  :
shmem_barrier_all();
shmem_double_put_nb(vtmp1, ed->v1, ed->nlstates, ed->from_nbs[neighb], NULL);
```

SHMEM “fast”: non-blocking PUT, local wait only

Other message passing paradigms

MPI-2: One-sided PUT

```
MPI_Put(ed->v1, ed->nlstates, MPI_DOUBLE, ed->to_nbs[0], 0, ed->nlstates, MPI_DOUBLE, win1);
MPI_Win_fence( 0, win1);
```

SHMEM: non-blocking PUT

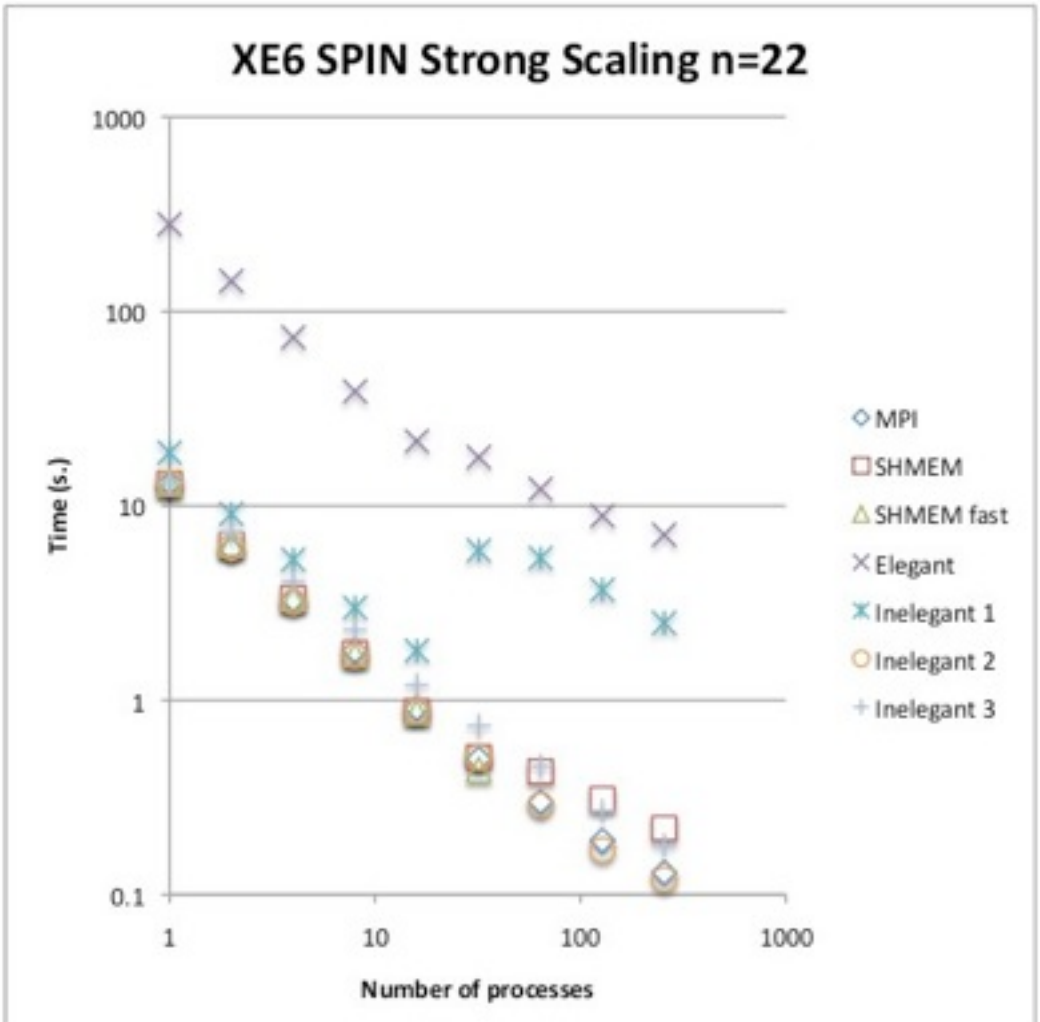
```
vtmp1 = (double *) shmalloc(ed->nlstates*sizeof(double));
  :
shmem_barrier_all();
shmem_double_put_nb(vtmp1, ed->v1, ed->nlstates, ed->from_nbs[neighb], NULL);
```

SHMEM “fast”: non-blocking PUT, local wait only

```
ed->v1[ed->nlstates] = ((double) ed->rank); /* sentinel */
for (l = 0; l < ed->m; ++l) {
  offset = l*(ed->nlstates+1); /* Offset into buffer */
  shmem_double_put_nb(&vtmp[offset],ed->v1, ed->nlstates+1,ed->to_nbs[l],NULL);
}
          :
tag = vtmp[offset+ed->nlstates];
while (tag != (double) ed->from_nbs[k-ed->nm]) { /* spin */
  tag = vtmp[offset+ed->nlstates];
}
for (i = offset, j=0; i < offset+ed->nlstates; ++i, ++j) {
   ed->v2[j] += ed->gamma * vtmp[i];
}
vtmp[l*(ed->nlstates+1)+ed->nlstates]=((double)-1); /*reset*/
```

SPIN strong scaling: Cray XE6, n=22,24; 10 iter.

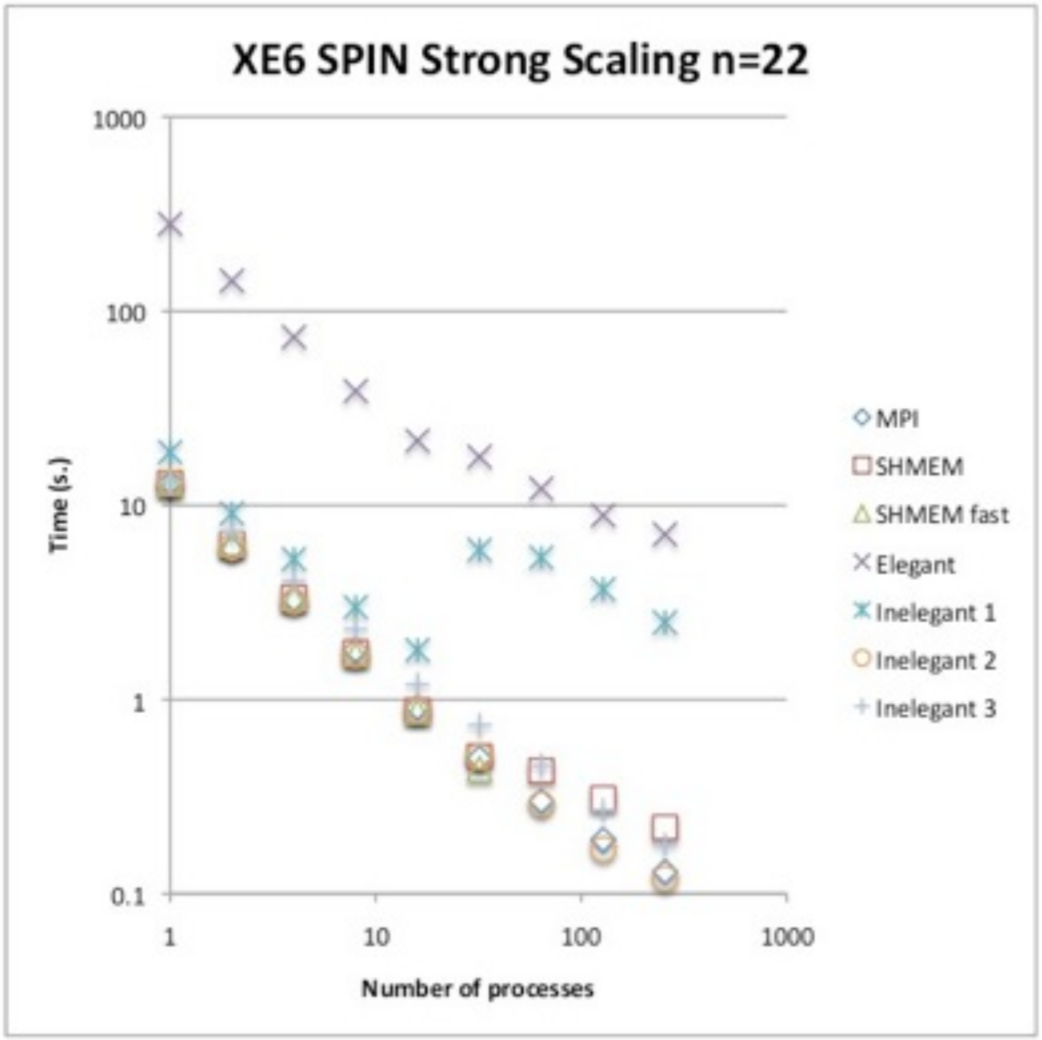
SPIN strong scaling: Cray XE6, n=22,24; 10 iter.

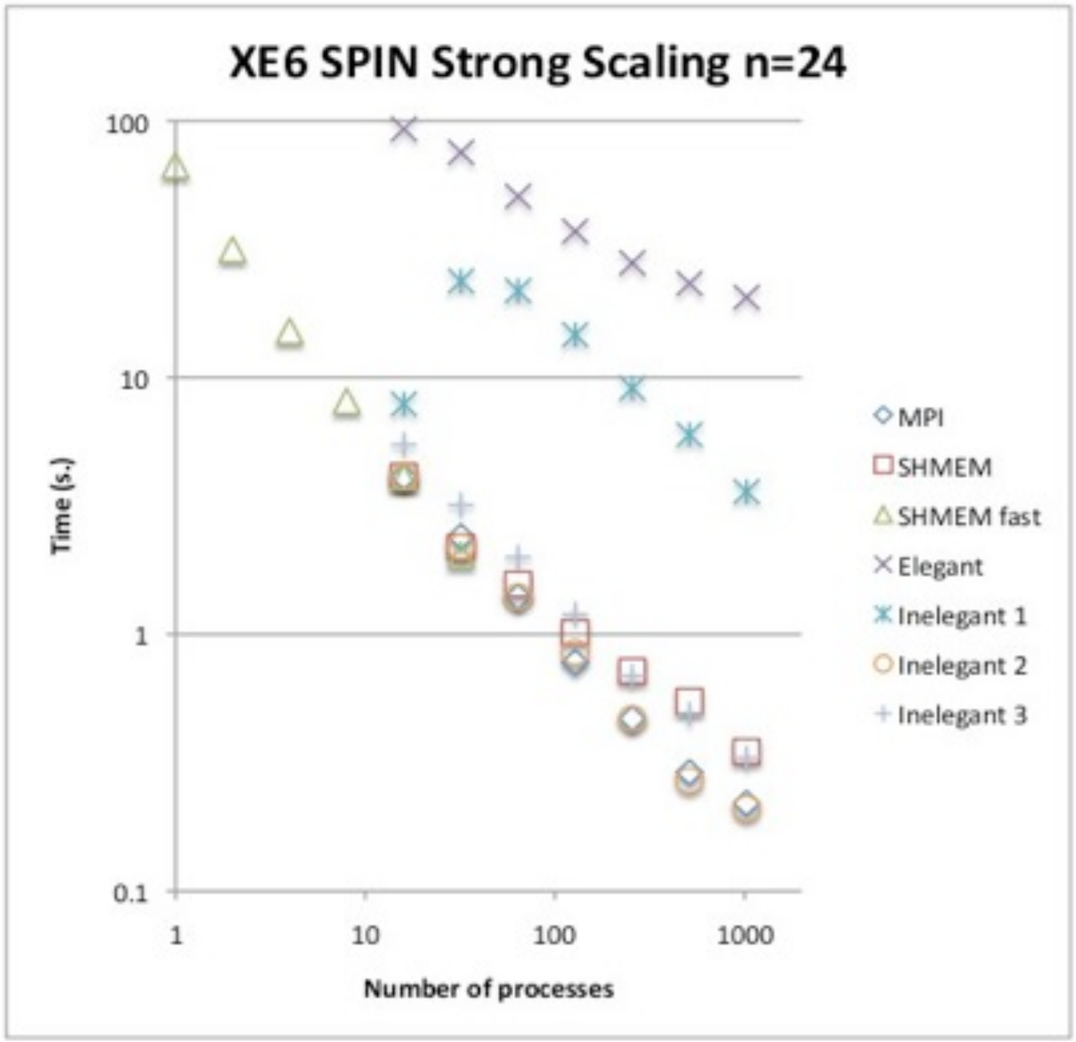


ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

SPIN strong scaling: Cray XE6, n=22,24; 10 iter.

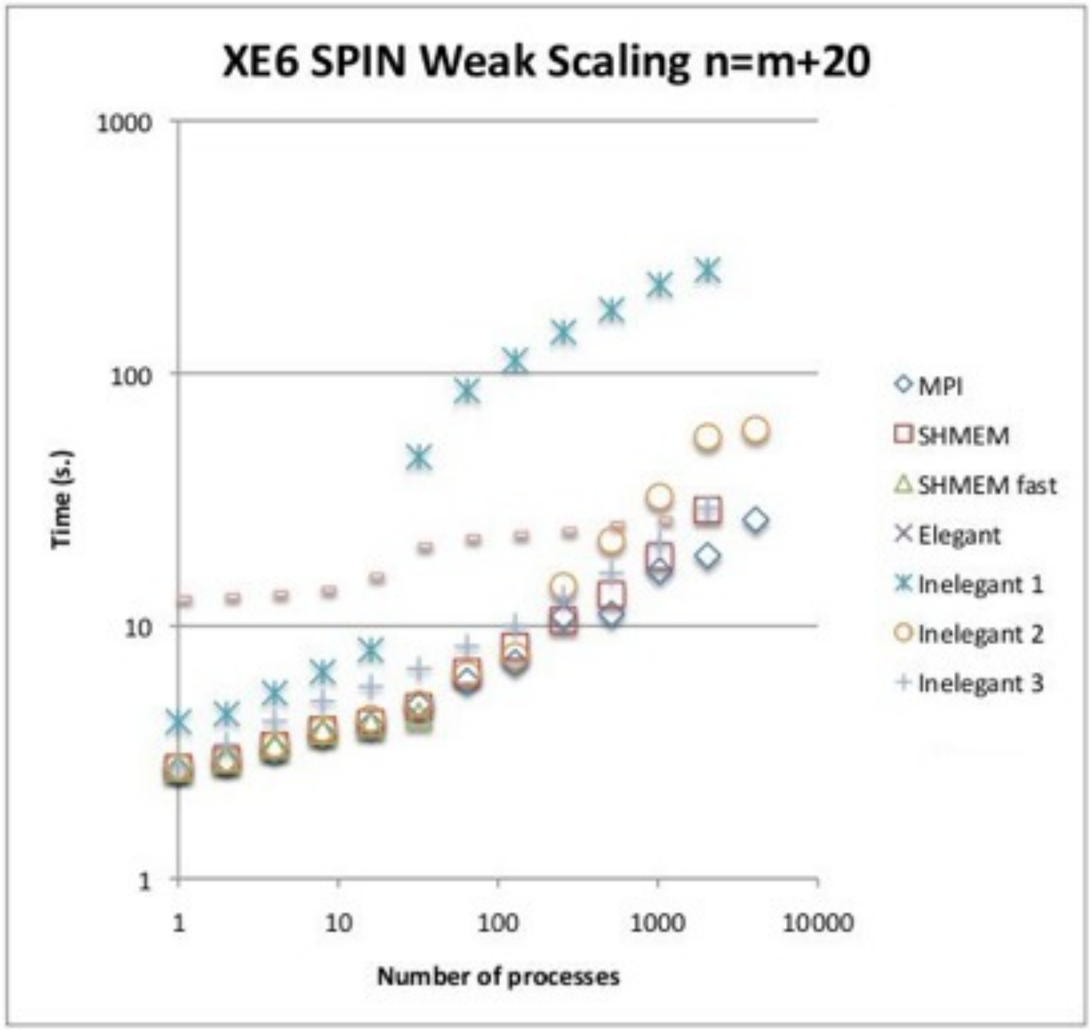




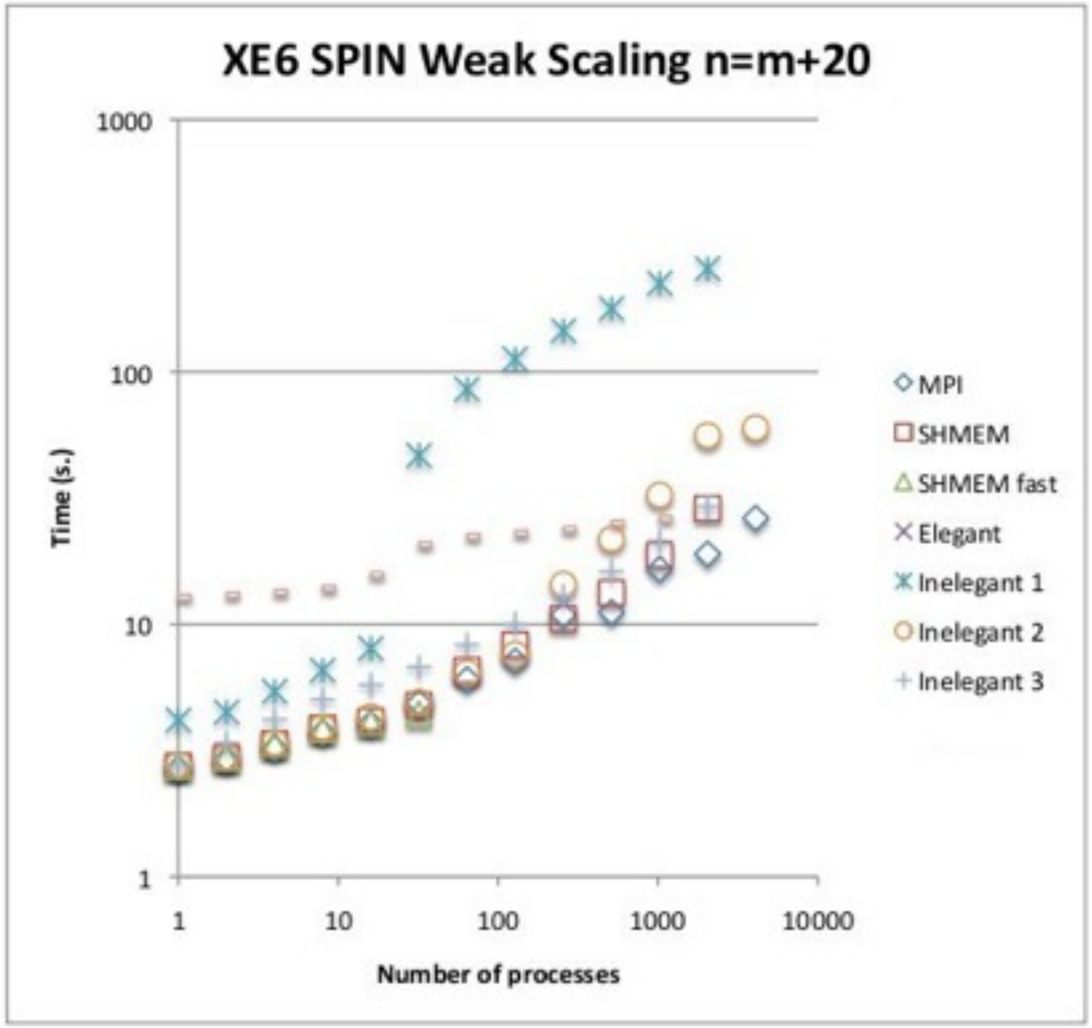
Friday, May 25, 2012

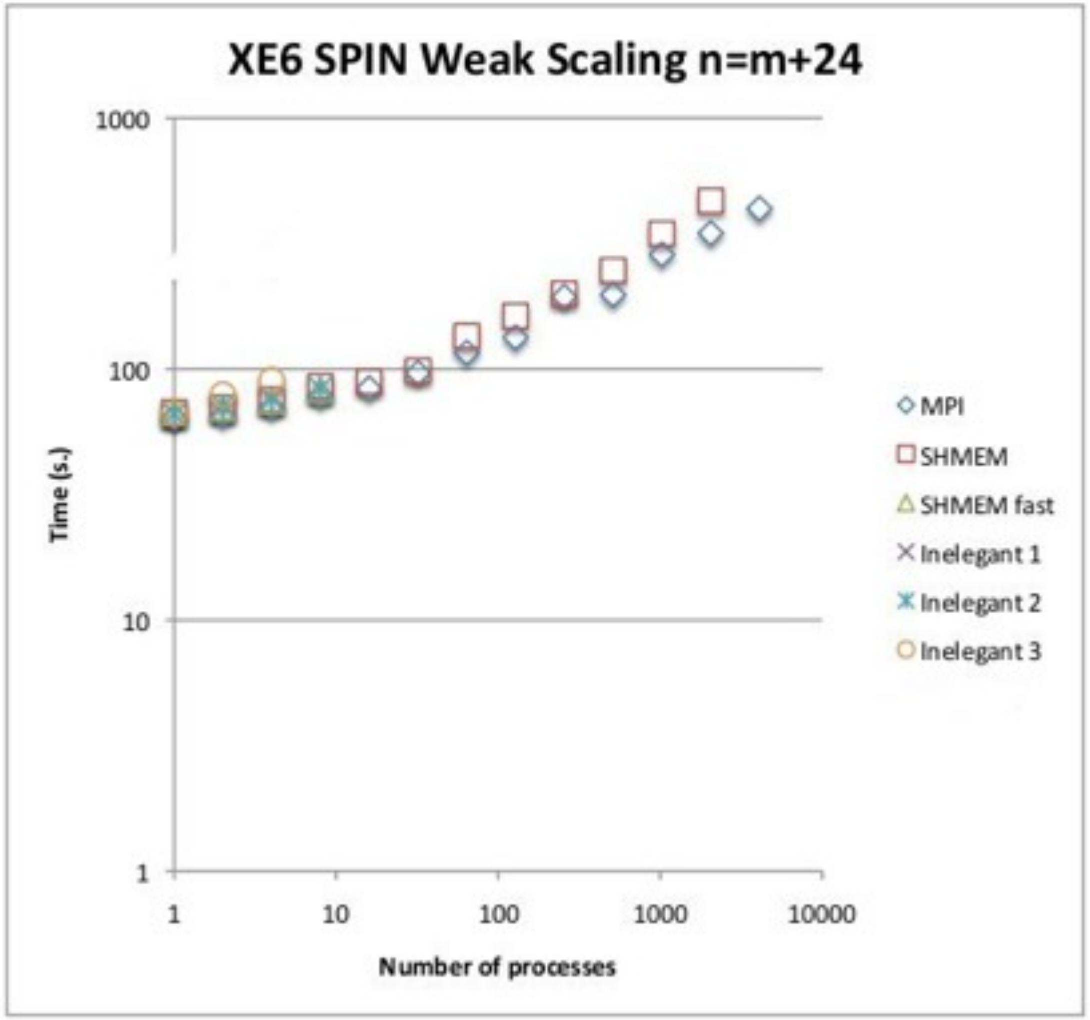
SPIN weak scaling: Cray XE6/Gemini, 10 iterations

SPIN weak scaling: Cray XE6/Gemini, 10 iterations



SPIN weak scaling: Cray XE6/Gemini, 10 iterations





Can UPC perform better than MPI two-sided?

Can UPC perform better than MPI two-sided?

- **Work:** original MPI two-sided version with double buffering

Can UPC perform better than MPI two-sided?

- **Work:** original MPI two-sided version with double buffering
- **Ref_MPI:** naive single buffered version

Can UPC perform better than MPI two-sided?

- **Work:** original MPI two-sided version with double buffering
- **Ref_MPI:** naive single buffered version
- **Opt_MPI:** multiple round-robin buffers utilizing MPI_Isend/Irecv

Can UPC perform better than MPI two-sided?

- **Work:** original MPI two-sided version with double buffering
- **Ref_MPI:** naive single buffered version
- **Opt_MPI:** multiple round-robin buffers utilizing MPI_Isend/Irecv
- **Opt_UPC_Fence:** blocking upc_memput with single fence

Can UPC perform better than MPI two-sided?

- **Work:** original MPI two-sided version with double buffering
- **Ref_MPI:** naive single buffered version
- **Opt_MPI:** multiple round-robin buffers utilizing MPI_Isend/Irecv
- **Opt_UPC_Fence:** blocking upc_memput with single fence
- **Opt_UPC_Fence_each:** blocking upc_memput with fence for each message

Can UPC perform better than MPI two-sided?

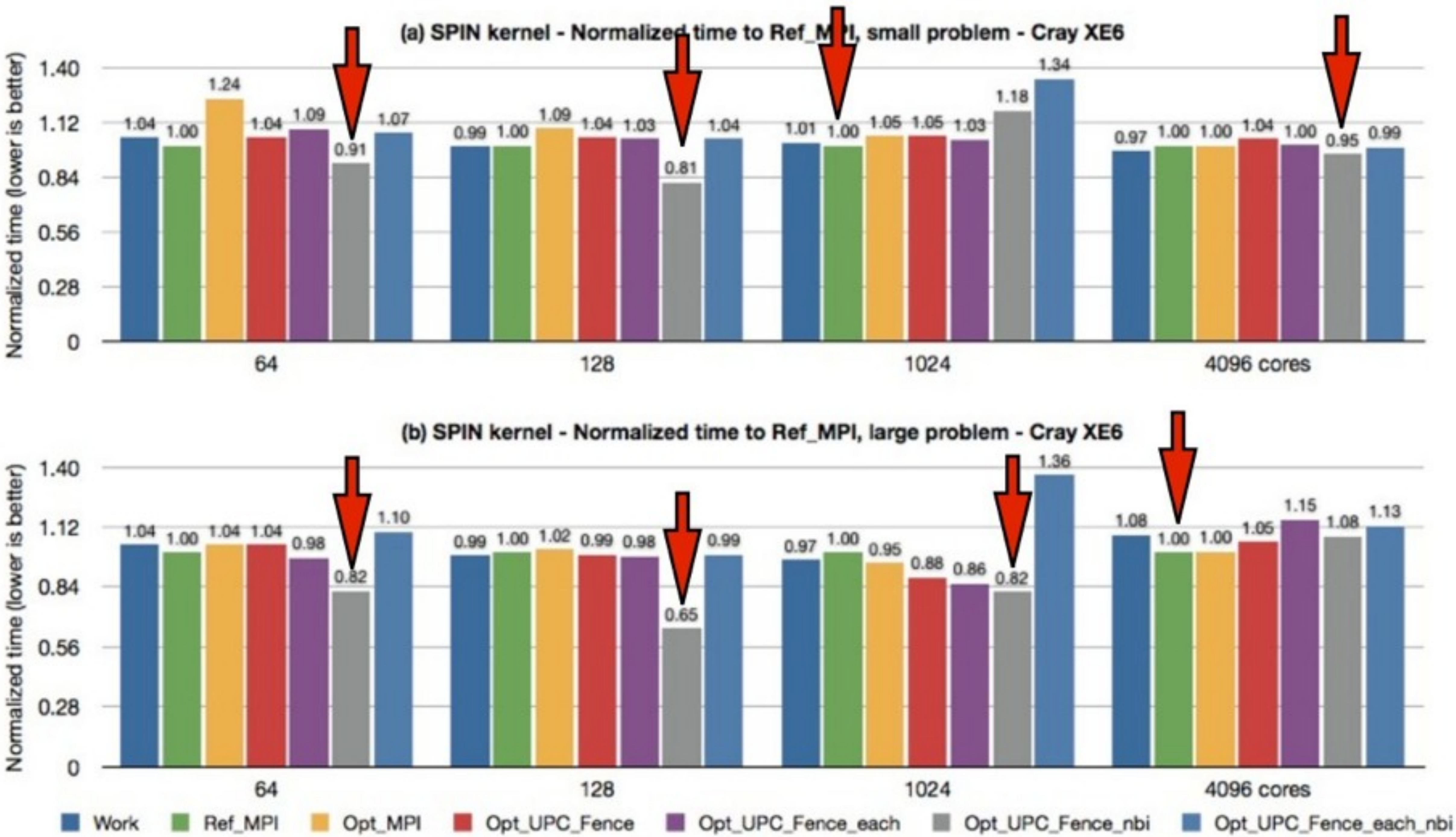
- **Work:** original MPI two-sided version with double buffering
- **Ref_MPI:** naive single buffered version
- **Opt_MPI:** multiple round-robin buffers utilizing MPI_Isend/Irecv
- **Opt_UPC_Fence:** blocking upc_memput with single fence
- **Opt_UPC_Fence_each:** blocking upc_memput with fence for each message
- **Opt_UPC_Fence_nbi:** Cray-specific implicit non-blocking memput with a single fence

Can UPC perform better than MPI two-sided?

- **Work:** original MPI two-sided version with double buffering
- **Ref_MPI:** naive single buffered version
- **Opt_MPI:** multiple round-robin buffers utilizing MPI_Isend/Irecv
- **Opt_UPC_Fence:** blocking upc_memput with single fence
- **Opt_UPC_Fence_each:** blocking upc_memput with fence for each message
- **Opt_UPC_Fence_nbi:** Cray-specific implicit non-blocking memput with a single fence
- **Opt_UPC_Fence_each_nbi:** Cray-specific implicit non-blocking memput with fence for each message

Optimized SPIN normed performance: Cray XE6

Optimized SPIN normed performance: Cray XE6



Take-home messages

Take-home messages

- PGAS languages can express communication elegantly

Take-home messages

- PGAS languages can express communication elegantly
- However, elegant codes tend to be inefficient

Take-home messages

- PGAS languages can express communication elegantly
- However, elegant codes tend to be inefficient
- Inelegant PGAS implementations can outperform MPI
  - On platforms where PGAS is implemented close to the hardware, e.g., Cray XE6, X2
  - Where communication is explicitly formulated as PUTs or GETs, inherently defeating the purpose of the PGAS language

Take-home messages

- PGAS languages can express communication elegantly
- However, elegant codes tend to be inefficient
- Inelegant PGAS implementations can outperform MPI
  - On platforms where PGAS is implemented close to the hardware, e.g., Cray XE6, X2
  - Where communication is explicitly formulated as PUTs or GETs, inherently defeating the purpose of the PGAS language
- PGAS is worthwhile to keep in mind, but

Take-home messages

- PGAS languages can express communication elegantly
- However, elegant codes tend to be inefficient
- Inelegant PGAS implementations can outperform MPI
  - On platforms where PGAS is implemented close to the hardware, e.g., Cray XE6, X2
  - Where communication is explicitly formulated as PUTs or GETs, inherently defeating the purpose of the PGAS language
- PGAS is worthwhile to keep in mind, but
- Currently the investment of changing paradigms does not seem worthwhile

Thank you for your attention!

wsawyer@cscs.ch