

# Parallelization of the Dirac operator

Pushan Majumdar

*Indian Association for the Cultivation of Sciences, Jadavpur, Kolkata*

# Outline

- Introduction
- Algorithms
- Parallelization
- Comparison of performances
- Conclusions

The use of QCD to describe the strong force was motivated by a whole series of experimental and theoretical discoveries made in the 1960's and 70's.

successful classification of particles by the quark model  
multiplicities of certain high-energy strong interactions.

QCD is a non-Abelian gauge theory with gauge group SU(3).  
The matter fields describe “quarks”, which are spin 1/2 objects transforming under the fundamental representation of SU(3).

The QCD lagrangian is :  $S = \bar{\psi} \not{D}(A) \psi + F^2(A)$ .

Till date QCD is our best candidate theory for describing strong interactions.

Observables in non-Abelian gauge theories

$$\langle \mathcal{O} \rangle = Z^{-1} \int \mathcal{D}A \mathcal{D}\bar{\psi} \mathcal{D}\psi \mathcal{O} e^{-S} \quad (1)$$

where  $S = \bar{\psi} \not{D}(A) \psi + F^2(A)$ .

$S$  is quartic in  $A$ . Functional integral cannot be done analytically.

Two options :

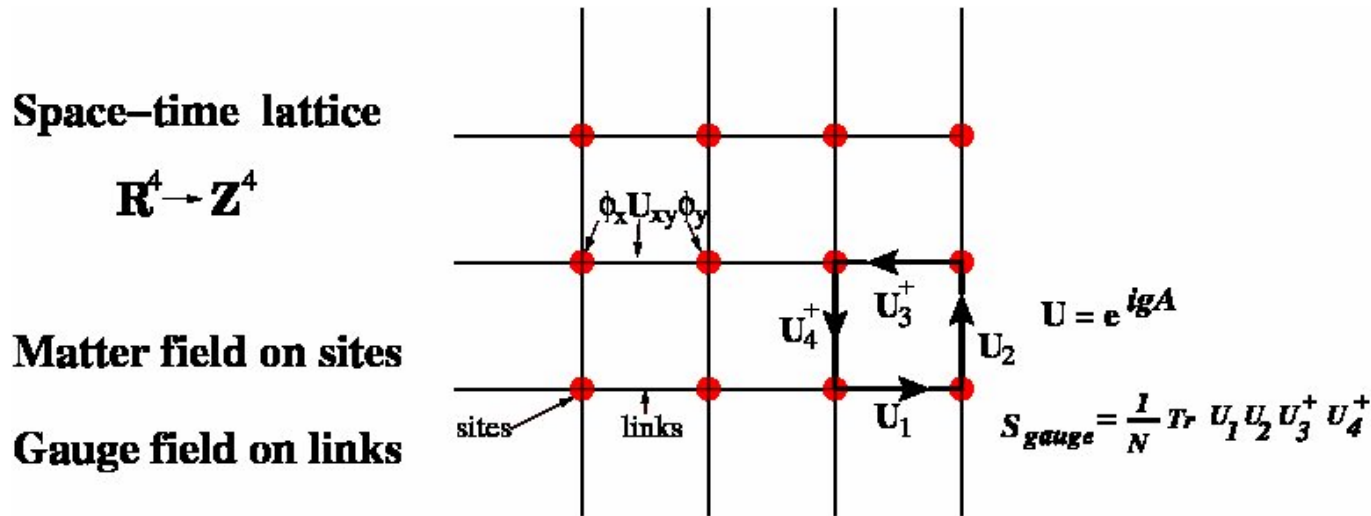
- 1) Expand  $e^{-S}$  as a series in  $g$ .
- 2) Estimate the integral numerically.

(1) works for small values of  $g$  and leads to perturbative QCD.

(2) is fully non-perturbative, but requires a discretization for implementation. This leads to lattice gauge theory.

**The value of  $g$  depends upon the energy with which you probe the system**

Wilson's LGT : Discretization maintaining gauge invariance.



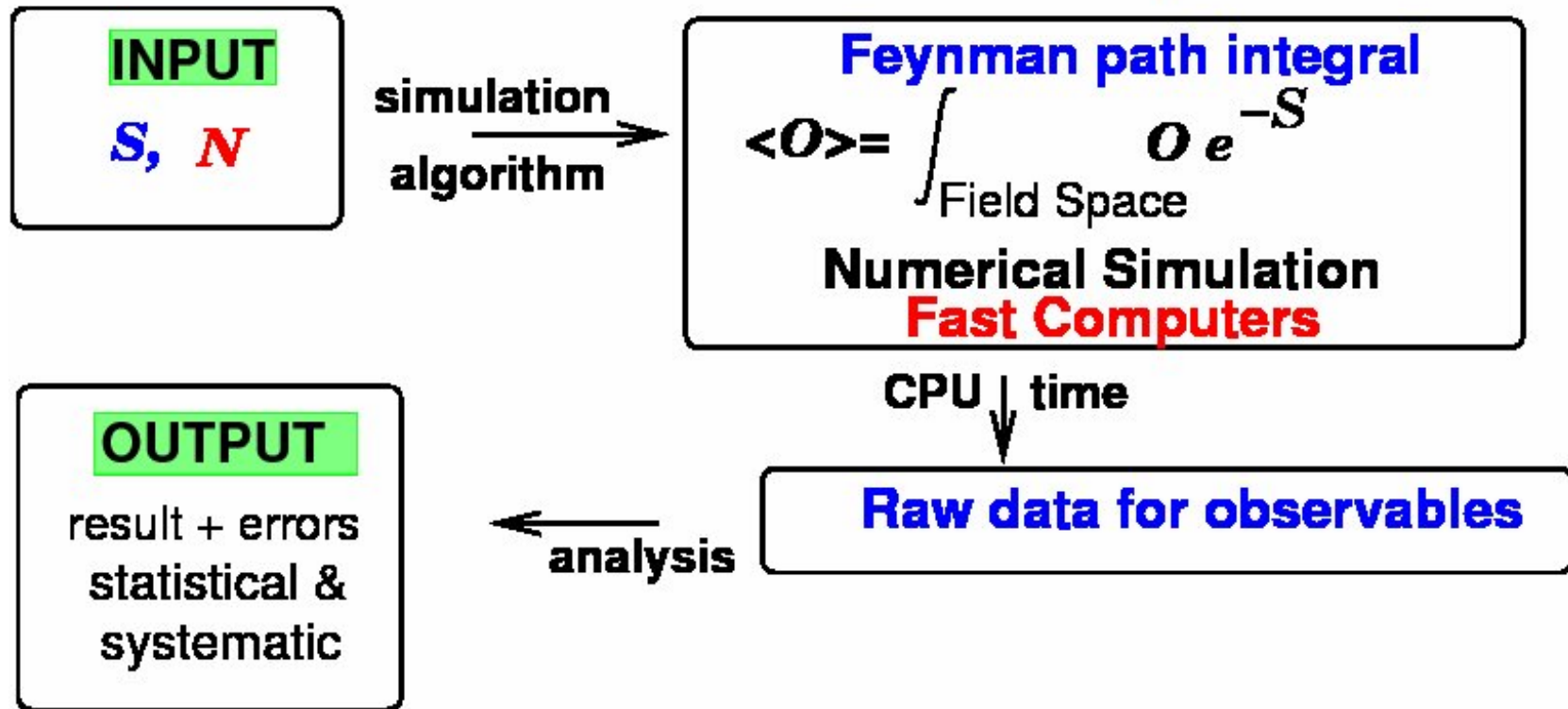
finite lattice spacing : UV cut off      finite lattice size : IR cut off

Periodic boundary conditions to avoid end effects  $\Rightarrow \mathbf{T}^4$  topology.

Finally one must extrapolate to zero lattice spacing.

**ACTION :**  $S(\mathbf{Fields}(x))$  , *bare couplings*)

**Finite Lattice** (Finite degrees of freedom) :  $N^4$  points



dimension of the integral :  $4 \times N^4$

domain of integration : manifold of the group SU(3)

Not possible to do exactly – use sampling techniques

## Molecular Dynamics

Euclidean path integral for quantum theory  $\Rightarrow$  Partition function for classical statistical mechanical system in 4 space dimensions. Dynamics in simulation time.

$$\langle O \rangle_{\text{can}}(\beta) \xrightarrow[\text{limit}]{\text{thermodynamic}} [\langle O \rangle_{\text{micro}}]_{E=\bar{E}(\beta)} \xrightarrow{\text{ergodic}} \lim_{T \rightarrow \infty} \frac{1}{T} \int_0^T d\tau O(\phi_i(\tau))$$

Canonical ensemble average  $\rightarrow$  microcanonical ensemble average at  $E = \bar{E}(\beta)$  (equilibrium temperature)  $\rightarrow$  Time average over classical trajectories.

$$\frac{d^2 \phi_i}{d\tau^2} = -\frac{\partial S[\phi]}{\partial \phi_i}$$

(configuration  $\phi$  has energy  $\bar{E}$ ).

Complexity of motion in higher dimension  $\longrightarrow$  Quantum fluctuations in original theory

$$\langle O \rangle = \frac{1}{Z} \int \mathcal{D}\phi O[\phi] e^{-S[\beta, \phi]} \quad (\text{Original theory})$$

$$\langle O \rangle = \frac{1}{\tilde{Z}} \int \mathcal{D}\phi \mathcal{D}\pi O[\phi] e^{-\frac{1}{2} \sum_i \pi_i^2 - S[\beta, \phi]} \quad (\text{with auxiliary field } \pi).$$

$$\mathcal{H}^{4-\text{dim}} = \frac{1}{2} \sum_i \pi_i^2 + S[\beta, \phi]$$

Equations of motion:

$$\dot{\phi}_i = \frac{\partial \mathcal{H}[\phi, \pi]}{\partial \pi_i} ; \quad \dot{\pi}_i = -\frac{\partial \mathcal{H}[\phi, \pi]}{\partial \phi_i}$$

Problems:

Not ergodic. Not possible to take the thermodynamic limit.

## Hybrid Algorithms

### Langevin dynamics + Molecular Dynamics

Uses MD to **move fast through configuration space**, but **choice of momenta from a random distribution** (Langevin step) makes it ergodic.

Implementation:

- (i) **Choose  $\{\phi_i\}$  in some arbitrary way.**
- (ii) **Choose  $\{\pi_i\}$  from some Gaussian (random) ensemble.**

$$P\{\pi_i\} = \left( \prod_i \frac{1}{\sqrt{2\pi}} \right) e^{-\frac{1}{2} \sum_i \pi_i^2}.$$

- (iii) **Evaluate  $\tilde{\pi}_i(1) = \pi_i(1 + \epsilon/2)$**
- (iv) **Iterate the MD equations for a few steps and store  $\phi_i(n)$ .  
(To be used as  $\phi_i(1)$  for the next MD update.)**
- (v) Go back to step (ii) and repeat.

**Systematic error introduced by finite time step remains.**

## Hybrid Monte Carlo

Hybrid algorithm + Metropolis accept/reject step.

Implementation:

(0) Start with a configuration  $\{\phi_i^1, \pi_i^1\}$ .

(i)-(iv) Do the first 4 steps as in the Hybrid algorithm to get to  $\{\phi_i^N, \pi_i^N\}$  after  $N$  MD steps.

(v) Accept the configuration  $\{\phi_i^N, \pi_i^N\}$  with probability

$$p_{\text{acc}} = \min \left\{ 1, \frac{\exp(-\mathcal{H}[\phi_i^N, \pi_i^N])}{\exp(-\mathcal{H}[\phi_i^1, \pi_i^1])} \right\}$$

(vi) If  $\{\phi_i^N, \pi_i^N\}$  is not accepted keep  $\{\phi_i^1, \pi_i^1\}$  and choose new momenta. Otherwise make  $\{\phi_i^N, \pi_i^N\}$  the new  $\{\phi_i^1, \pi_i^1\}$  and choose new momenta.

This algorithm satisfies detailed balance (necessary for equilibrium distribution) and eliminates error due to finite time step.

## Equilibration

$\beta = 5.2$ ,  $am = 0.03$   
from top to bottom:

mean plaquette ;

# of CG iterations  
for accept/reject step ;

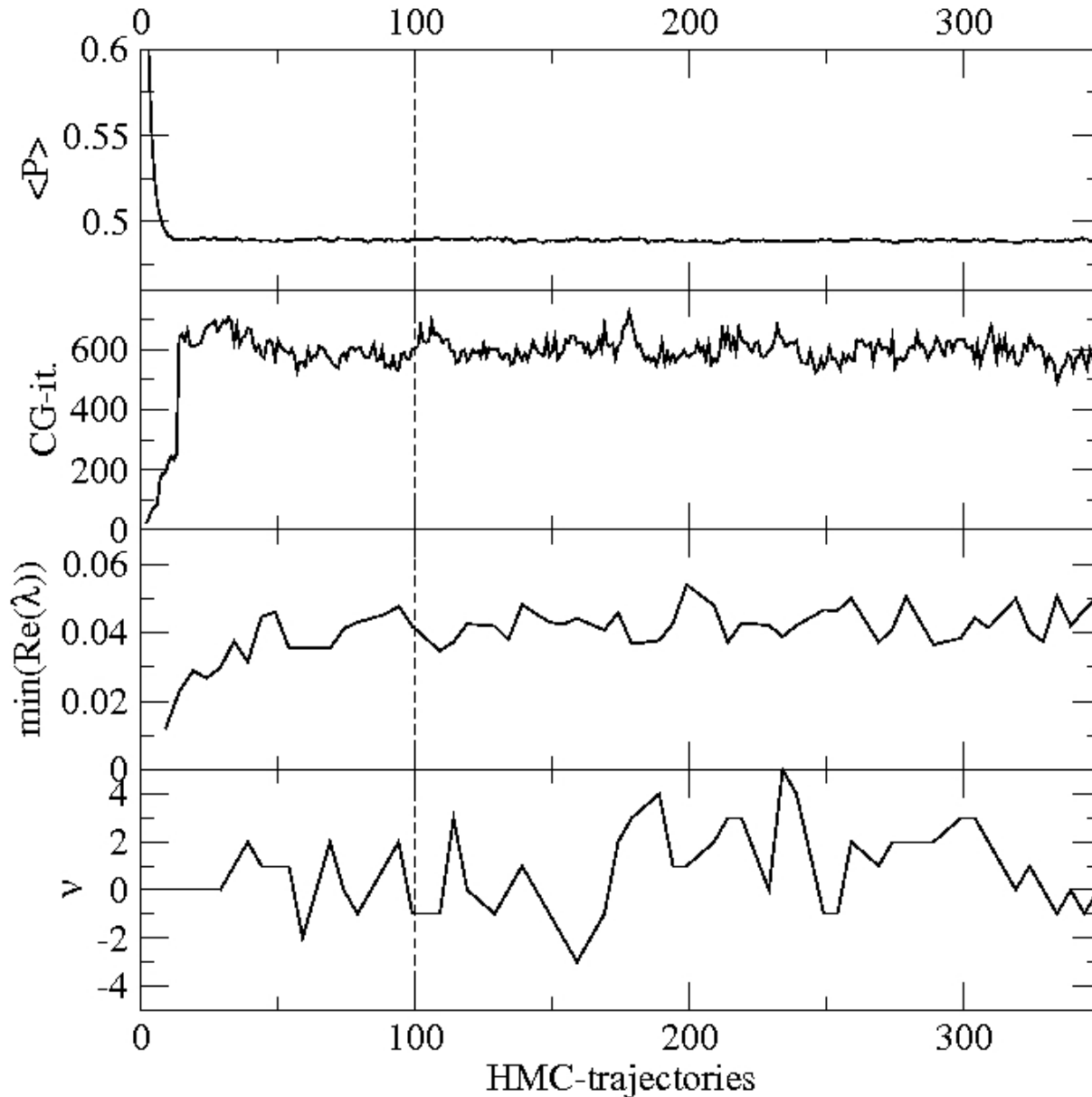
$\min(\text{Re}(\lambda))$

$\lambda$ : eigenvalue of Dirac  
operator.,

topological charge

$\nu$

The lower two rows' values have been determined only for every 5th configuration.



## Hybrid Monte Carlo - Parallelization

- **Generate the Gaussian noise** (random momenta)  
Cheap.
- **Integrate the Hamiltonian equations of motion.**  
Integration done using the leap-frog integration scheme. Improved integration schemes viz. Sexton-Weingarten, Omelyan etc. exist.  
Expensive. Target for parallelization.
- **Accept/Reject step** : Corrects for numerical errors introduced during integration of the equations of motion.

## Example : Lattice gauge theory

Degrees of freedom : SU(3) matrices & complex vector fields.

For two flavours of mass degenerate quarks :

$$\mathcal{H} = \frac{1}{2} \text{tr} p^2 + \text{Gauge Action}(\mathcal{U}) + \phi^\dagger (\mathcal{M}^\dagger \mathcal{M})^{-1}(\mathcal{U}) \phi$$

where  $\mathcal{M}$  is a matrix representation of the Dirac operator and  $\mathcal{U}$ 's are SU(3) matrices.

Hamiltonian equations of motion update  $p$ 's and  $\mathcal{U}$ 's.

$\phi$  is held constant during the molecular dynamics.

Equations of motion are :  $\dot{\mathcal{U}} = ip\mathcal{U}$  and  $\partial\mathcal{H}/\partial t = 0$

## The conjugate gradient algorithm

Most expensive part of calculation is evaluation of

$$\frac{\partial}{\partial t} \phi^\dagger (\mathcal{M}^\dagger \mathcal{M})^{-1} \phi$$

which requires the evaluation of  $(\mathcal{M}^\dagger \mathcal{M})^{-1} \phi$  at every molecular dynamics step.

For a hermitian matrix  $A$ , the inversion is best done by conjugate gradient.

If we want to evaluate  $x = A^{-1}b$ , we can as well solve the linear system  $Ax = b$  or minimize the quadratic form  $|Ax - b|$ .

Let  $\{p_n\}$  be a sequence of conjugate vectors. ( $\langle p_i, Ap_j \rangle = 0$ )

Then  $\{p_n\}$  forms a basis in  $R^n$ .

Let the system  $Ax = b$  be solved by  $x_* = \alpha_i p_i$ .

$$\text{Then } \alpha_i = \frac{\langle p_i, b \rangle}{\langle p_i, Ap_i \rangle}$$

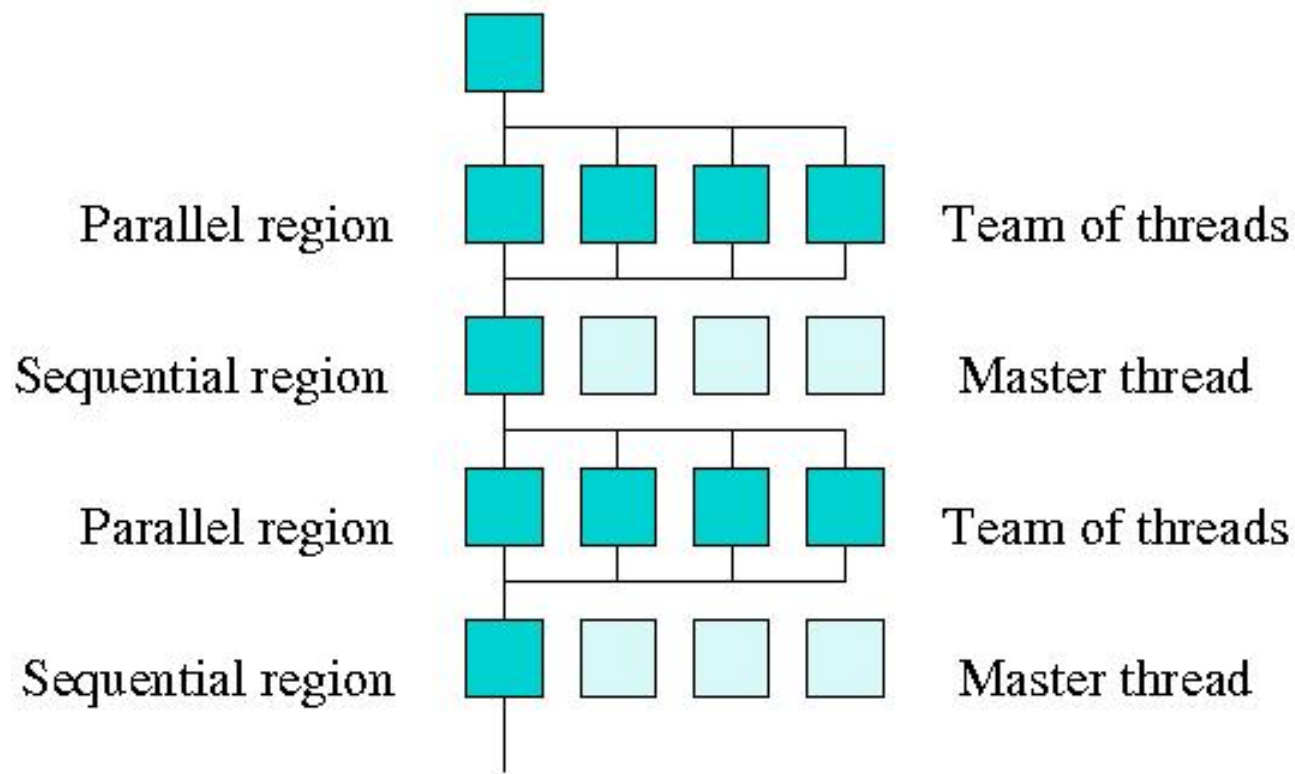
Requires repeated application of  $A$  on  $p_i$  and then dot products between various vectors.

Objective : Try to parallelize the matrix-vector multiplication  $Ap$

Two options : **OpenMP** & **MPI**

# What is OpenMP ???

Master – Worker team thread pattern



OpenMP concentrates on parallelizing loops.

Compiler needs to be told which loops can be safely parallelized.

### Evaluation of $b = Da$

```
!$OMP PARALLEL DO
  do isb = 1,nsite
    do idb = 1,4
      do icb = 1,3
        b(icb,idb,ism) = diag*a(icb,idb,ism)
      end do
    end do
  end do
!$OMP END PARALLEL DO
```

```

!$OMP PARALLEL DO PRIVATE(isb,ida,ica,idb,icb,idir,isap,isam,uu, &
    & uudag) SHARED(a,b)
do isb = 1,nsite
do idir=1,4

    isap=neib(idir,isb)
    isam=neib(-idir,isb)

    call uextract(uu,idir,isb)
    call udagextract(uudag,idir,isam)

internal loop over colour & dirac indices (ida,ica,idb,icb)
    b(icb,idb,isb) = b(icb,idb,isb) &
    &    -gp(idb,ida,idir)*uu(icb,ica)*a(ica,ida,isap) &
    &    -gm(idb,ida,idir)*uudag(icb,ica)*a(ica,ida,isam)

    enddo
enddo
!$OMP END PARALLEL DO

```

## Pros:

- Simple
- Data layout and decomposition is handled automatically by directives.
- Unified code for both serial and parallel applications
- Original (serial) code statements need not, in general, be modified when parallelized with OpenMP.

## Cons:

- Currently only runs efficiently in shared-memory multiprocessor platforms
- Requires a compiler that supports OpenMP.
- Scalability is limited by memory architecture.
- Reliable error handling is missing.

## What is MPI?

A standard protocol for passing messages between parallel processors.

Small: **Require only six library functions to write any parallel code.**

Large: **There are more than 200 functions in MPI-2.**

It uses “communicator” a handle to identify a set of processes and “topology” for different pattern of communication.

**A communicator is a collection of processes that can send messages to each other.**

Can be called by Fortran, C or C++ codes.

Small MPI:

```
MPI_Init MPI_Comm_size MPI_Comm_rank MPI_Send MPI_Recv MPI_Finalize
```

```
program hmc_main
  use paramod
  use params
  implicit none
  integer i, ierr, numprocs
  include "mpif.h"

  call MPI_INIT( ierr )
  call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
  call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

  print *, "Process ", rank, " of ", numprocs, " is alive"
  call init_once
  call do_hmc_wilson

  call MPI_FINALIZE(ierr)
end program hmc_main
```

```
mpirun -n 8 ./executable < input > out &
```

```

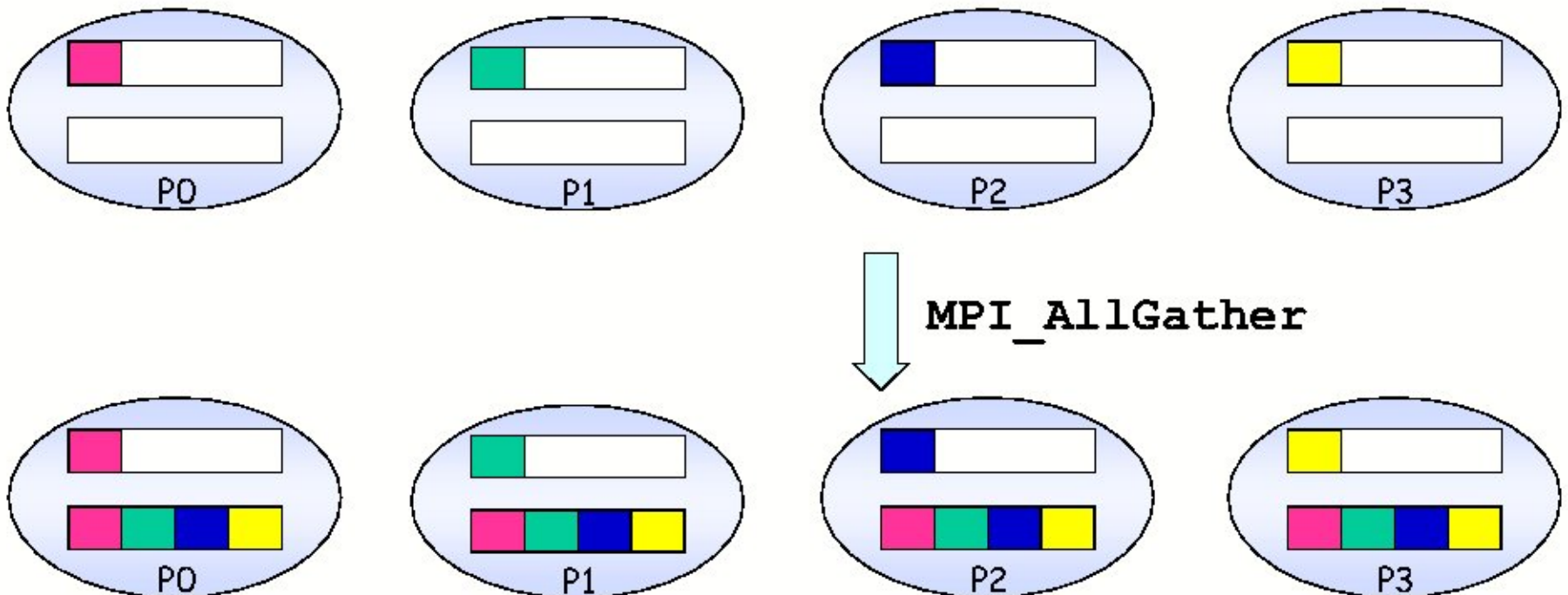
subroutine ddagonvec(b,a)  ! Evaluates  $b = D a$ .
  base=rank*nsb
  do isb = 1,nsb
loop over colour and dirac indices (icb,idb)
      b(icb,idb,isb) = diag*a(icb,idb,isb+base)
  end do
  do isb = base+1,base+nsb
    do idir=1,4
      isap=neib(idir,isb)
      isam=neib(-idir,isb)
      call uextract(uu,idir,isb)
      call udagextract(uudag,idir,isam)
loop over internal colour & dirac indices (ida,ica,idb,icb)
      b(icb,idb,isb-base) = b(icb,idb,isb-base) &
&          -gp(idb,ida,idir)*uu(icb,ica)*a(ica,ida,isap) &
&          -gm(idb,ida,idir)*uudag(icb,ica)*a(ica,ida,isam)
    enddo
  enddo
  return
end subroutine ddagonvec

```

```
subroutine matvec_wilson_ddagd(vec,res)
  use lattsize
  use paramod
  include "mpif.h"
  implicit none
  complex(8), dimension(blocksize*nsite), intent(in) :: vec
  complex(8), dimension(blocksize*nsite) :: vec1
  complex(8), dimension(blocksize*nsite), intent(out) :: res
  complex(8), dimension(blocksize*nsub) :: res_loc1,res_loc2
  integer :: ierr
  call donvec(res_loc1,vec)
  call MPI_ALLGATHER(res_loc1,blocksize*nsub,MPI_DOUBLE_COMPLEX, &
    vec1,blocksize*nsub,MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
  call ddagonvec(res_loc2,vec1)
  call MPI_ALLGATHER(res_loc2,blocksize*nsub,MPI_DOUBLE_COMPLEX, &
    res,blocksize*nsub,MPI_DOUBLE_COMPLEX,MPI_COMM_WORLD,ierr)
end subroutine matvec_wilson_ddagd
```

# AllGather

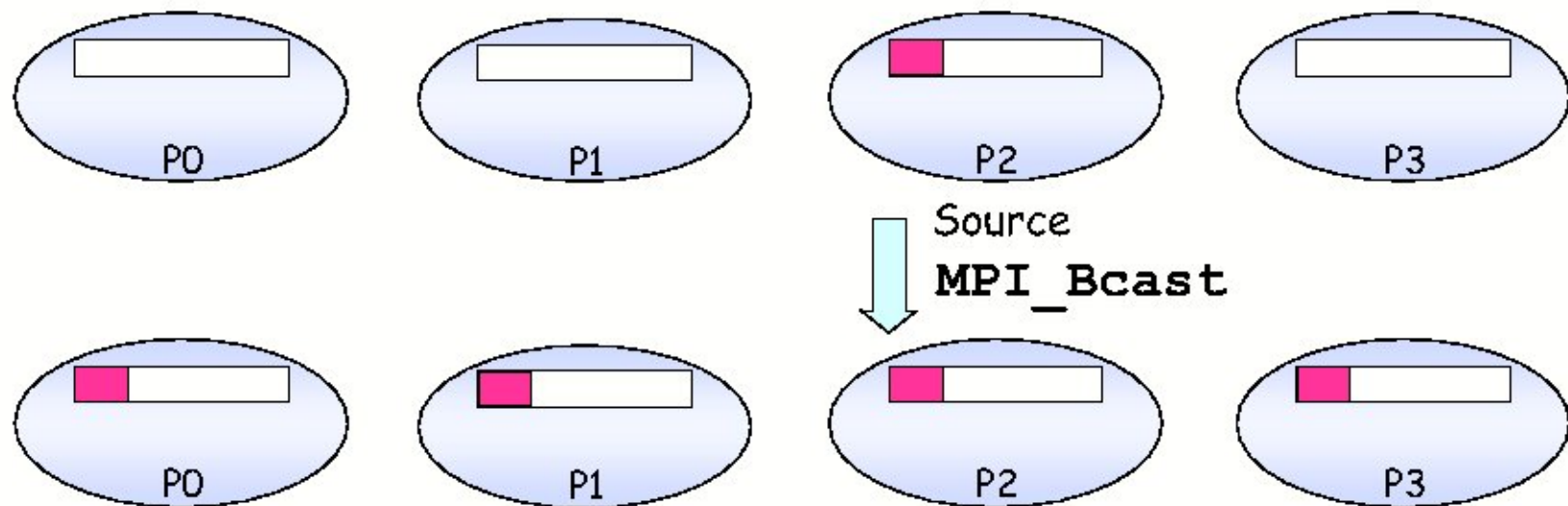
```
Call MPI_Allgather(sendbuf, sendcount, senddatatype,  
recvbuf, recvcount, recvdatatype, MPI_comm)
```




## Broadcast

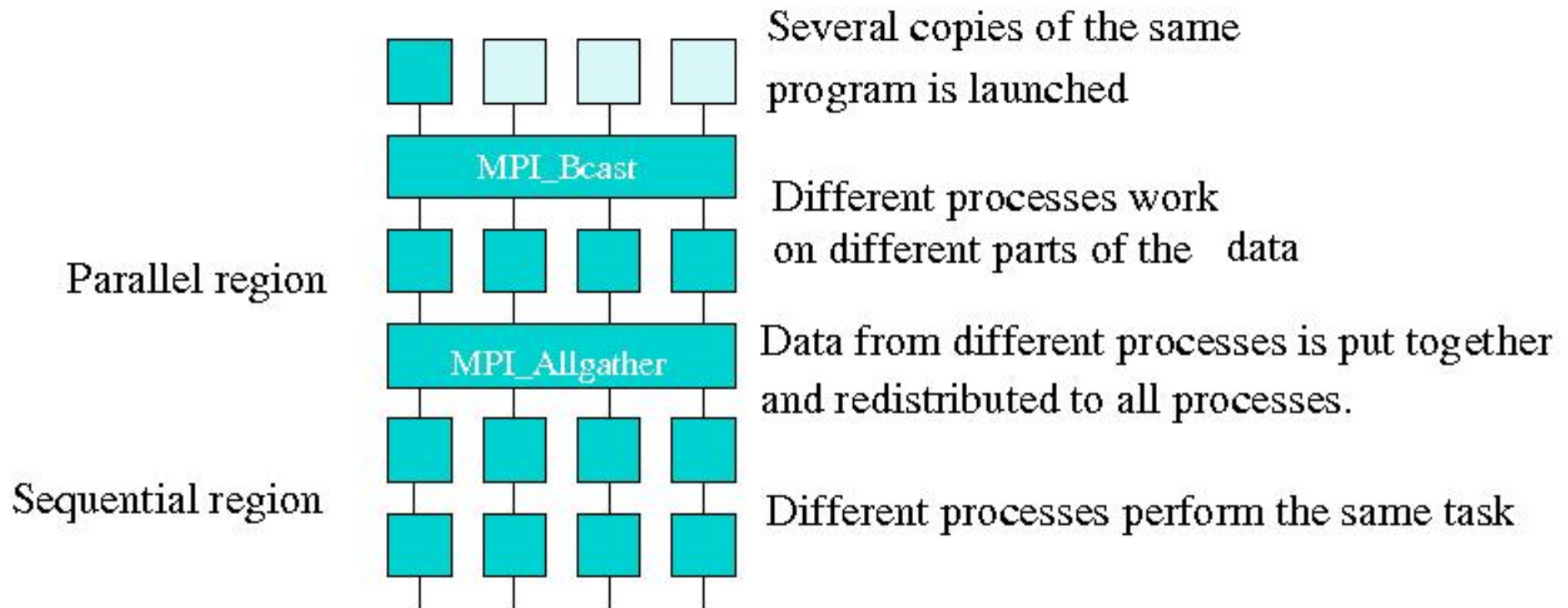
Call `MPI_Bcast(buf, count, datatype, source, MPI_Comm, Err)`

- Sends data stored in buffer `buf` of process `source` to all the other processes in the group `comm`



Note:  contains `count * datatype` elements

## MPI flow of program



## Performance comparisons

**Amdahl's Law:**

$$S = \frac{1}{f + (1 - f)/P}$$

**S:** Speed up factor over serial code

**f:** Fraction of code that is serial

**P:** Number of processes

If  $f = 0.1$ , then even if  $P \rightarrow \infty$ ,  $S \leq 10$ .

Similarly if  $f = 0.2$  then  $S \leq 5$ .

Tests were performed on a IBM p-590 which has 32 cores and 64 GB memory.

Each core can handle 2 threads giving 64 logical CPU's.

Compiler used was threadsafe IBM fortran compiler xlf90.

Lattice	OpenMP no. of threads	MPI no. of processes
$4^4$	2 4 8	2 4 8
$8^4$	2 4 8	2 4 8
$12^4$	2 4 8 16	2 4 8 16
$16^4$	2 4 8 16	2 4 8 16

All timings are for 5 full MD trajectories + Accept/Reject step.

Conjugate Gradient tolerance was kept at  $10^{-10}$ .

Each MD trajectory involved 50 integration steps with a time step of 0.02.

### Typical inversion time

OMP	Lattice	Serial	2 threads	4 threads	8 threads	16 threads
	$12^4$	10.67	11.86/2	14.32/4	11.66/8	13.64/16
	$16^4$	46.57	43.25/2	41.68/4	62.75/8	59.45/16

MPI	Lattice	Serial	2 procs	4 procs	8 procs	16 procs
	$12^4$	10.67	6.52	4.09	2.74	2.26
	$16^4$	46.57	31.13	15.35	12.11	10.52

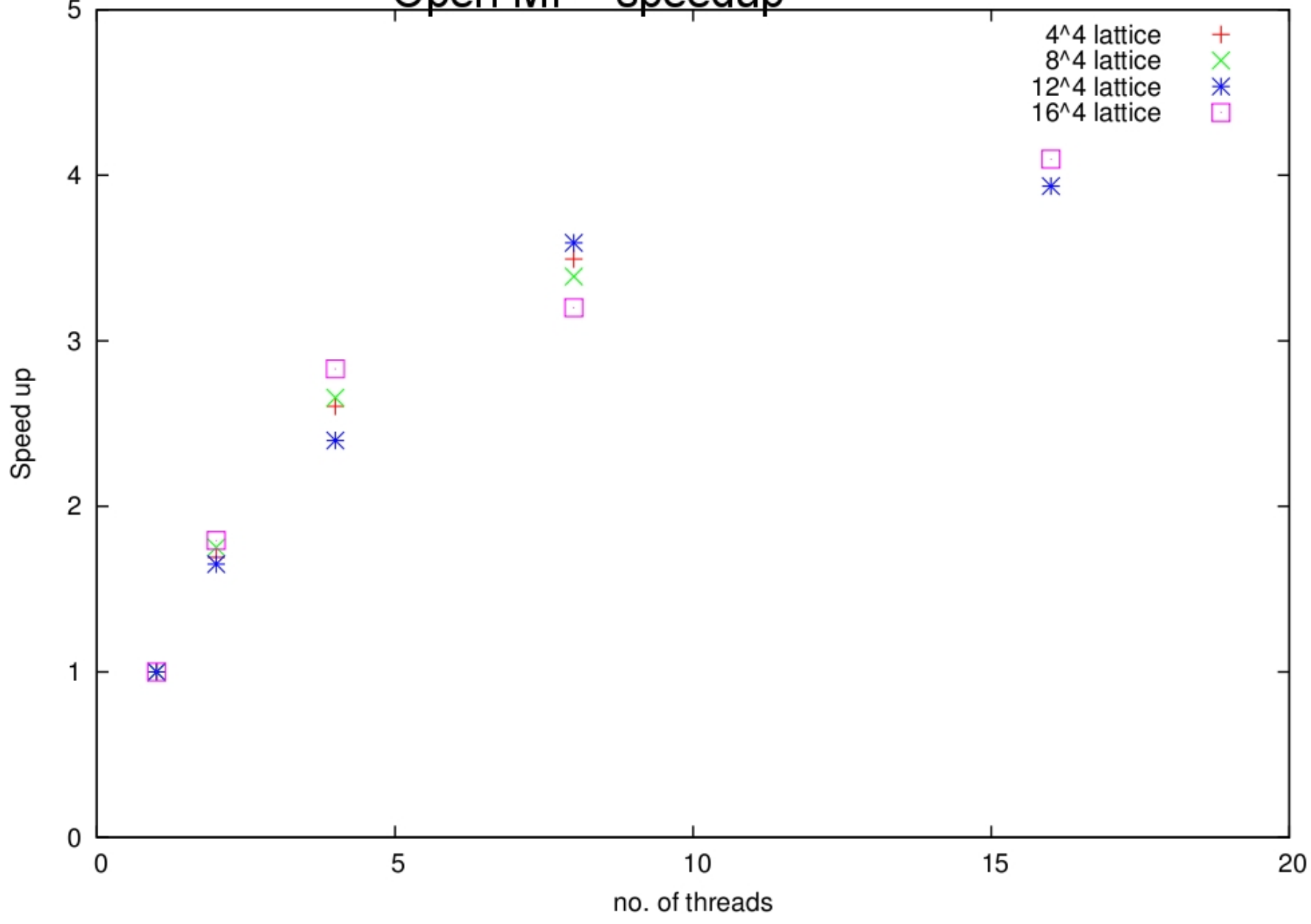
### Typical force calculation time

OMP	Lattice	Serial	2 threads	4 threads	8 threads	16 threads
	$12^4$	2.99	2.97/2	4.28/4	2.95/8	3.23/16
	$16^4$	9.52	9.44/2	9.43/4	13.21/8	12.73/16

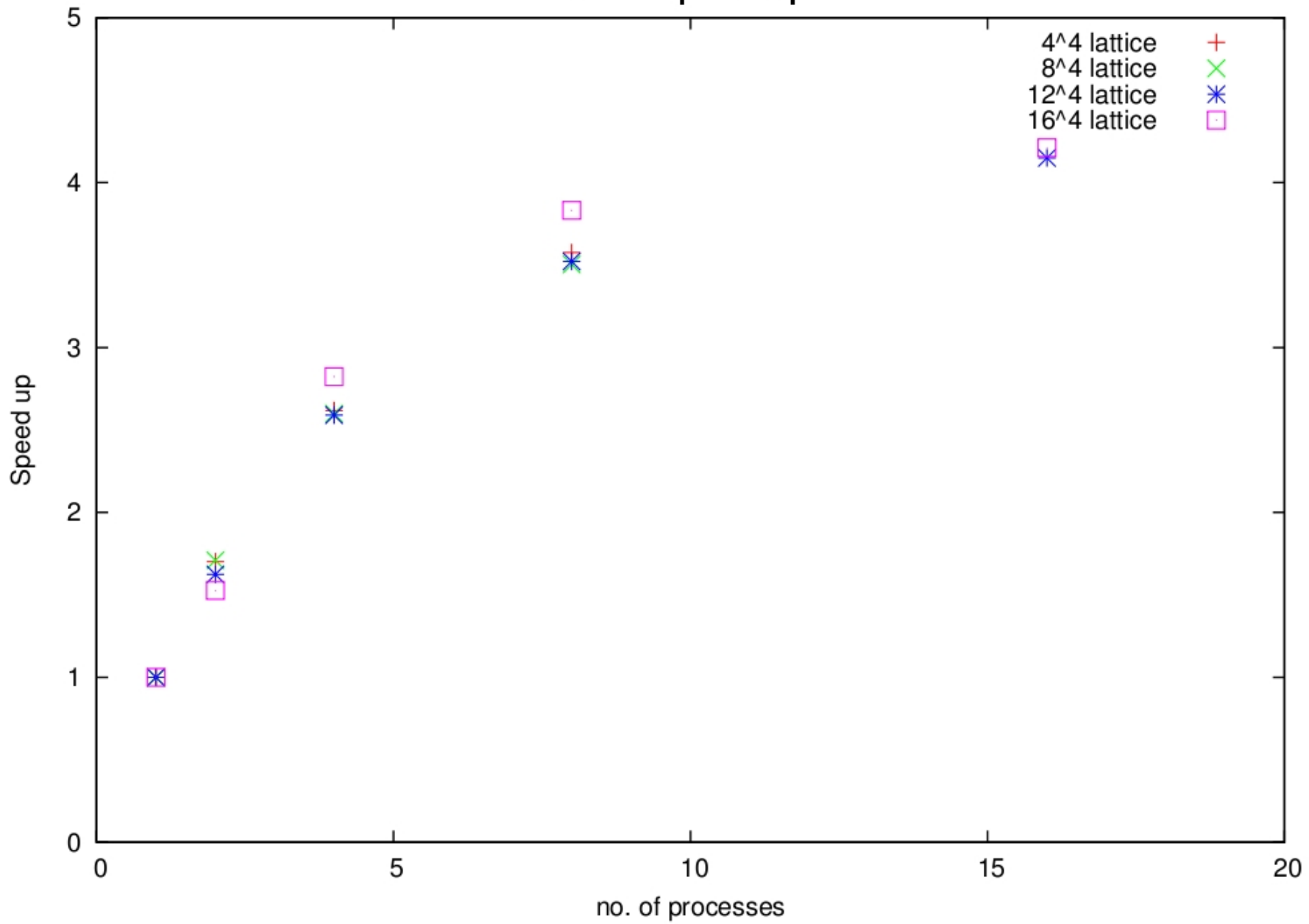
MPI	Lattice	Serial	2 procs	4 procs	8 procs	16 procs
	$12^4$	2.99	1.56	0.82	0.43	0.25
	$16^4$	9.52	6.36	2.57	1.38	0.84

Reference:  $(16/12)^5 = 4.213$   $(16/12)^4 = 3.16$

# Open MP speedup



# MPI speedup



## Conclusions

- Both MPI and OpenMP parallelizations show a speedup by about a factor four using 16 processes.
- Applying Amdahl's law we see that our parallelization is about 85%
- The overhead for MPI is higher than OpenMP.
- OpenMP can scale upto 64 or at most 128 processors. But this is extremely costly. A 32 core IBM p590 (2.2GHz PPC) costs > Rs. 2 crores.

- Memory is an issue for these kinds of machines. Again the limit is either 64 GB or at most 128 GB. Wilson Dirac operator with Wilson gauge action on a  $16^4$  lattice takes up close to 300 MB of memory. A  $64^4$  lattice will require  $256 \times 0.3 = 76.8$  GB of memory.
- With OpenMP one has to be careful about synchronization. A variable may suffer unwanted updates by threads. Use PRIVATE & SHARED declarations to avoid such problems.
- With OpenMP coding is easier. Unlike MPI where parallelization has to be done globally, here parallelization can be done incrementally loop by loop.

- Main advantage of MPI is that it can scale to thousands of processors. So it does not suffer from the Memory constraints.
- **If memory is not a problem and you have enough independent runs to do which uses up all your independent nodes, then use OpenMP. If not you will have to use MPI.**

Thank you