Roberto De Pietri

Esperienze openACC e openMP a Parma

Problems porting application
to many-core and GPU systems

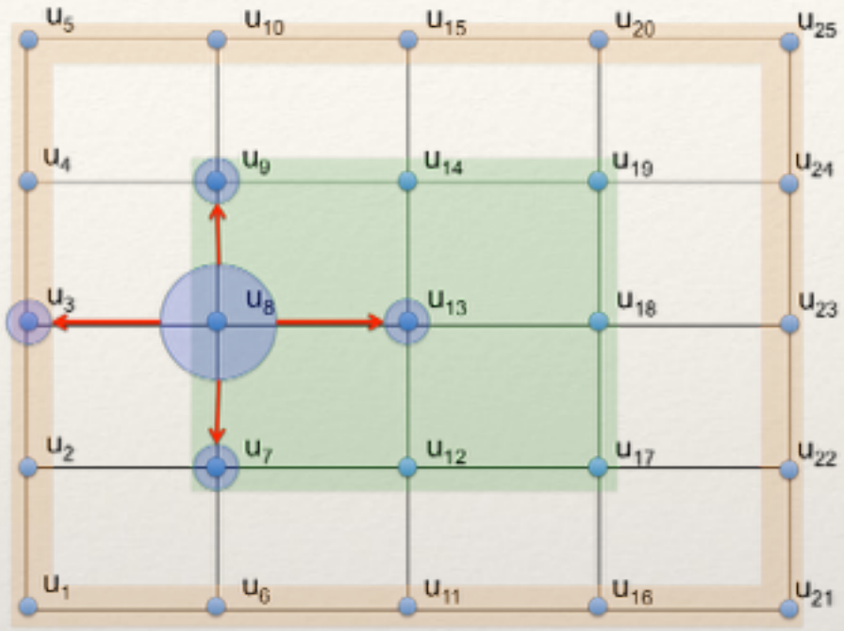
R De Pietri, R. Alfieri,
M. Borelli, A.Feo, P. Leoni

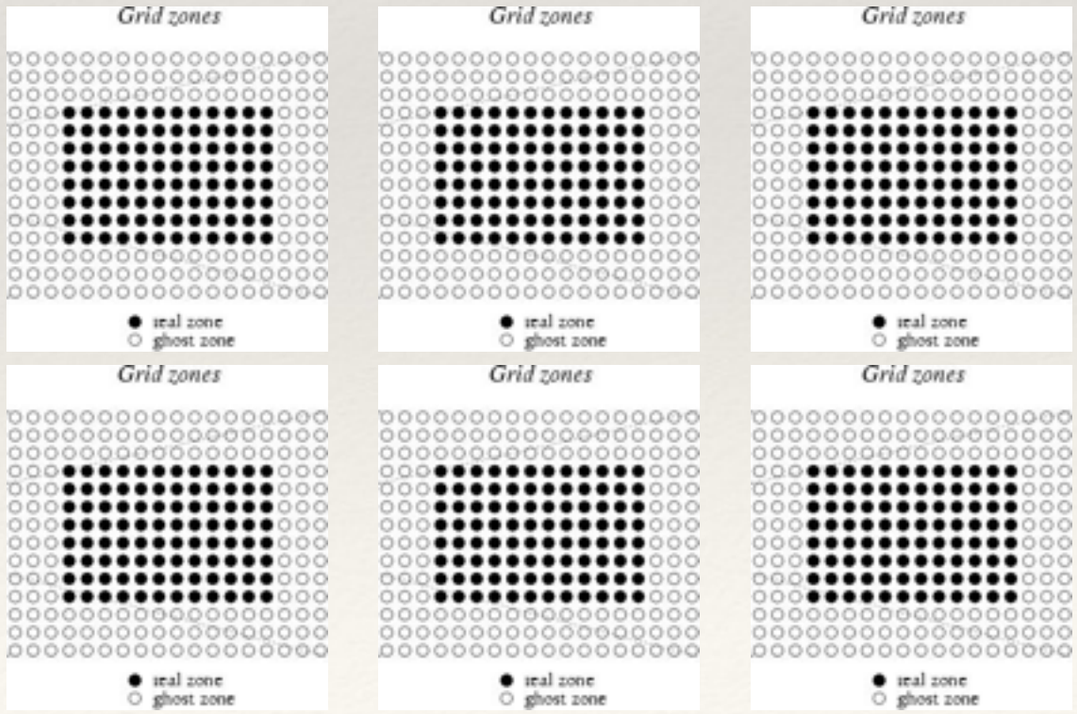
Summary - purpose of the research

- To check if it is easy to create a simulation program that can be easily ported to different architecture
- How to write a program that run in many-core, GPU, standard CPU and communicating through the MPI-library
- How well it performs …..
- Which ones are the best strategy to tune it …
- **MINIMAL MODIFICATION TO STANDARD CODE**

Problem type we are looking for

- **Time evolution of Partial Differential Equations on a cartesian grid**
- **The evolution use differential operation with a fixed stencil size**
- **The variables that correspond to the "physical space" need to be partitioned to be executed on different physical computing nodes that comunicate which each other**



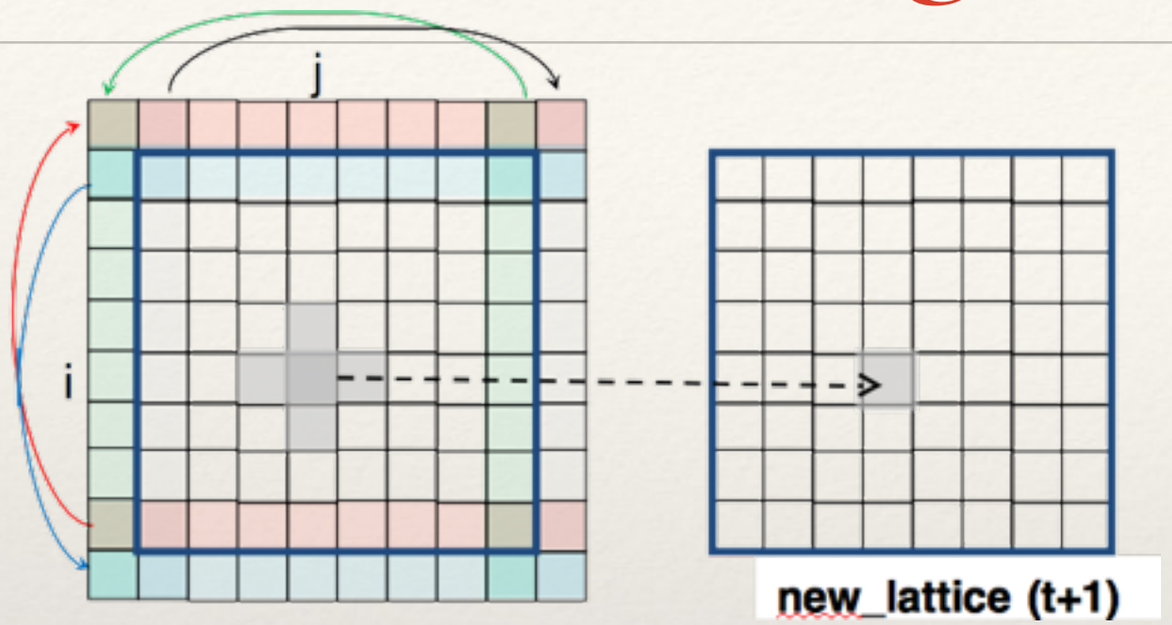


The conditions we would like to match

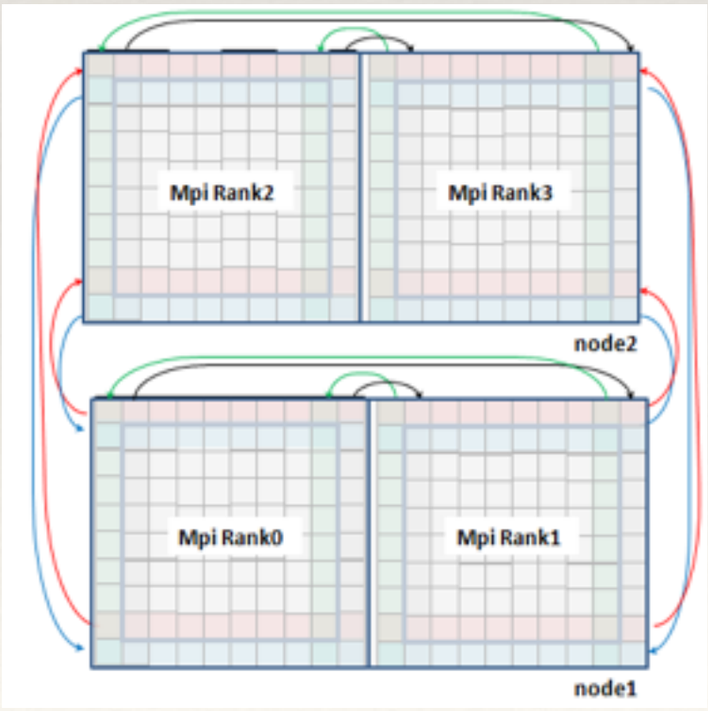
- **3d grids**
- **variable number** of ghost size
- **different number** of floating point **variables** associated **to each grid point**
- **different number** of floating points operation **needed for the update** of the variables associated to each grid-point.
- **When a single physical node do not have all the memory needed to store the configuration.**

- A grid size of 1000x1000x1000 with 10 **variables for each grid point** and 3 time levels requires 300GBytes of memory
- IF the update of a grid variable for each point requires 50 Flops for a time step, the total **number** of floating point operations that would be required is 1 TFlops …. Usually we need to do, at least, 10000-20000 time steps….
- **clock frequency is not going up** and indeed the # of floating point operations **for functional units** will not increase.
- **We need to partition the computation !**

The “game of life”



- Just a simple update rule
- + a local computation that can get easily vectorized by compiler
  Nc -> sum = a[i]*b[i]
- Total computations (26GFlop/5.28 TFlop(Nc=1000))



Our main update routine

- Total computations
- # update=10
- max grid 17000x17000
- vector computation = ncomp*2 Flops
- rule = 10 Flops
- ncomp=0 : 26GFlop
- ncomp=1000 : 5.28TFlop

```
 // Compute Internals
#pragma acc parallel present(grid[nrows+2][ncols+2],\
       next_grid[nrows+2][ncols+2],sum,A[0:ncomp],B[0:ncomp]) \
        async(2) num_gans(100) vector_length(16)
{
  #pragma acc loop gang independent
  #pragma omp for private(i,j,k,neighbors,sum)
  for (i=rmin_int; i<=rmax_int; i++) {
    #pragma acc loop worker independent
    for (j=cmin_int; j<=cmax_int; j++) {
      #pragma ivdep
      #pragma vector aligned
      #pragma acc loop vector independent reduction(+: sum) \
              private(sum)
      for (k=0; k < ncomp; k++)  sum += A[k] + B[k];
      // ffff
      neighbors = grid[i+1][j+1] + grid[i+1][j] + grid[i+1][j-1]
       + grid[i][j+1] + grid[i][j-1] + grid[i-1][j+1]+grid[i-1][j]
       + grid[i-1][j-1];
      if ( ( neighbors > 3.0 ) || ( neighbors < 2.0 ) )
        next_grid[i][j] = 0.0;
      else if ( neighbors == 3.0 )
        next_grid[i][j] = 1.0;
      else
        next_grid[i][j] =  grid[i][j];
    }
  }
}  // end Compute Internale
```

The Nvidia KEPLER K20- GPU

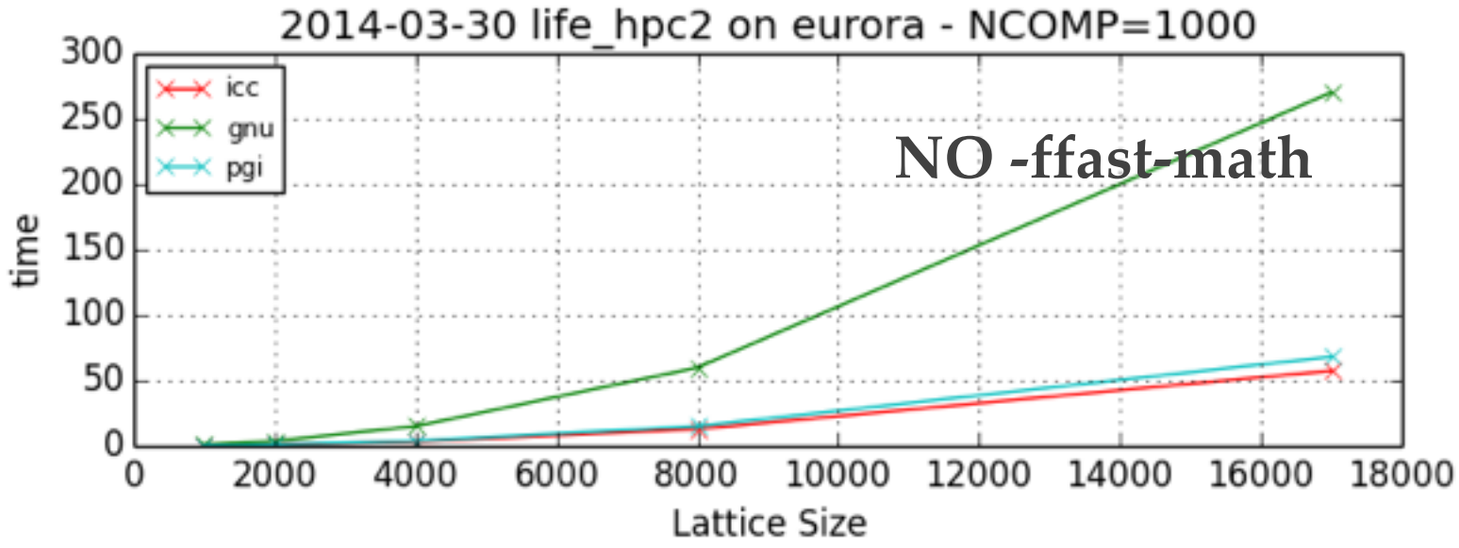
- GPU are much different from CPU and need a lot of effort to write a program that run efficiently on this architecture (CUDA/OpenCL?)

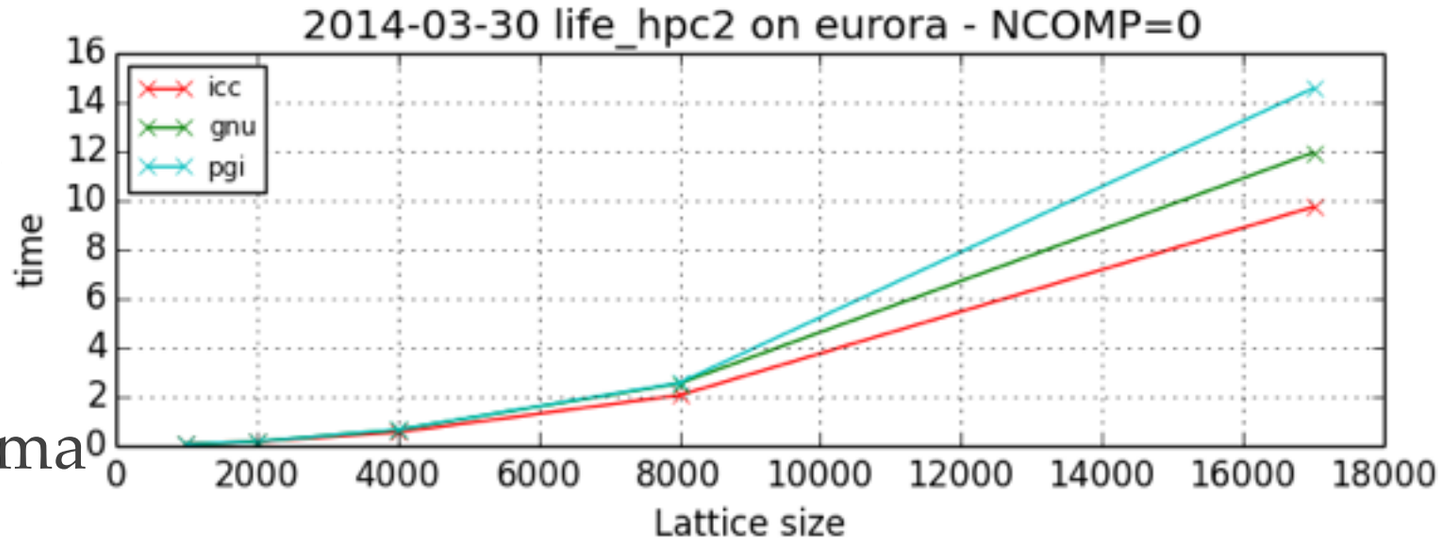
OpenACC vs OpenMP

- OpenACC is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The standard is designed to simplify parallel programming of heterogeneous CPU/GPU systems.
- Like in OpenMP, the programmer can annotate C, C++ and Fortran source code to identify the areas that should be accelerated using PRAGMA compiler directives and additional functions.
- OpenMP (from 4.0) will be available not only on the CPU but also on GPU …. we need to check it out … (gcc-4.9)

OpenMP -different compiler

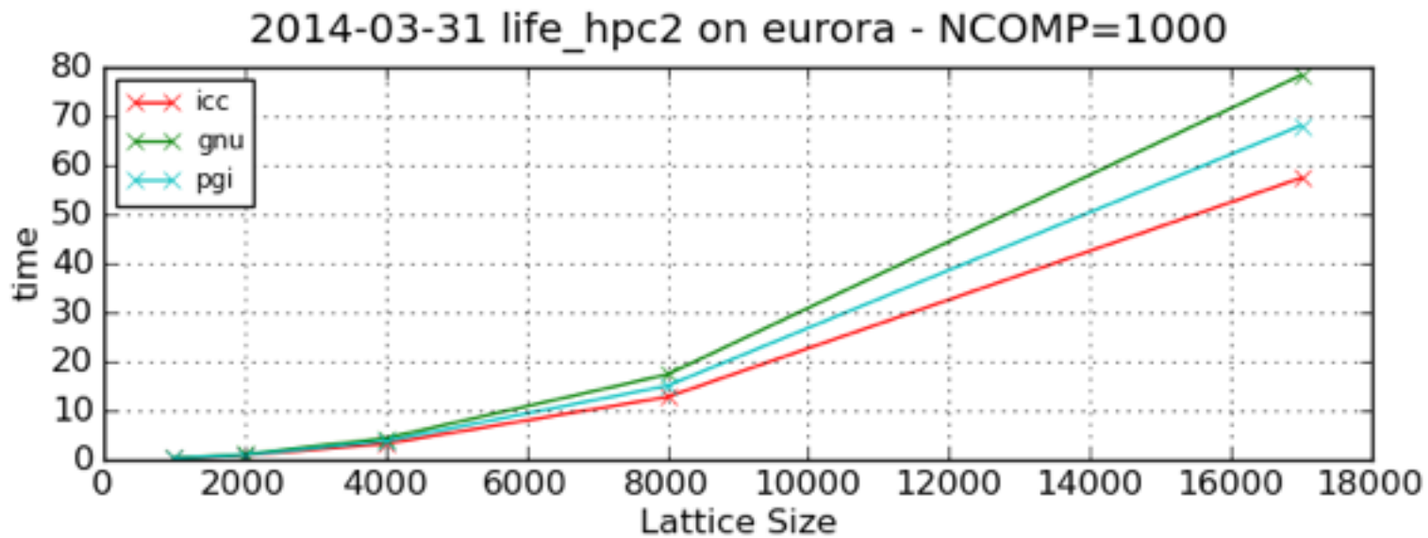
- Nthreads=16 eurora
- ICC:
  icc -xavx -fopenmp
  -vec-report2
- GNU:
  gcc -mavx -fopenmp
  -ftree-vectorize -ffast-map
  -ftree-vectorizer-verbose=1
- PGI:
  pgcc -tp=sandybridge-64
  -Mvect=simd:256 -mp=numa
  -fast -Minfo=vec,mp

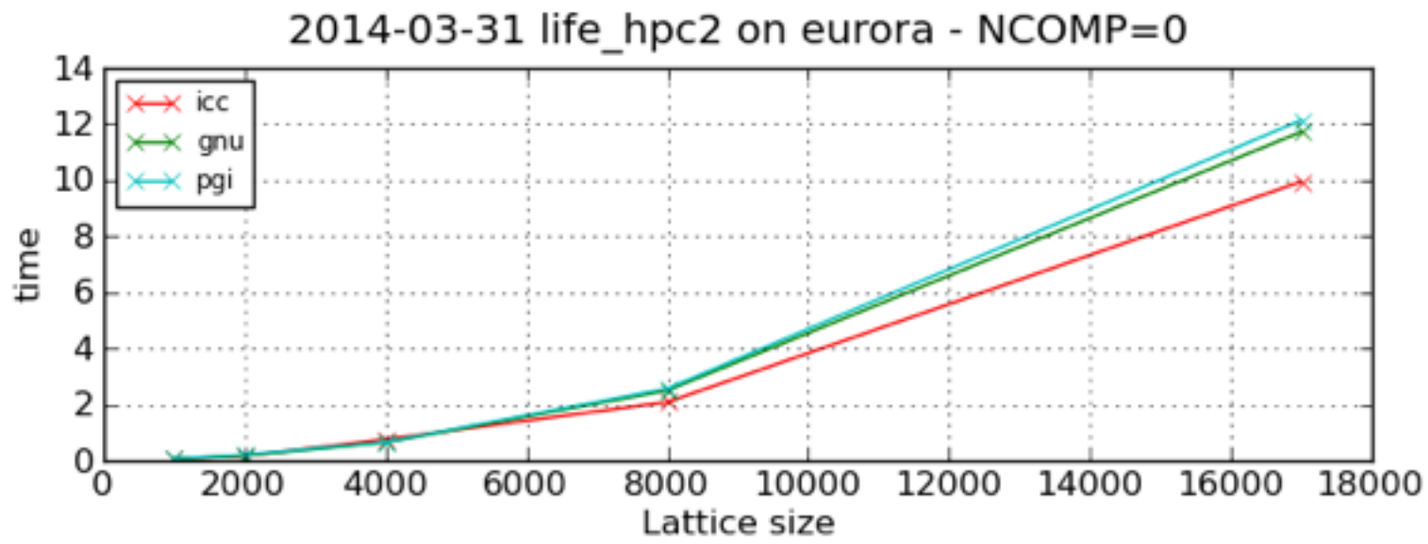




OpenMP -different compiler

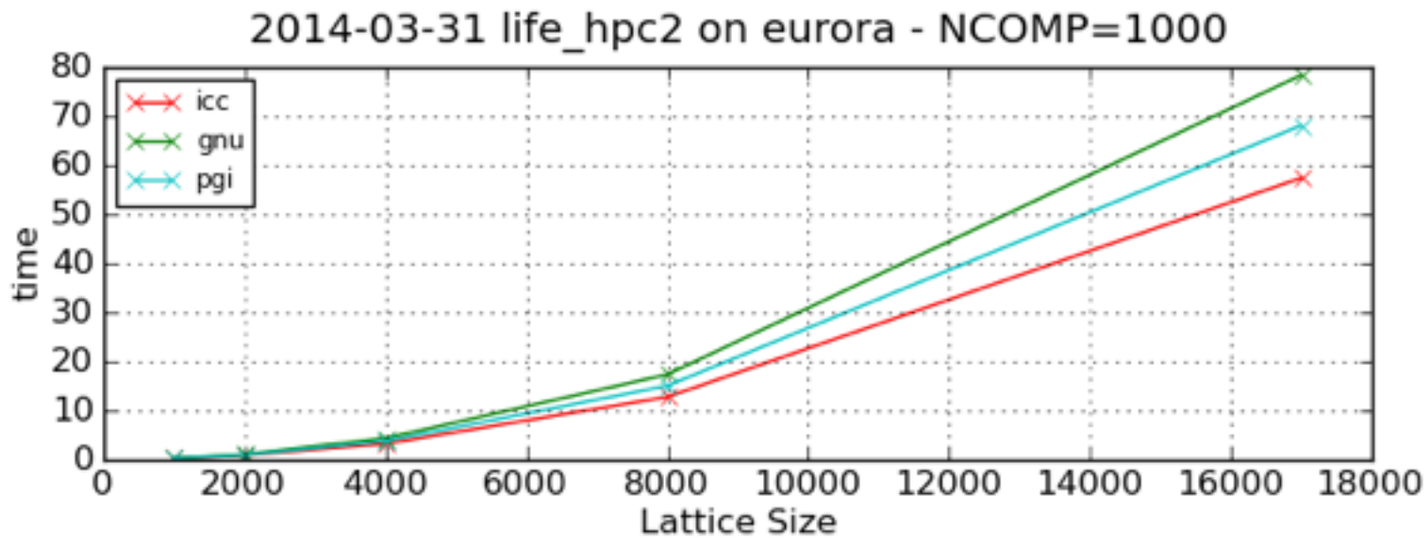
- Nthreads=16 eurora
- ICC:
  icc -xavx -fopenmp
  -vec-report2
- GNU:
  gcc -mavx -fopenmp
  -ftree-vectorize -ffast-ma
  -ftree-vectorizer-verbose
- PGI:
  pgcc -tp=sandybridge-64
  -Mvect=simd:256 -mp=
  -fast -Minfo=vec,mp

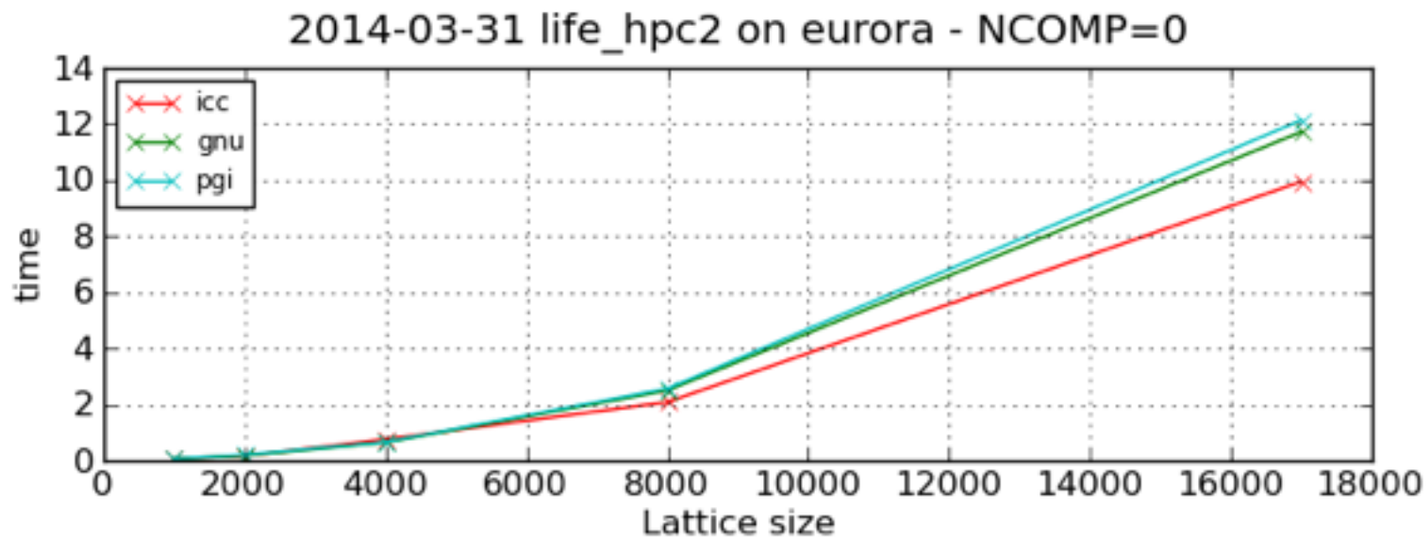




OpenMP -different compiler

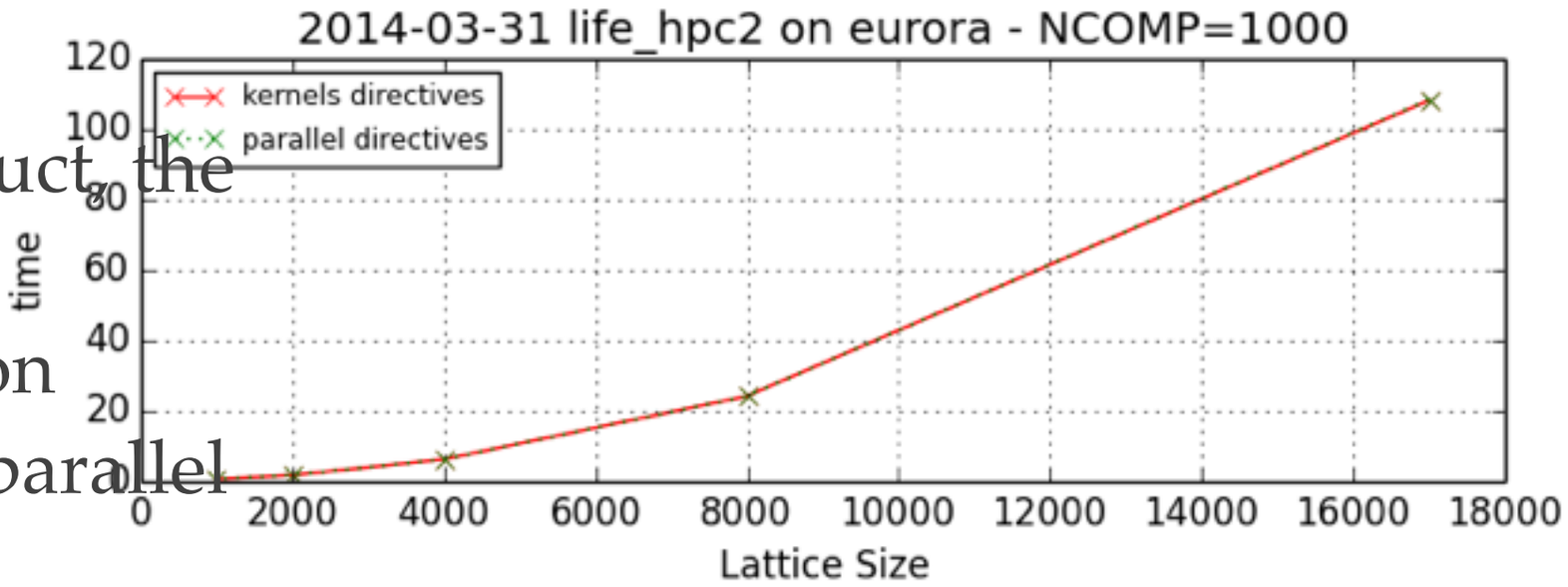
- Nthreads=16 eurora
- ICC:
  icc -xavx -fopenmp
  -vec-report2
- GNU:
  gcc -mavx -fopenmp
  -ftree-vectorize -ffast-ma
  -ftree-vectorizer-verbose
- PGI:
  pgcc -tp=sandybridge-64
  -Mvect=simd:256 -mp=
  -fast -Minfo=vec,mp



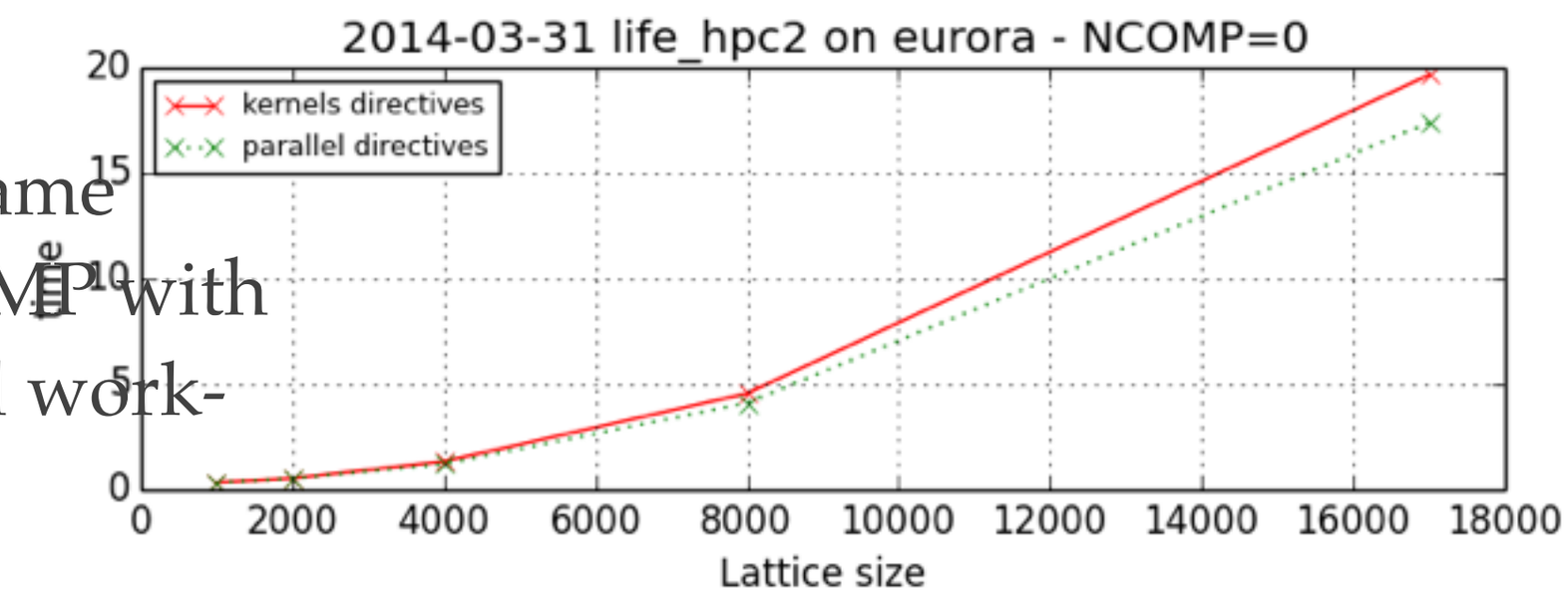


OpenACC – #kernel vs #parallel

- **#pragma acc kernels**
  Within a kernels construct, the compiler uses classical automatic parallelization technology to identify parallel loops.



- **#pragma acc parallel**
  This is effectively the same behavior as with OpenMP with the nowait clause on all work-sharing loops..

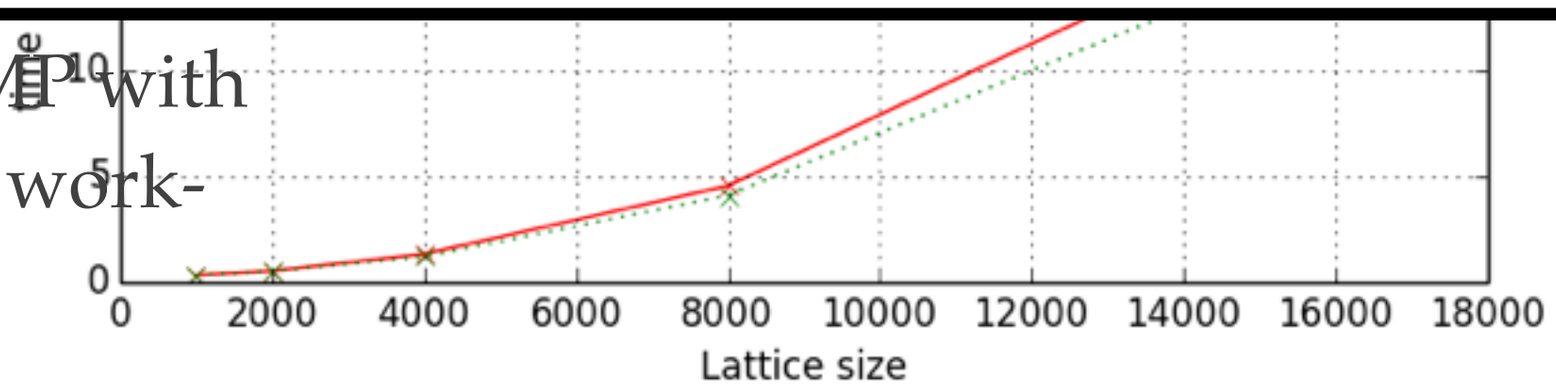


http://insidehpc.com/2012/08/30/michael-wolfe-on-openacc-kernels-and-parallel-constructs/
https://www.pgroup.com/lit/articles/insider/v4n2a1.htm

OpenA

- **#pragma acc ker**
  Within a k
  compiler uses cla
  automatic paralle
  technology to ide
  loops.
- **#pragma acc par**
  This is effectively
  behavior as with OpenMP with the nowait clause on all work-sharing loops..

```
// Compute Internals
#pragma acc kernels present(grid[nrows+2][ncols+2], \
            next_grid[nrows+2][ncols+2],sum,A[0:ncomp],B[0:ncomp]) async(2)
{
  #pragma acc loop gang(100) independent
  for (i=rmin_int; i<=rmax_int; i++) {  // righe
    #pragma acc loop workers independent
    for (j=cmin_int; j<=cmax_int; j++) {  // colonne
     #pragma vector aligned
      #pragma acc loop vector(16) reduction(+: sum) independent private(sum)
      for (k=0; k < ncomp; k++)  sum += A[k] + B[k]; // COMP
      // LIFE
      neighbors = grid[i+1][j+1] + grid[i+1][j] + grid[i+1][j-1]
                + grid[i][j+1] + grid[i][j-1] + grid[i-1][j+1]+grid[i-1][j]
                +grid[i-1][j-1];
      if ( ( neighbors > 3.0 ) || ( neighbors < 2.0 ) )
        next_grid[i][j] = 0.0;
      else if ( neighbors == 3.0 )
        next_grid[i][j] = 1.0;
      else
        next_grid[i][j] =  grid[i][j];
    }
  }
}  // end Compute Internals
```



http://insidehpc.com/2012/08/30/michael-wolfe-on-openacc-kernels-and-parallel-constructs/
https://www.pgroup.com/lit/articles/insider/v4n2a1.htm

OpenA

- **#pragma acc ker**
  Within a k
  compiler uses cla
  automatic parall
  technology to id
  loops.
- **#pragma acc par**
  This is effectivel
  behavior as with
  the nowait claus
  sharing loops..

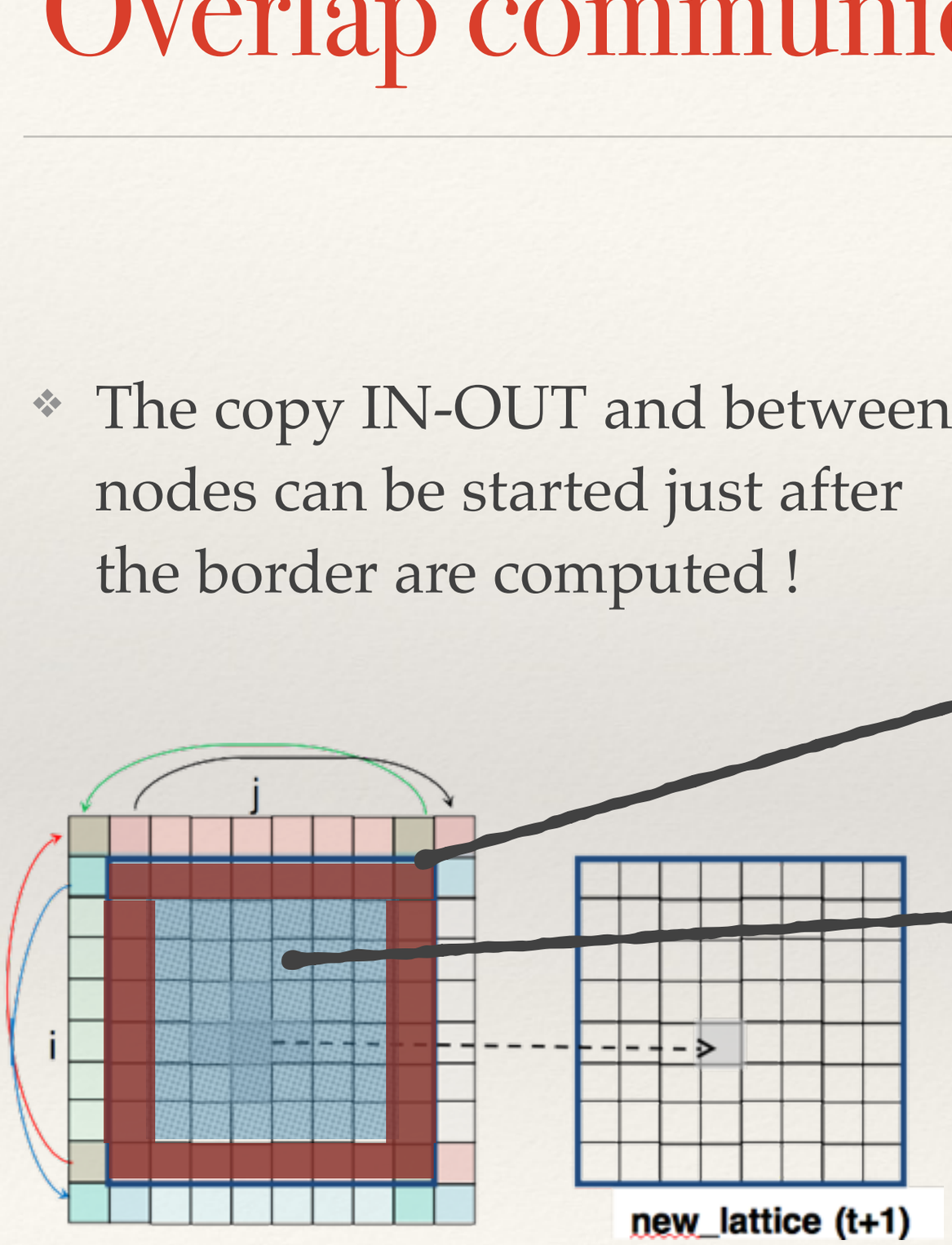
```
// Compute Internals
#pragma acc kernels present(grid[nrows+2][ncols+2], \
            next_grid[nrows+2][ncols+2],sum,A[0:ncomp],B[0:ncomp]) async(2)
{
  #pragma acc loop gang(100) independent
  for (i=rmin_int; i<=rmax_int; i++) {  // righe
    #pragma acc loop workers independent
```

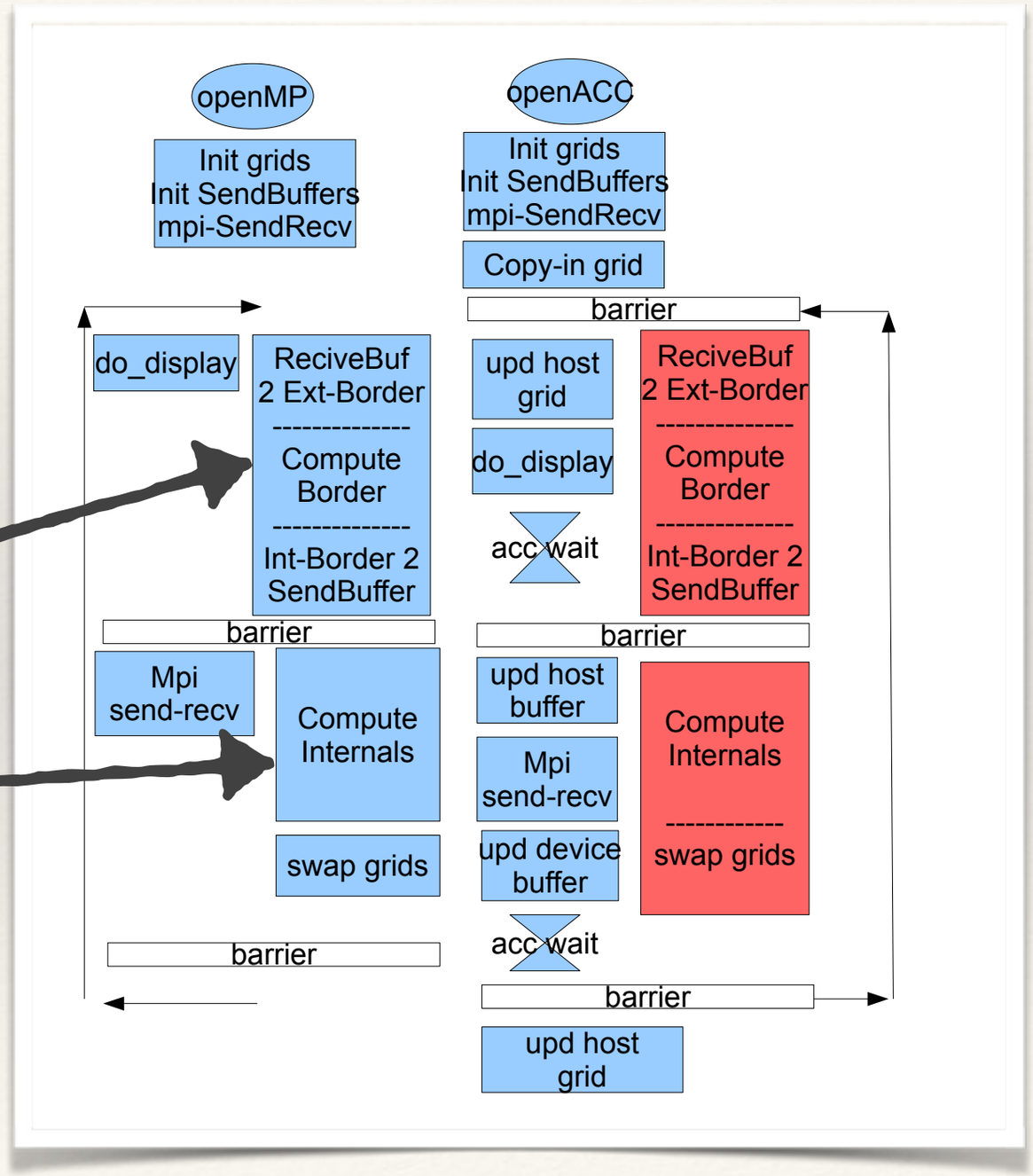
```
// Compute Internals
#pragma acc parallel present(grid[nrows+2][ncols+2],\
      next_grid[nrows+2][ncols+2],sum,A[0:ncomp],B[0:ncomp]) \
       async(2) num_gans(100) vector_length(16)
{
  #pragma acc loop gang independent
  #pragma omp for private(i,j,k,neighbors,sum)
  for (i=rmin_int; i<=rmax_int; i++) {
    #pragma acc loop worker independent
    for (j=cmin_int; j<=cmax_int; j++) {
      #pragma ivdep
      #pragma vector aligned
      #pragma acc loop vector independent reduction(+: sum) private(sum)
      for (k=0; k < ncomp; k++)  sum += A[k] + B[k];
      // LIFE
      neighbors = grid[i+1][j+1] + grid[i+1][j] + grid[i+1][j-1]
                + grid[i][j+1] + grid[i][j-1] + grid[i-1][j+1]+grid[i-1][j]
                +grid[i-1][j-1];
      if ( ( neighbors > 3.0 ) || ( neighbors < 2.0 ) )
        next_grid[i][j] = 0.0;
      else if ( neighbors == 3.0 )
        next_grid[i][j] = 1.0;
      else
        next_grid[i][j] =  grid[i][j];
    }
  }
}  // end Compute Internals
```

http://insidehpc.com/2012/08/30/michael-wolfe-on-openacc-kernels-and-parallel-constructs/
https://www.pgroup.com/lit/articles/insider/v4n2a1.htm

Overlap communication/computation

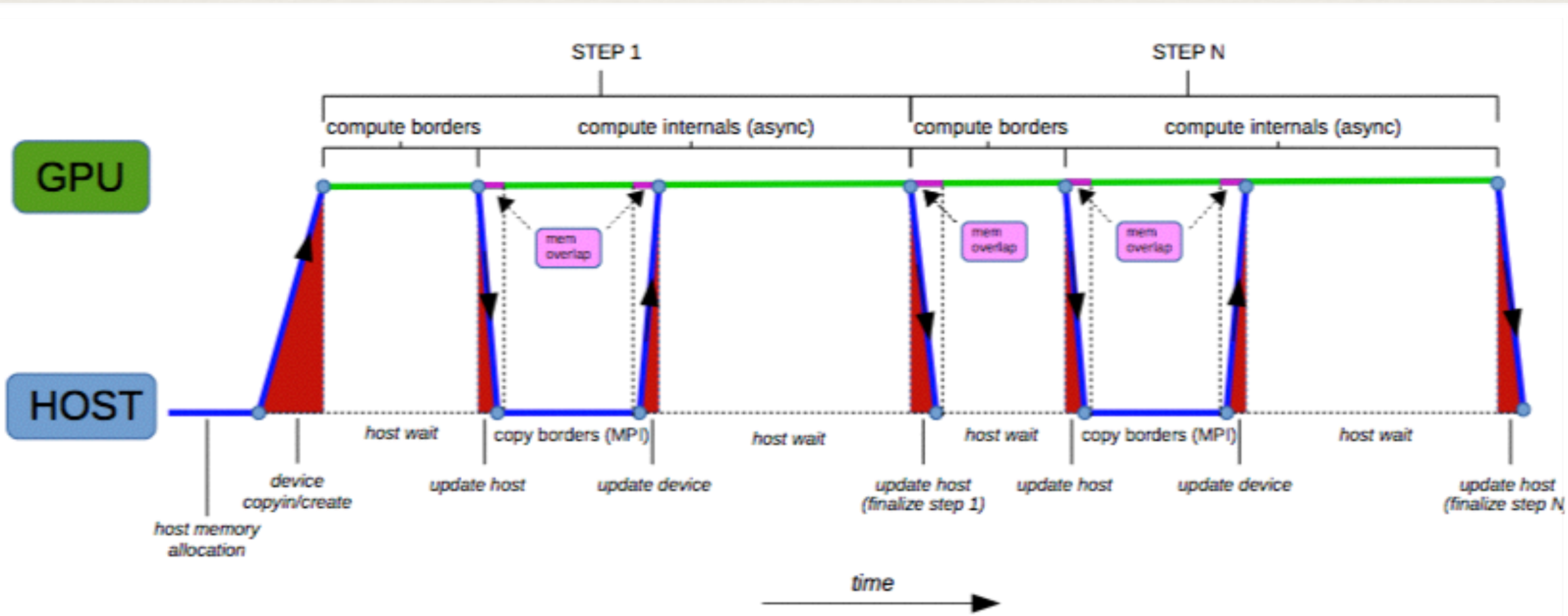
- The copy IN-OUT and between nodes can be started just after the border are computed !





Overlap communications and computations

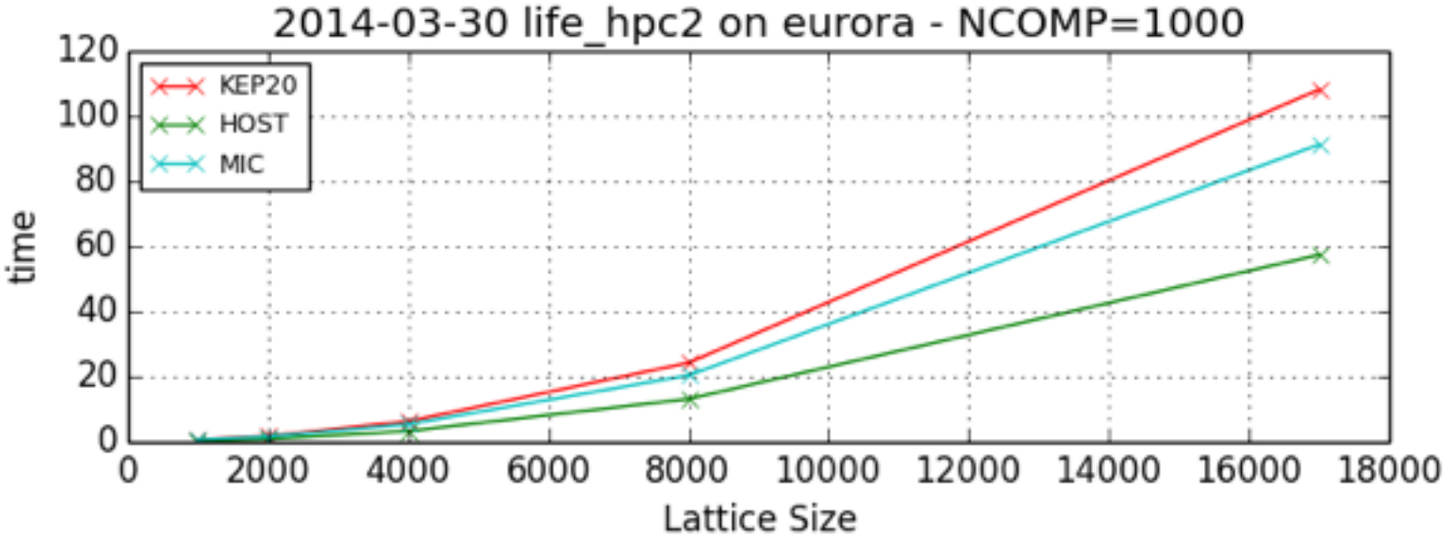
- The diagram below show how the communications between the GPU and the CPU and the computations got split

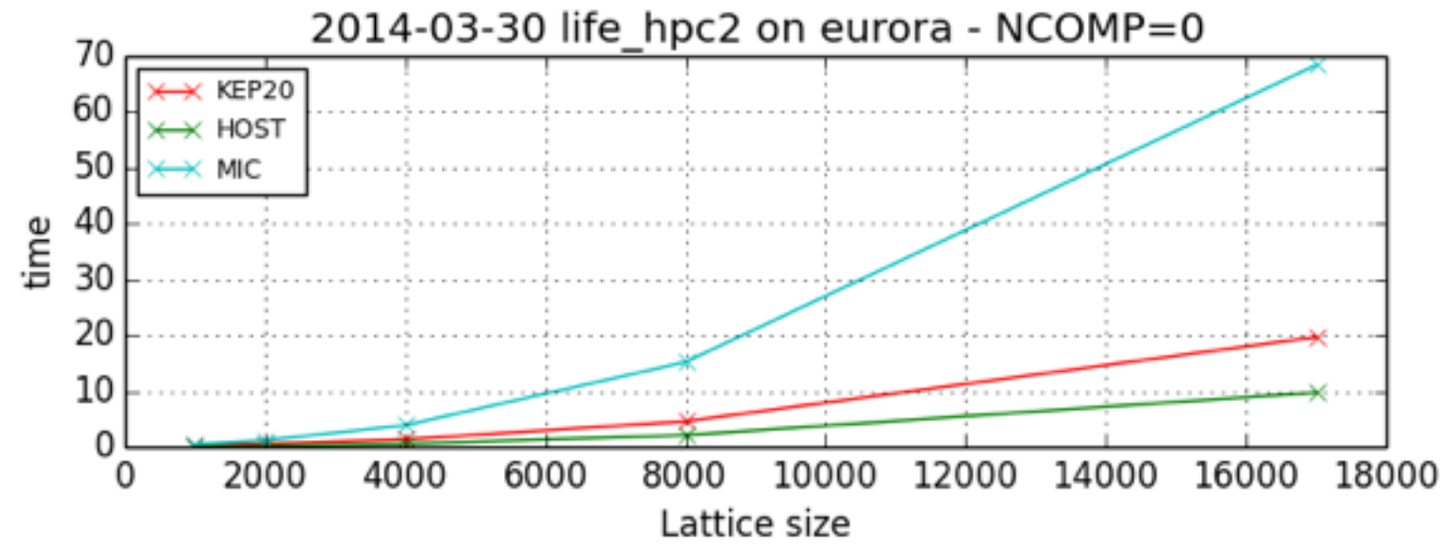


The comparison of the three “devices”

- 17000x17000 is the maximum size we can load on K20 GPU
- Nc=0 : 26GFlop
- Nc=1000 : 5.28TFlop
- **K20** is declared of having a peak performance of 1.17 Tflops
- The single node on *Eurora* 16*2.1*8= 0.268 Tflops

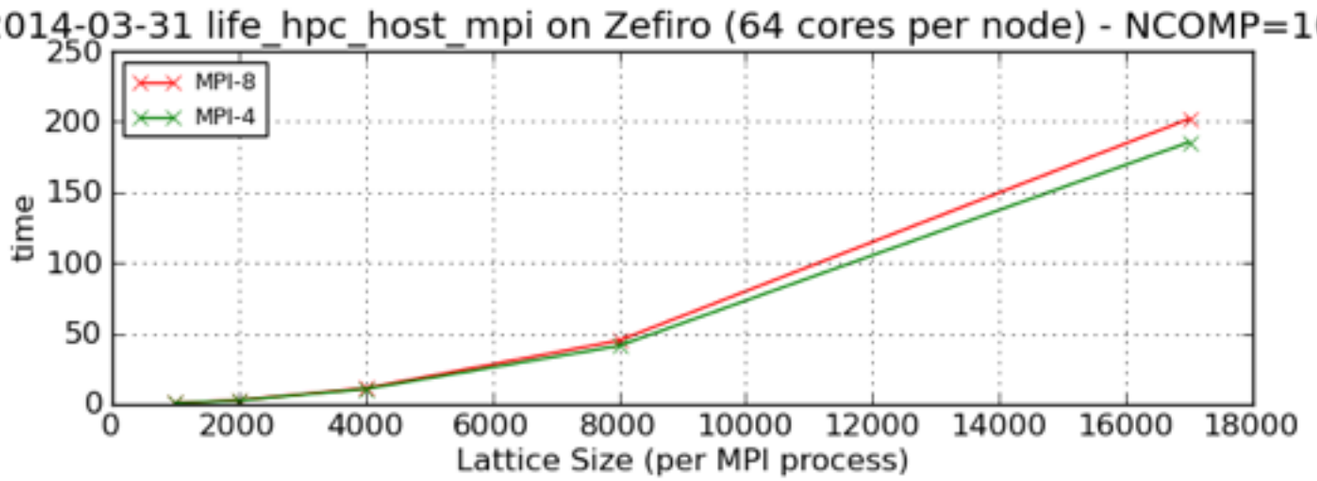
 # pragma acc kernel

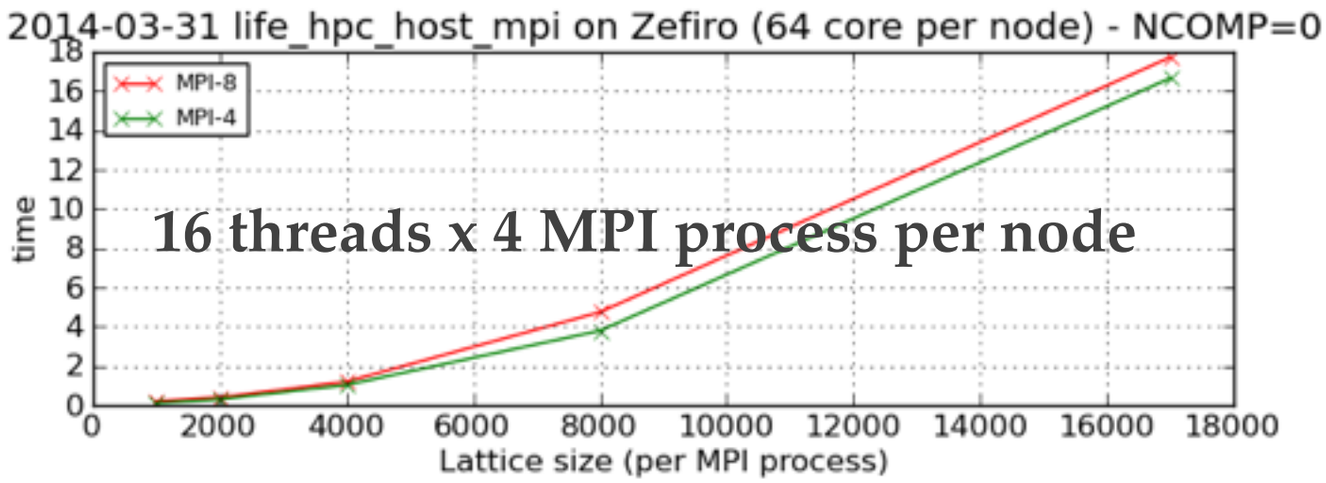




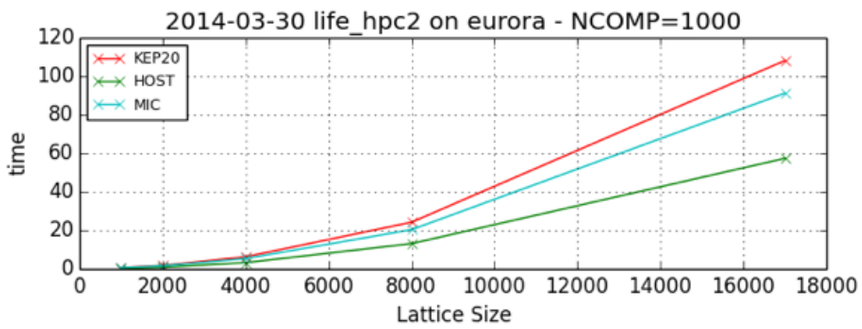
Zefiro .vs. Eurora

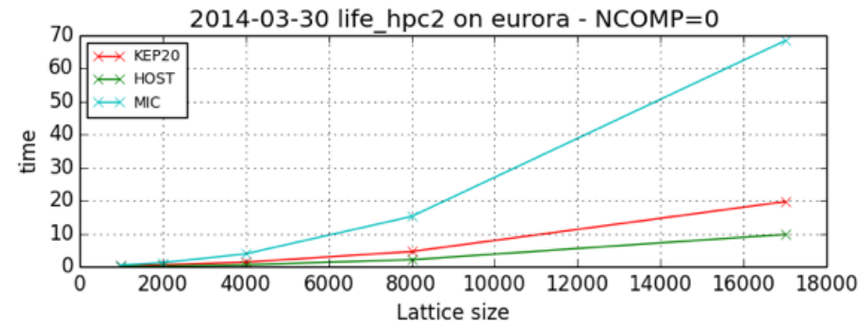
- The same test on ZEFIRO

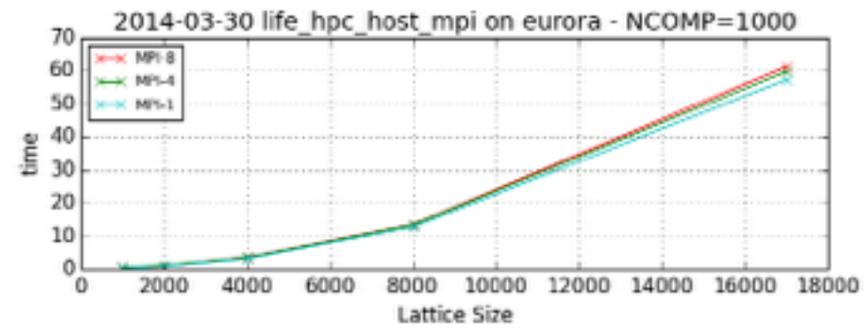


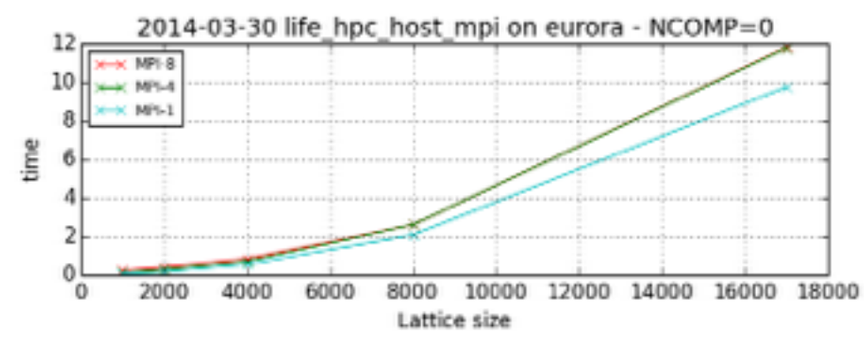


16 threads x 4 MPI process per node



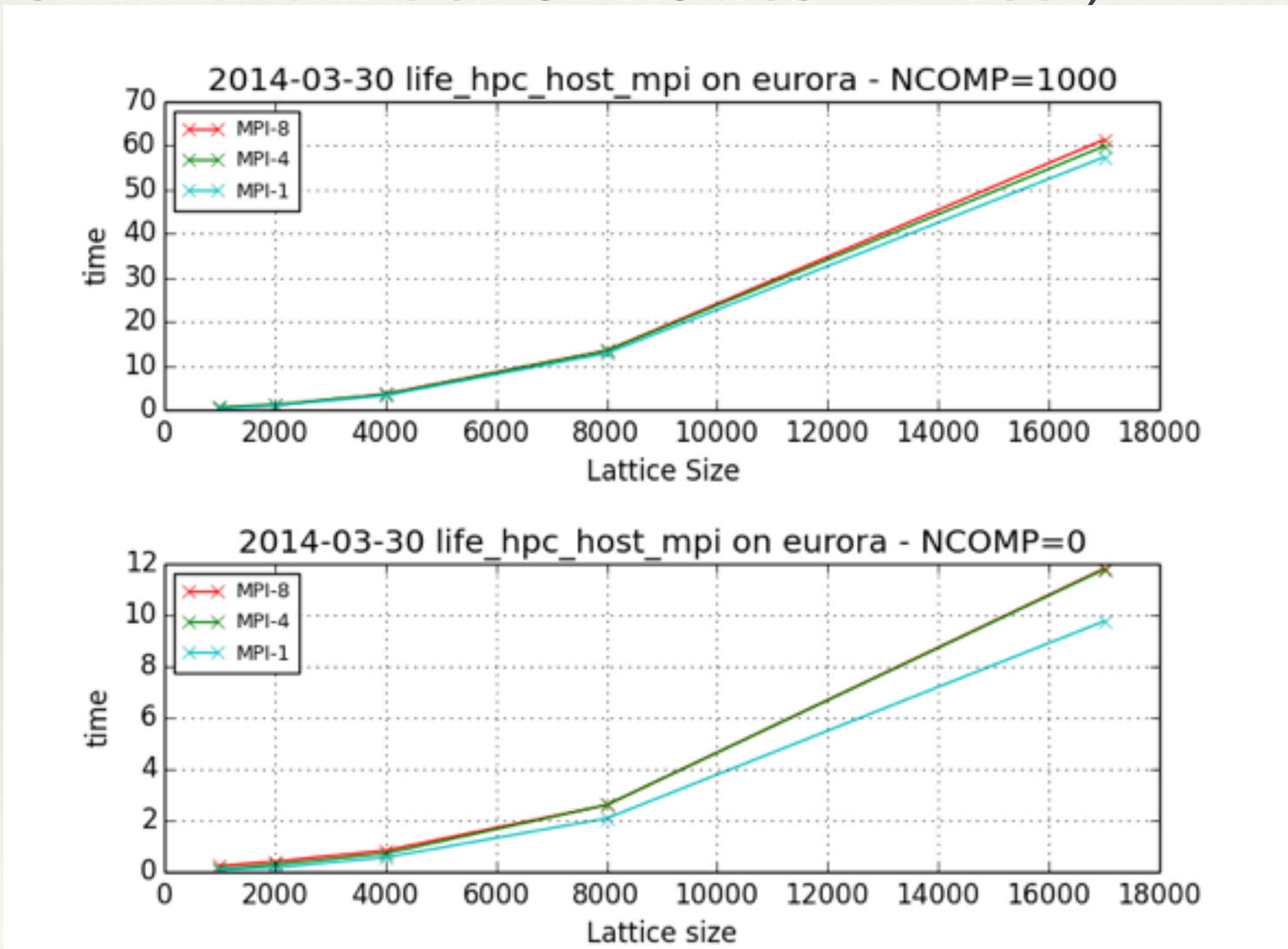






Test MPI – "eurora" -1-4-8 nodi

- First test on EURORA with some MPI communications (still a small number of nodes —- icc)



Conclusions.....

- OpenMP / OpenACC are very promising compiler technologies for creating portable programs that run on different “future” architectures for HPC systems

- Still difficult to get improvement with respect to OpenMP on a full node.
- Performance are still not satisfactory !
- We hope to get better performances going to more Physical systems where the computational load is better balanced.