# Parallel Programming for Physicists

François Gelis
Grégoire Misguich

IPhT, June 7, 14, 21 & 28, 2019

Institut de
Physique Théorique
CEA/DRF Saclay

# Material for the course (code, Makefiles)

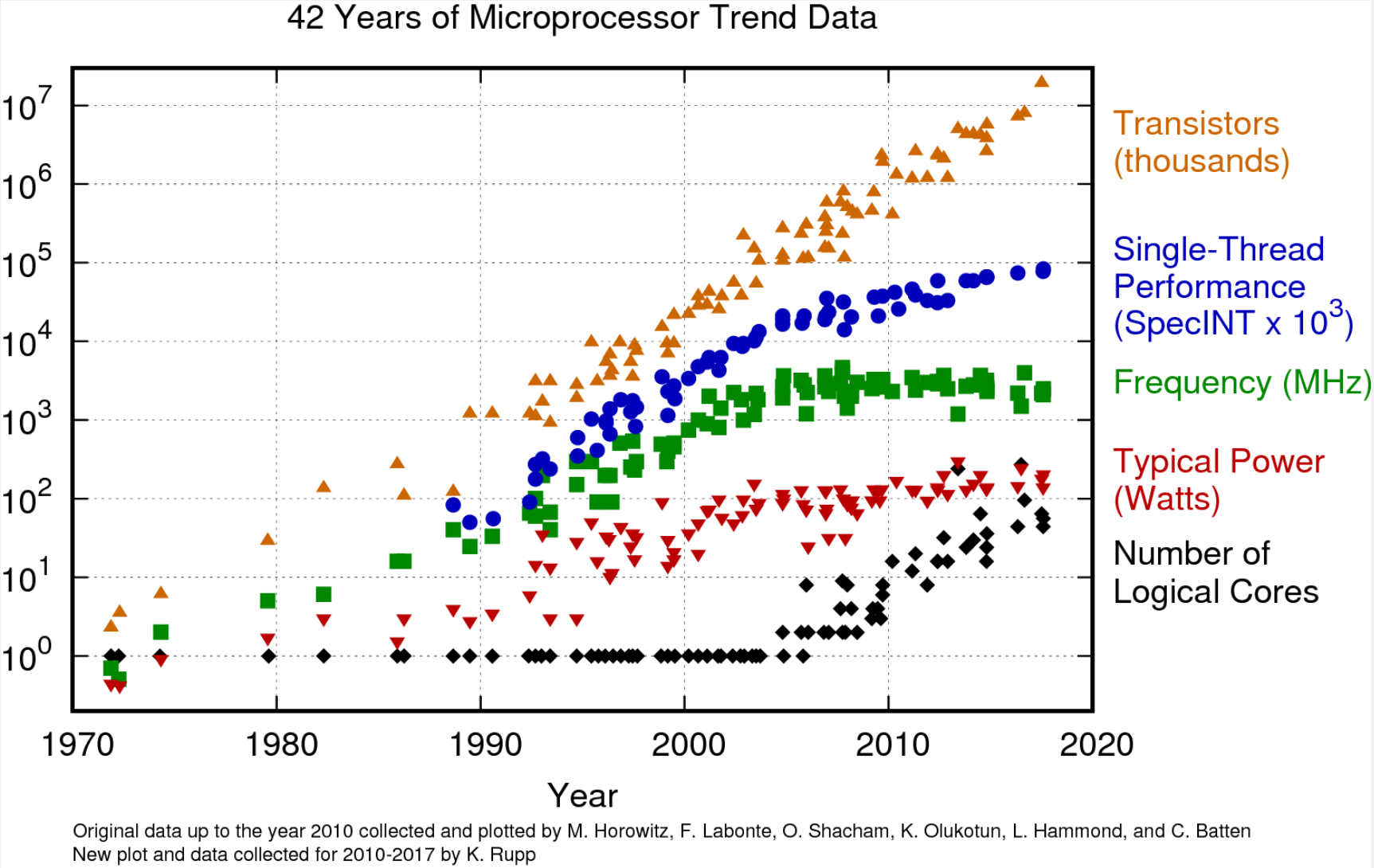
https://www.ipht.fr/Pisp/gregoire.misguich/pp.php

- uses a recent gcc/g++ compiler suite
- an example uses the fftw3 library
- for latest course: need openmpi

# Outline

- Hardware Considerations
- Parallellization in Existing Software
- Shared Memory: OpenMP
- Distributed Memory: MPI
- Hybrid parallelization: OpenMPI + MPI

# Hardware Considerations


42 Years of Microprocessor Trend Data
$10^7$
$10^6$
$10^5$
$10^4$
$10^3$
$10^2$
$10^1$
$10^0$
1970
1980
1990
2000
2010
2020
Year
Transistors (thousands)
Single-Thread Performance (SpecINT x $10^3$)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

# My Laptop [DELL XPS 13 (2015)]

- CPU: 2.4 to 3.0 GHz, 2 cores
- Peak performance: 46 GFLOP/sec (double precision)

  *(this would have placed my laptop on the top500 list of the world's most powerful computers, circa 1999...)*

# Let's do the math...

- At 3.0 GHz, 46 GFLOP/sec is equivalent to 15.3 FLOP/cycle

  i.e., 7.7 FLOP/cycle/core

  $\Longrightarrow$ there is a high amount of *instruction-level parallelism*
  In this case: each core uses AVX instructions to perform 4 double precision OPs at once, and has two Floating Point Units

$$3.0 \text{ GHz} \times (2 \text{ cores}) \times (2 \text{ FPUs}) \times 4(\text{AVX}) = 48 \text{ GFLOP/sec}$$

- What bandwidth do we need for this?

  To simplify, assume 1 FLOP = 2 reads + 1 write

$$\begin{aligned} 48 \text{ GFLOP/sec} =& 48 \times 8 \text{ (Bytes in a double)} \\ & \times 3 \text{ (2 reads + 1 write)} \qquad \text{GB/sec} \\ =& 1152 \text{ GB/sec} \end{aligned}$$

# My Laptop [cont..]

- Memory Bandwidth:
  - from RAM: 17-20 GB/sec
  - from L2 cache: 100-140 GB/sec
  - from L1 cache: 570-980 GB/sec
- Memory Latency:
  - from RAM to CPU: 300 cycles
  - from L2 cache to CPU: 12 cycles
  - from L1 cache to CPU: 4 cycles
  - *from L2 (core 1) to L2 (core 2): 90 cycles* $\longleftarrow$ *relevant for OpenMP*
- When the access pattern is regular, the CPU does some data prefetching, and these latencies are somewhat avoided
- With many cores, the needs for memory bandwidth become more severe

# Cache: use correct loop order in nested loops

- Good:

```
for (i=0;i<N;i++){
   for (j=0;j<N;j++){
     tmp += a[j+N*i];
   }
  }
```

- Not good:

```
for (j=0;j<N;j++){
   for (i=0;i<N;i++){
     tmp += a[j+N*i];
   }
  }
```

# Cache: sometimes, "loop tiling" helps

- Example with no really ideal ordering:

```
for (i=0;i<N;i++){
     for (j=0;j<N;j++){
   b[j+N*i] = a[i+N*j];
     }
    }
```

- Improvement: slice the $j$ loop in blocks of size $B$:

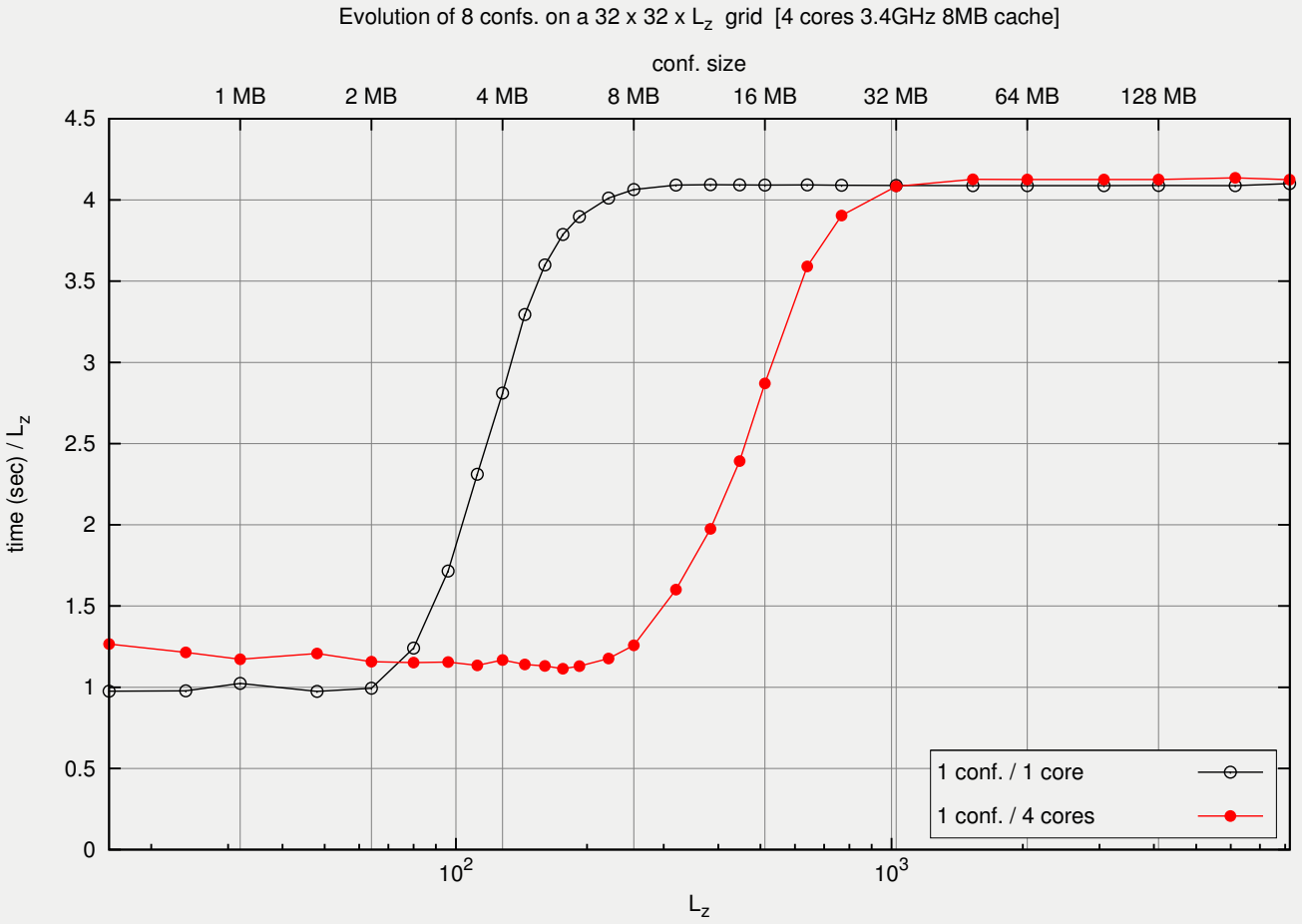
```
for (jb=0;jb<N;jb+=B){
    for (i=0;i<N;i++){
     for (j=0;j<B;j++){
   b[j+jb+N*i] = a[i+N*(j+jb)];
     }
    }
  }
```

# Cache: sometimes, "loop tiling" helps

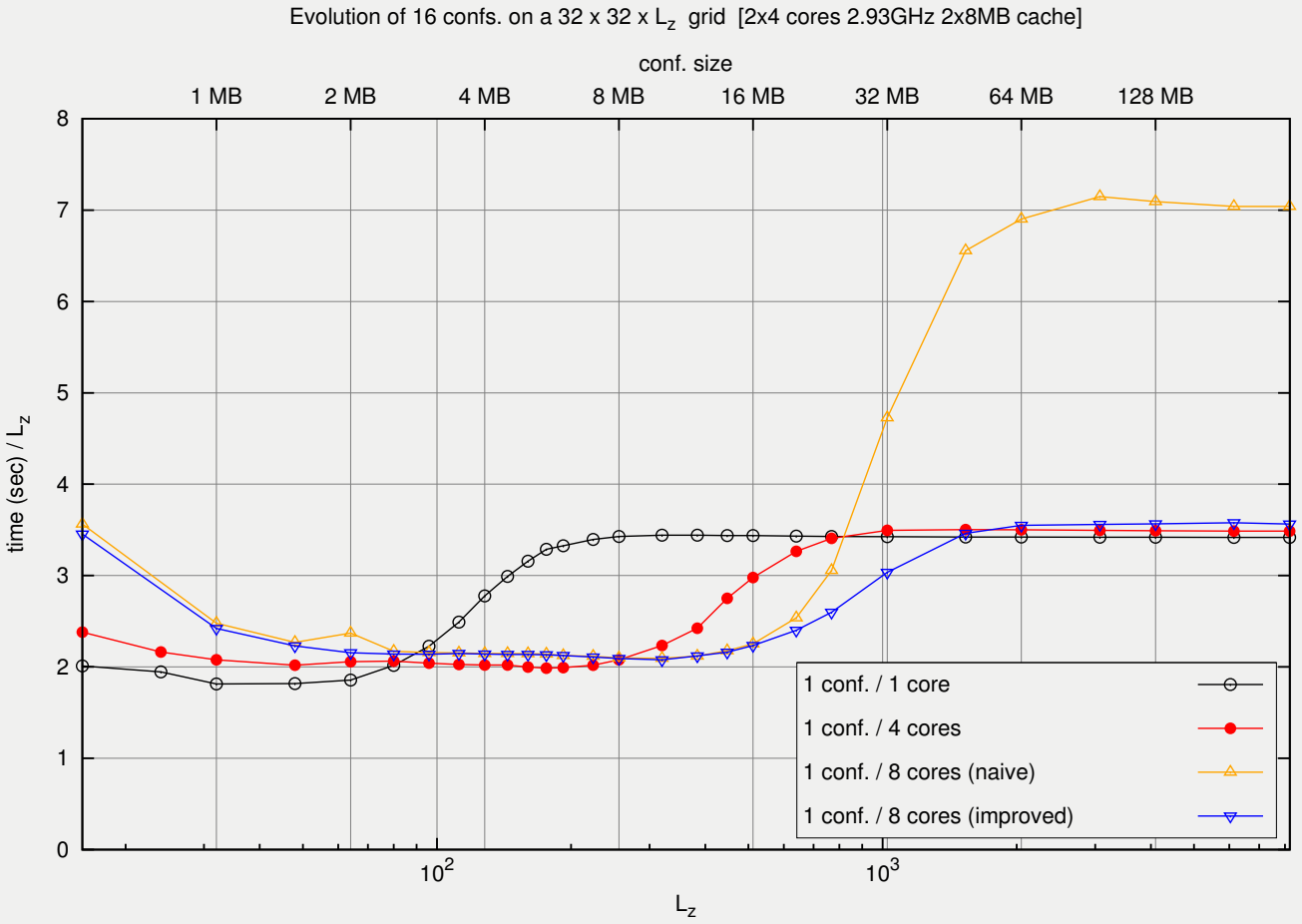
- How does it work?
  - This transformation makes the two innermost loops work on a contiguous range of size $N * B$ doubles
  - For $N = 2^{13}$, the optimal block-size is around $B = 2^4$ (empirical)
  - $2^{13} \times 2^4 \times 8$ (Bytes in a double) $=$ 1 MByte = order of L2 cache

- Note: the compiler can do this transformation automatically (but perhaps not choose the best block-size)

# Example: using parallelization to better use the cache



# Example: using parallelization to better use the cache



# Vector instructions

- SSE: 128 bit registers
  - 2 double precision
  - 4 simple precision
- AVX: 256 bit registers (most CPUs since 2015)
  - 4 double precision
  - 8 simple precision
- AVX512: 512 bit registers (only high-end CPUs)
- How to check:
  gcc -march=native -dM -E - < /dev/null | egrep "SSE|AVX" | sort

# Vectorization

- The compiler tries automatically with $-O3$, but not perfect
- Biggest obstacle: dependence among loop indices:

```
for (int j = 1; j < N; ++j){
 result[j] = 1.0/(1.0+result[j-1]);
}
```

- Less if the dependence has a range longer than the vector length

```
for (int j = 4; j < N; ++j){
 result[j] = 1.0/(1.0+result[j-4]);
}
```

# Summary

$$\text{PEAK Perf} = \underbrace{\text{Frequency}}_{\text{cannot change}} \times \underbrace{(\text{\# FPUs}) \times (\text{Width of vector inst.})}_{\text{well exploited only for appropriate code}} \times \underbrace{(\text{\# cores})}_{\text{try this?}}$$

Parallelization on shared memory:

- Very easy to implement via OpenMP, with tiny code modifications
- Should scale perfectly for independent tasks
- More tricky to split across cores tightly dependent tasks

# A multicore Bash script...

```
#!/bin/bash
maxjobs=8  ## adjust to the number of cores
runningjobs=0
for i in * ## modify to your needs
   do
      echo $i
      (sleep 1)& ## do something more useful here
      runningjobs=$(($runningjobs+1))
      if [ $runningjobs -ge "$maxjobs" ]
      then
        wait
        runningjobs=0
    echo " "
      fi
   done
```

# Parallelization in Existing Software

# Shared Memory: OpenMP

# Distributed Memory Parallelization

# Distributed memory "computers"

- Organization of a computer cluster:
  - one front-end node for compilation and administration tasks
  - many nodes for computations, not directly accessible
  - computation tasks submitted via a batch system
- The nodes are connected via a network
- Types of network connections:
  - 1 Gb/sec ethernet (125 MB/sec, latency around 1 ms)
  - 10 Gb/sec ethernet (bandwidth $\times 10$, similar latency)
  - Infiniband (latency around 1 microsec)
- Communications are slow (compared to shared memory)
- Communications must be handled explicitly in the program

# Example of task submission script (for PBS/Torque)

***Listing 1:*** *script.pbs*

```
#PBS —S /bin/bash
#PBS —N boltzmann
#PBS —e job.err
#PBS —o job.log
#PBS —m abe
#PBS —M francois.gelis@ipht.fr
#PBS —l nodes=32:ppn=16

module load openmpi/1.6.4
cd $PBS_O_WORKDIR

mpirun —npernode 1 ./my_program
```

- This example will start 32 copies of *my_program* (one per node)
- Then, do: *qsub script.pbs*
- Other commands: *qstat, qdel*
- This is sufficient to start independent tasks on several nodes
- Non-interactive: I/O to files only

# Message Passing Interface (MPI)

- MPI provides high level functions to exchange data between jobs on several nodes, that hide the network details
- Standardized since 1994: various implementations with compatible calls (openmpi, mpich, mpi/LAM, ...)
- Can be used from FORTRAN, C, C++, Python, ...
- Can be used in addition to OpenMP
- Most common functions:
  - Initialization: *MPI_Init, MPI_Comm_size, MPI_Comm_rank*
  - Cleanup: *MPI_Finalize*
  - Basic data exchange: *MPI_Send, MPI_Recv* (plus some variants)
  - Check communication status: *MPI_Wait, MPI_Test*
  - Collective communication: *MPI_Bcast, MPI_Gather, MPI_Scatter*
  - Synchronization: *MPI_Barrier*
- In C,C++: wrapper script around the compiler (mpicc) to load transparently the appropriate libraries/include files

# Toy example (each job prints its rank)

***Listing 2:*** *hello.c*

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

int main(int argc,char *argv[]){
  int rank,size;
  MPI_Init(&argc,&argv);                    // Initialize MPI
  MPI_Comm_size(MPI_COMM_WORLD,&size); // Number of CPUs
  MPI_Comm_rank(MPI_COMM_WORLD,&rank); // Rank in the pool of CPUs
  fprintf(stderr,"I am node %d in a pool of %d nodes\n",rank,size);
  MPI_Finalize();                           // Finalize just before exit
  return 0;
}
```

***Listing 3:*** *hostfile.sh*

```
localhost slots=16
```

- Compile with: *mpicc -o hello hello.c*
- Run locally with: mpirun -np 10 –hostfile hostfile.sh ./hello

# Communication domain

- *MPI_COMM_WORLD* is the set made of all the nodes
- One may define other “communication domains” (i.e., subsets)
- This is useful to restrict collective communications to a subgroup of the nodes

# Basic communication

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

int main(int argc,char *argv[]){
  int rank,size;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank>0) {
    srand(123+rank);      // Safe because each job has a fully private memory space
    int random = rand();
    fprintf(stderr,"node %d generated number %d\n",rank,random);
    MPI_Send(&random,1,MPI_INT,0,0,MPI_COMM_WORLD); // Send to rank=0
  } else {
    int count = 1, tmp;
    while (count<size) {
      MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,NULL); // Receive
      fprintf(stderr,"Node 0 received number %d\n",tmp);
      count++;
    }}

  MPI_Finalize();
  return 0;
}
```

- Note: the receiver does not know which node sent the data

# Basic communication, keeping track of sender

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

int main(int argc,char *argv[]){
  int rank, size;
  MPI_Status mpi_status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank>0) {
    srand(123+rank);
    int random = rand();
    fprintf(stderr,"node %d generated number %d\n",rank,random);
    MPI_Send(&random,1,MPI_INT,0,0,MPI_COMM_WORLD);
  } else {
    int count = 1, tmp, sender;
    while (count<size) {
      MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
      sender = mpi_status.MPI_SOURCE;  // Use status variable to discover who is the sender
      fprintf(stderr,"Node 0 received number %d from node %d\n",tmp,sender);
      count++;
    }}

  MPI_Finalize();
  return 0;
}
```

# MPI TYPES

| MPI Data type | C Data Type |
|---|---|
| MPI_CHAR | char |
| MPI_SHORT | short int |
| MPI_INT | int |
| MPI_LONG | long int |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED_INT | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |

- MPI provides functions for defining additional types (e.g., a block out of a block representation of a matrix)

# Source, Destination and Tag

- MPI_Recv(Buf,Size,Type,**Source**,**Tag**,MPI_COMM_WORLD,&status)
- MPI_Send(Buf,Size,Type,**Dest**,**Tag**,MPI_COMM_WORLD)

- **Source** : specifies the origin
  Source=*MPI_ANY_SOURCE* : unspecified origin
- **Dest** : specifies the destination

- **Tag** : integer that labels a specific communication channel
  Must match between sender and receiver, unless the receiver uses *MPI_ANY_TAG*

- When using MPI_ANY_SOURCE and/or MPI_ANY_TAG, Source and Tag may be obtained afterwards from the *status* variable, as
  - status.MPI_SOURCE
  - status.MPI_TAG

# Beware: MPI_Send, MPI_Recv are blocking functions

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

// Each node passes to the next a value received from the previous one
// Will lock, because of a "chicken and egg problem"

int main(int argc,char *argv[]){
  int rank, size, tmp;
  MPI_Status mpi_status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // Each job posts a Receive, then a Send
  MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
  fprintf(stderr,"node %d received token %d from node %d\n",rank,tmp,mpi_status.MPI_SOURCE);
  if (rank==0) tmp=-1;
  MPI_Send(&tmp,1,MPI_INT,(rank+1)%size,0,MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

# Avoiding deadlocks by ordering Sends/Receives

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

int main(int argc,char *argv[]){
  int rank, size, tmp;
  MPI_Status mpi_status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank>0) { // Nodes>0: post a (blocking) Receive
    MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
    fprintf(stderr,"node %d received token %d from node %d\n",rank,tmp,mpi_status.MPI_SOURCE);
  }

  if (rank==0) tmp=-1; // Everybody: Send "tmp" to next node
  MPI_Send(&tmp,1,MPI_INT,(rank+1)%size,0,MPI_COMM_WORLD);

  if (rank==0) { // Node 0: final Reveive
    MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&mpi_status);
    fprintf(stderr,"node %d received token %d from node %d\n",rank,tmp,mpi_status.MPI_SOURCE);
  }

  MPI_Finalize();
  return 0;
}
```

# Variants of MPI_Send

- *MPI_Send* : blocks until the array containing the data to be sent out can be reused safely (does not imply that the data has already reached its destination, since it could be buffered)
- This function has several variants
  - *MPI_Ssend* : synchronous send; blocks until a matching receive has been posted. Avoids buffering, but higher risk of locking
  - *MPI_Isend* : immediate send; non-blocking, but one cannot reuse the array safely. Should be used in conjunction with *MPI_Wait* or *MPI_Test* to check when it is safe to reuse the array
  - *MPI_Bsend* : buffered send; returns immediately, and the array can be reused immediately. Degrades performance

# Variants of MPI_Recv

- *MPI_Recv* : blocks until the array in which the data to be received arrives is ready to be used
- This function has one variant
  - *MPI_Irecv* : immediate receive; non-blocking, but one cannot use the array yet. Should be used in conjunction with *MPI_Wait* or *MPI_Test* to check when it is safe to use the array
- Pros/cons of non-blocking communication:
  - allows to perform other communications or computations instead of waiting
  - can reduce latency by posting receives early
  - less chances of locking
  - BUT: one needs to test whether the operation has completed

# Usage of MPI_Wait and MPI_Test

- *MPI_Wait* : blocking function

```
MPI_Request request;
MPI_Status status;
//
MPI_Irecv(Buf,Size,Type,Source,Tag,MPI_COMM_WORLD,&request);
//
// do something useful (that does not use Buf)
// now, assume that we need Buf
//
MPI_Wait(&request,&status); // blocks until receive is completed
```

- *MPI_Test* : non-blocking polling function

```
MPI_Request request;
MPI_Status status;
int done = FALSE;
//
MPI_Irecv(Buf,Size,Type,Source,Tag,MPI_COMM_WORLD,&request);
//
while (done==FALSE) {
  // do something useful (that does not use Buf)
  MPI_Test(&request,&done,&status); // returns immediately
 }
```

- *MPI_Waitall* : waits for a set of communications

# Simple example (with multiple pending requests)

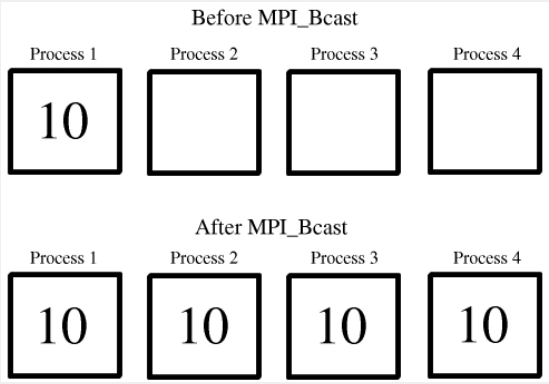
```
#include <stdio.h>
#include <mpi.h>

int main (int argc, char *argv[]){
  int size, rank, next, prev, buf[2], tag1=1, tag2=2;
  MPI_Request requests[4];
  MPI_Status status[4];
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  prev = (rank-1+size)%size;
  next = (rank+1)%size;
  // Initiate a bunch of communications (they all return immediately)
  MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &requests[0]);
  MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &requests[1]);
  MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &requests[2]);
  MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &requests[3]);
  //
  // do something else, then wait for completion of all communications
  //
  MPI_Waitall(4, requests, status);
  printf("Task %d communicated with tasks %d & %d\n",rank,prev,next);

  MPI_Finalize();
}
```

# Avoiding deadlocks with asynchronous communication

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

int main(int argc,char *argv[]){
  int rank, size, tmp, final;
  MPI_Status status;
  MPI_Request request;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // Node 0 posts an early non-blocking receive, others post a blocking one
  if (rank==0) {MPI_Irecv(&final,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&request);tmp = -1;}
  else {MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);}

  // Everybody: pass value to the next node
  MPI_Send(&tmp,1,MPI_INT,(rank+1)%size,0,MPI_COMM_WORLD);

  if (rank==0) MPI_Wait(&request,&status); // Node 0 must check that Receive has completed
  fprintf(stderr,"node %d received token %d from node %d\n",rank,tmp,status.MPI_SOURCE);

  MPI_Finalize();
  return 0;
}
```

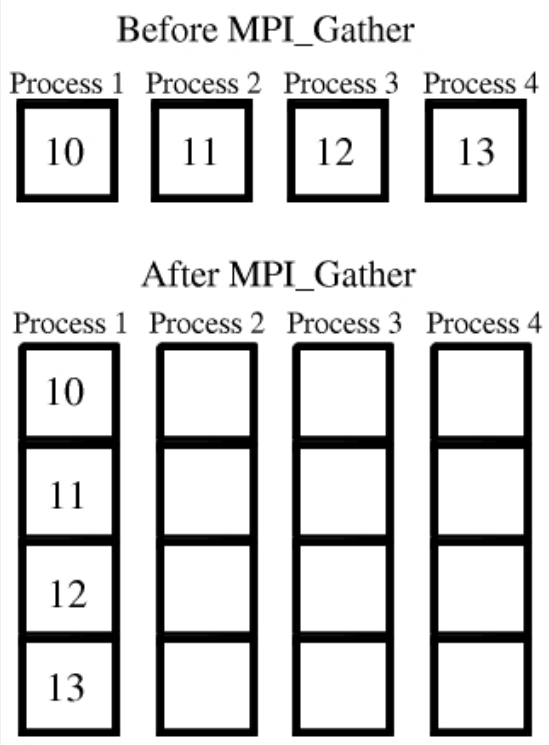
# Collective communication: Broadcast

- MPI_Bcast(Buf,N,Type,0,MPI_COMM_WORLD)



# Collective communication: Gather

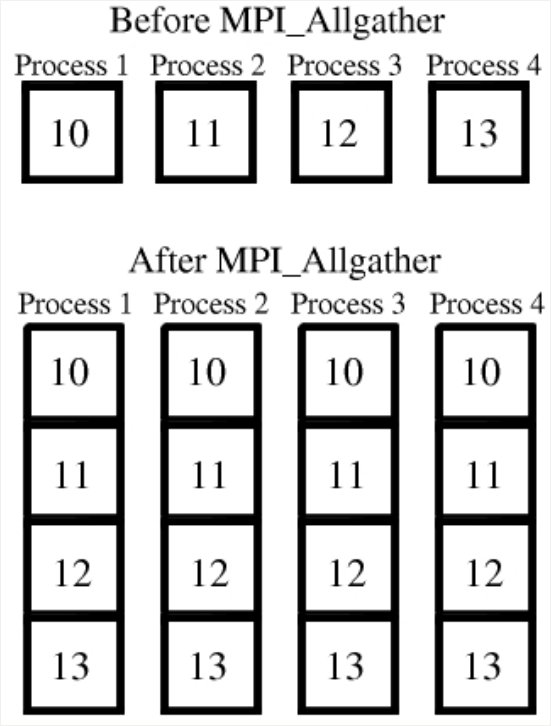
- MPI_Gather(Source,N,Type,Dest,N,Type,0,MPI_COMM_WORLD)




- Variant: MPI_Gatherv: gather variable size chunks of data, and place them at variable offsets in the Dest array

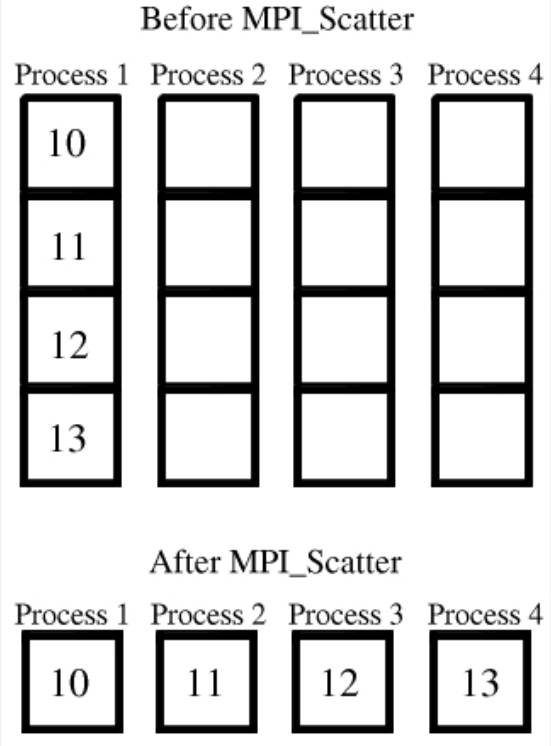
# Collective communication: AllGather

- MPI_Allgather(Source,N,Type,Dest,N,Type,MPI_COMM_WORLD)



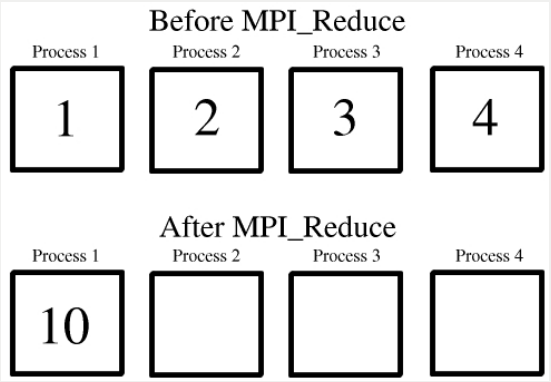
# Collective communication: Scatter

- MPI_Scatter(Source,N,Type,Dest,N,Type,0,MPI_COMM_WORLD)



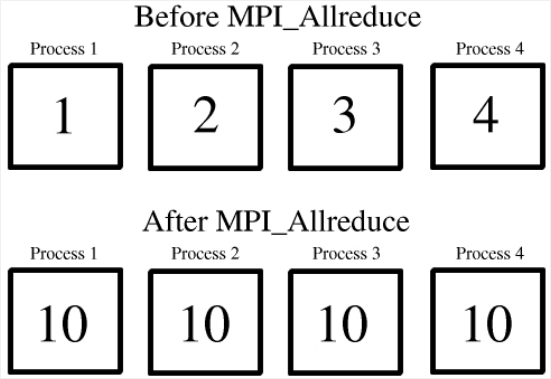
# Global reductions: Reduce

- MPI_Reduce(Source,Dest,N,Type,Op,o,MPI_COMM_WORLD)
- Op = MPI_SUM, MPI_PROD, etc...



# Global reductions: AllReduce

- MPI_Allreduce(Source,Dest,N,Type,Op,MPI_COMM_WORLD)



# Hybrid: MPI+OpenMP

- Superimposing MPI and OpenMP parallelization poses no problem
- In general, each MPI job should fork in a number of threads equal to the number of physical cores on one node
- Various hybrid scenarios are possible

# Easiest: independent computations on each node

- Each MPI job performs a (lengthy) computation, independently of the other jobs
- These computations are parallelized using OpenMP
- Output is sent back to MASTER node for final processing (e.g., averaging, saving in a file, etc...). For this, the MASTER node forks a special thread that collects this output
- Input data:
  - Monte-Carlo: input is a random configuration, generated locally on each node (make sure each MPI job uses its own RNG seed)
  - Alternative: the MASTER node sends tasks taken from a list to each node as soon as a result returns

# Threads on MASTER node

- Receiving/processing computed results is a very light task compared to the computations $\Rightarrow$ the MASTER node can also perform computations without significant penalty

- Call *omp_set_nested(1)* on all nodes

- Create two parallel sections:
  - First section: executed only by MASTER (empty on other nodes)
    Post MPI_Recv to receive results, post-process results, MPI_Send to send tasks, etc...
  - Second section: executed by everybody
    performs actual computations $\Rightarrow$ further fork in $\mathrm{N}_{\mathrm{cores}}$ threads

# Harder: Two-level slicing of the computation domain

- Large array to be processed (e.g., evolved in time)
- Divide the array in $N_{\text{nodes}} \times N_{\text{cores}}$ slices, and assign a slice to each thread

- Case 1: evolution is "almost" local, i.e. depends at most on a few neighboring sites (e.g., discretization of a Laplacian)

  At the beginning of each timestep, each thread must be given a copy of the layers just before and just after its slice

- Case 2: evolution is completely non-local (update of a point $i$ depends on all other points)

  2.a. Array is small enough: each node can have its own copy of the full array (it must be refreshed at the beginning of each timestep)

  2.b. Array is too large: communications will probably make parallelization very inefficient

# Case study: deterministic Boltzmann solver

# Boltzmann Equation

- Two-body elastic interactions
- Spatially homogenous
- Isotropic particle distribution
- Scattering amplitude may be momentum dependent

$$\partial_t f_1 = \frac{1}{E_1} \int_{\mathbf{p}_{2,3,4}} \left|M(1,2,3,4)\right|^2 \left[f_3 f_4 (1+f_1)(1+f_2) - f_1 f_2 (1+f_3)(1+f_4)\right]$$

- Collision integral reduces to 4-dim integral thanks to momentum conservation and isotropy

# Sketch of the Algorithm

- Discretize $|\mathbf{p}| \quad \rightarrow \quad p_i \quad (0 \leq i < N)$

- At each time-step:

  1. For each $i$, compute $\Delta f_i$ (given by a 4-dim integral $\Rightarrow$ slow)
  2. Check if $f_i + \Delta f_i \geq 0$ (for all $i$)
     If FALSE, re-run the previous loop with a smaller timestep
  3. Do $f_i + \Delta f_i \rightarrow f_i$ and return to step 1

- Note: cost of computing $\Delta f_i$ not uniform (50% variation)
  Thus, we expect that the parallelization of the "big loop" will not be perfectly efficient (the computation time will align to that of the slowest bins)

# Sequential version (sketch of the relevant bit of code)

```
// Core of the function that evolves f[i]

double *df = (double *)malloc(N*sizeof(double));
for (i=0;i<N;i++) df[i] = dt*C(i,f); // depends on f[k] with k!=i; computation of C(i,f) very slow

status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;
if (status==1) return 1;

for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);
return 0;
```

- Large fraction of the time spent in the first loop
- For a reasonable grid size: $30$ minutes/timestep
- Typical evolution: $2000$ timesteps ($1000$ hours, or $42$ days...)

# OpenMP version

```
// Core of the function that evolves f[i]

double *df = (double *)malloc(N*sizeof(double));
#pragma omp parallel for num_threads(NT) schedule(dynamic) // <——— ONLY ADDITION
for (i=0;i<N;i++) df[i] = dt*C(i,f);

status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;
if (status==1) return 1;

for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);
return 0;
```

- Define *NT* (number of threads) in the scope of the function
- Add *-fopenmp* to compiler options
- Better use of threads if *NT* divides *N*

# MPI+OpenMP version

```
// Core of the function that evolves f[i]

int n = N/size, imin = rank*n, imax = (rank+1)*n;                    // <—— workload of each mode
double *df=NULL, *local_df = (double *)malloc(n*sizeof(double)); // <—— Node—local storage
if (rank==0) df = (double *)malloc(N*sizeof(double));                // <—— MASTER needs a buffer for df[i]

#pragma omp parallel for num_threads(NT) schedule(dynamic)
for (i=imin;i<imax;i++) local_df[i—imin] = dt*C(i,f);

MPI_Gather(local_df,n,MPI_DOUBLE,df,n,MPI_DOUBLE,0,MPI_COMM_WORLD); // <—— Gather partial results on MASTER
if (rank==0) {status = 0; for (i=0;i<N;i++) if (f[i]+df[i]<0) status = 1;}
MPI_Bcast(&status,1,MPI_INT,0,MPI_COMM_WORLD);                       // <—— Broadcast test result
if (status==1) return 1;

if {rank==0} {for (i=0;i<N;i++) f[i] = f[i]+df[i]; free(df);}
MPI_Bcast(f,N,MPI_DOUBLE,0,MPI_COMM_WORLD);                          // <—— Broadcast updated f[i]
free(local_df);
return 0;
```

- *size* and *rank* needed in the update function
- Add *#include <mpi.h>* in the header
- Output to files done by MASTER node only

# Timings

- Running time:
  - Sequential: 1830 seconds/timestep
  - OpenMP (16 cores): 154 seconds/timestep
    ($\times 12$ speedup, $75\%$ efficiency)
  - MPI+OpenMP (32 nodes $\times$ 16 cores): 6 seconds/timestep
    ($\times 306$ speedup, $60\%$ efficiency)
    Computation time reduced from 1000 hours to 3 hours 16 minutes

- Coding time:
  - Sequential: one week (about 600 lines of code)
  - +OpenMP: +one minute (+1 line)
  - +MPI: +one hour (+10 lines)

# Thank You