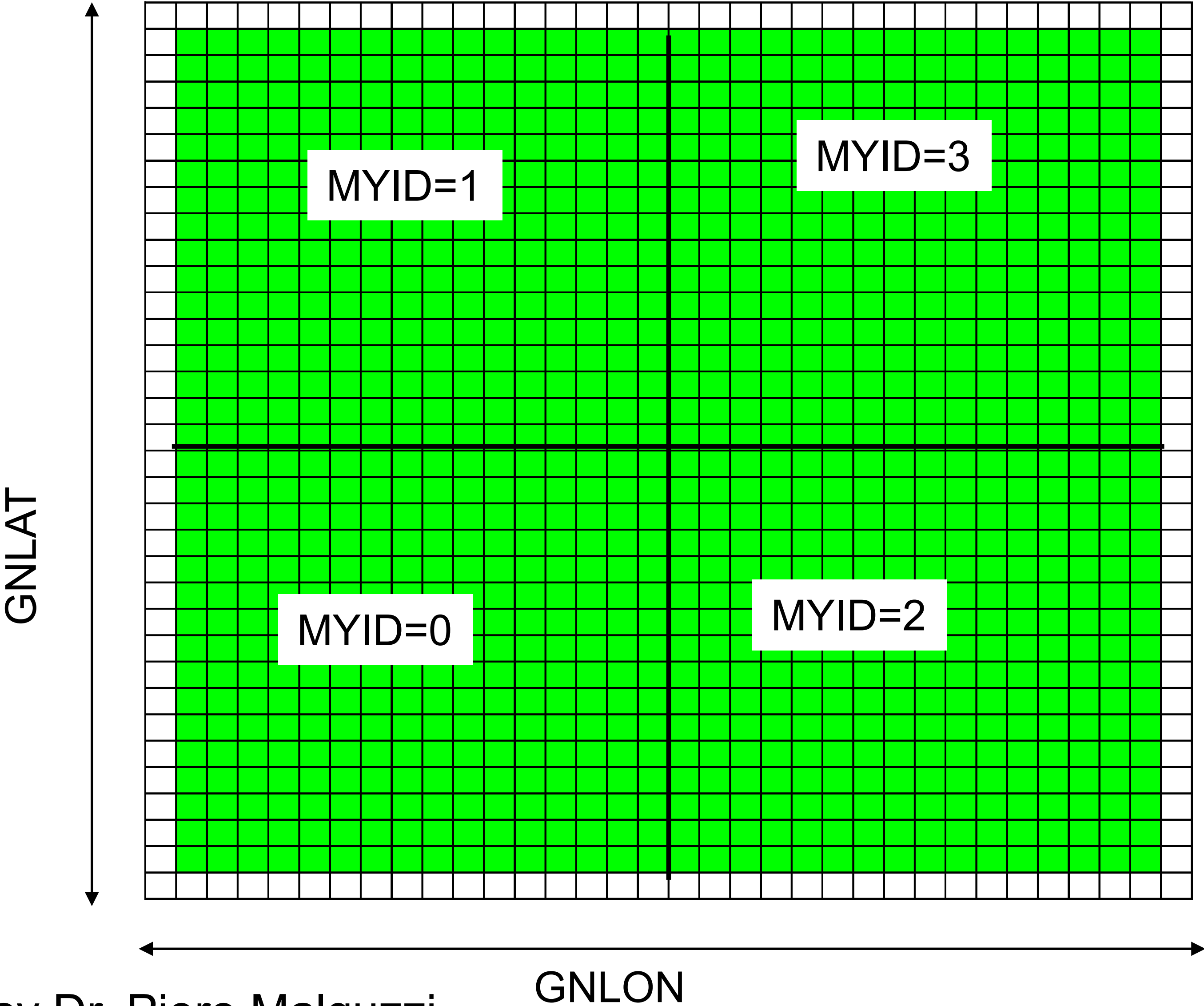
Developing (and mantaining) large, parallel, Fortran codes to modern (possibly heterogeneous) computer architectures

Piero Lanucara (CASPUR)

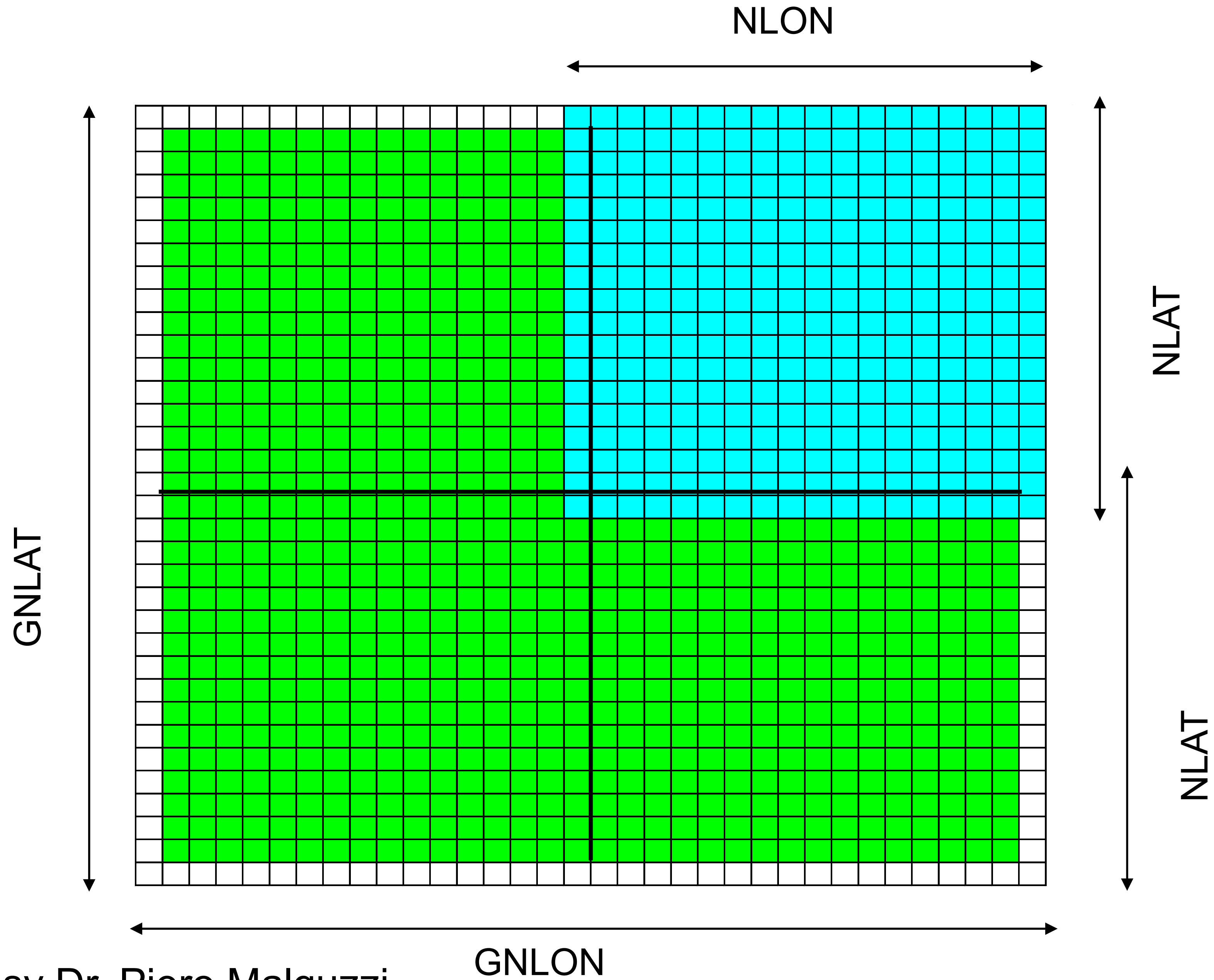
Bolam parallelization

MPI based ‘Domain decomposition’



slide courtesy Dr. Piero Malguzzi

Halo comp is introduced for exchange boundary values



slide courtesy Dr. Piero Malguzzi

NPROCSX, NPROCSY

NLON=(GNLON-2)/NPROCSX+2

NLAT=(GNLAT-2)/NPROCSY+2

GNLON=(NLON-2)*NPROCSX+2

GNLAT=(NLAT-2)*NPROCSY+2

(NLON e NLAT numeri pari)

slide courtesy Dr. Piero Malguzzi

Bolam Speedup

- Architecture dependent
- Hyper-trading?
- Mapping of the physical domain onto core/processors
- Mostly 8 with 16 processes
- Speedup saturation with huge number of cores (>100)

“Future” Bolam development?

- SMS parallelization
- Exploiting communication/computation patterns
- Porting onto GPU architectures
- Other….

A non standard approach towards parallelization

- The Problem: porting of serial, large Fortran legacy applications to cluster

MPI ? Of course but….

- MPI learning curve is too high for standard researchers
- MPI is targeted for performance but huge skills are needed for this purpose
- Not so easy to mantain different release of the same code

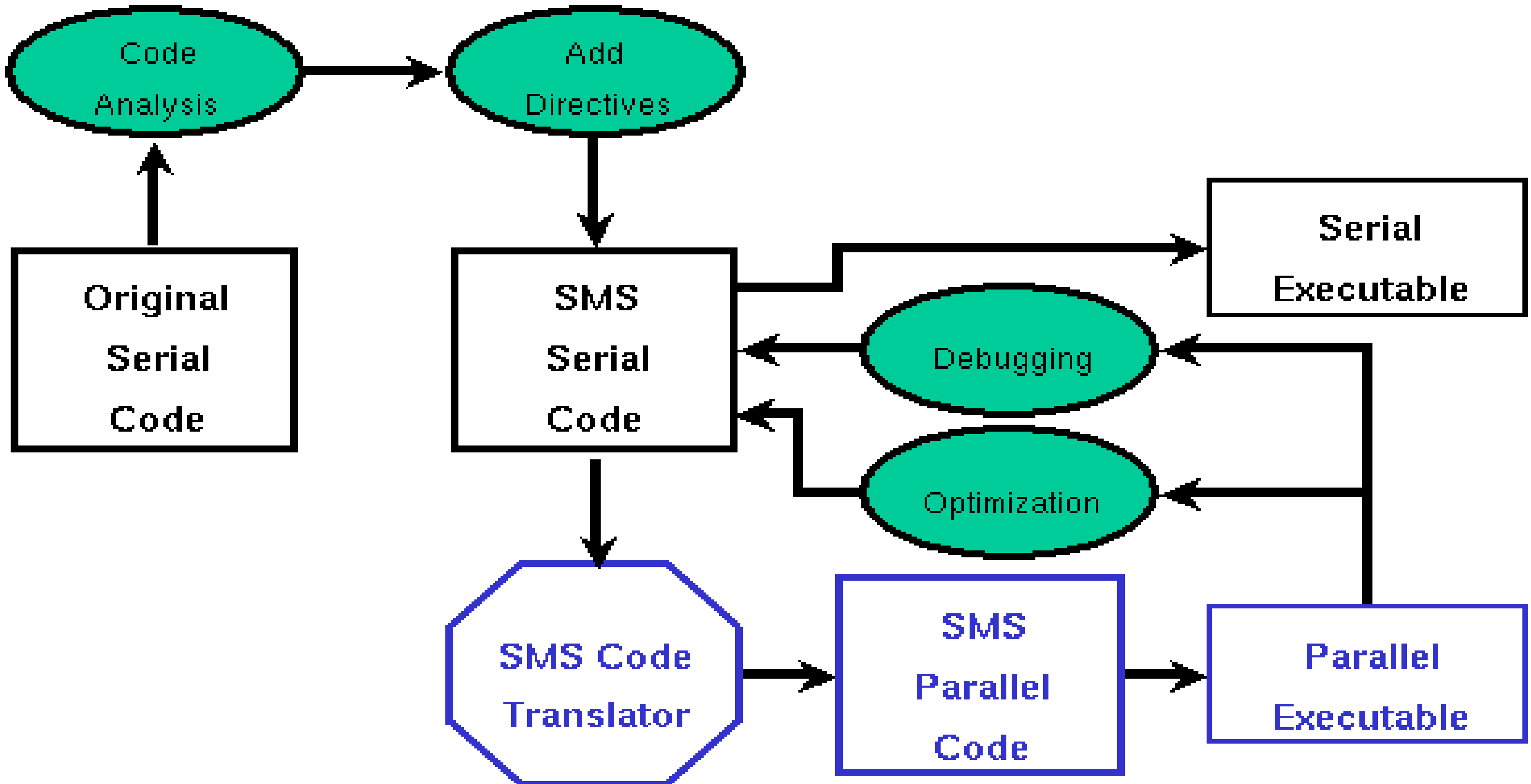
New Parallel version of Princeton Ocean Model (CEPOM)

- Target: porting of POM serial code to clusters without using MPI
- The tool: Scalable Modeling System (SMS), which is a tool developed at NOAA by the Advanced Computing Section of NOAA's Earth System Research Laboratory team

SMS key features:

- Directive based (“a la OpenMP”): portability
- Source to source translator: only one code
- MPI based tool: efficiency across different platform

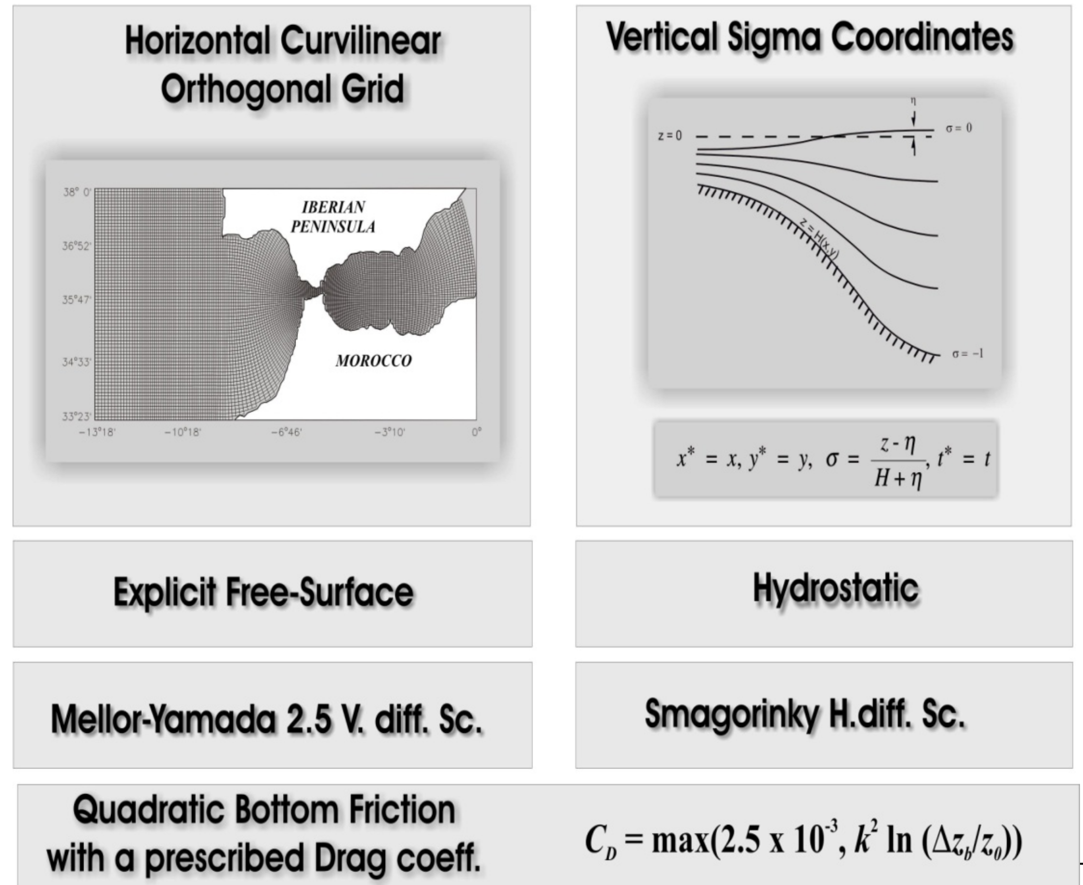
SMS Workflow



The CEPOM model

CEPOM is a modified version of the Princeton Ocean Model code developed together with ENEA Casaccia researchers

The code is completely parallelized using SMS Tool



High resolution studies within the Strait of Gibraltar

Development of a 3D seismic code

- Source code from Praga University in the framework of a collaboration with INGV Roma (**Site Effects study**)
- Fortran 95 code that it solves elasticity PDE using finite difference technique (some code rewriting in order to better support SMS parallelization)
- **563x852x94** huge computational grid
- Only 500 SMS directives needed to parallelize the code; good parallel performances up to hundreds of cores for realistic simulations

Exploiting communication/computation patterns

- Source of bottleneck in highly parallel codes
- Typical workaround includes:
  - Overlapping communication and computation
  - Low communication High extra computation numerical schemes (redundant computation)

Overlapping communication and computation

- Goal: reduce the cost of waiting for data transfer, overlapping this communication with local computation
- Some technique:
  - Overdecomposition
  - Non-blocking communication (it includes double buffering, posting receivers early….)

Exploiting LCHC schemes

- This strategy can yeld significan performance benefits (application dependent)
- Using redundant computation for:
  - Eliminate (or reduce) communications latency
  - Reduce bandwidth usage

Exploiting LCHC schemes

- A simple example taken from SMS Manual

Adobe Reader - ScalableModelingSystem.pdf

File Edit View Document Tools Window Help

Save a Copy Search Select 100% Help Organize your digital photos

```
CSMS$HALO_UPDATE( a<1,1>)
  do 150 i= 2, IM-1
         y(i) = a(i) - a(i+1) - a(i-1)
         z(i) = a(i) - (a(i+1) - a(i-1)) * 0.5
  150 continue
CSMS$HALO_UPDATE( y<1,1>,z<1,1> )
      do 250 i= 2, IM-1
         x(i) = y(i)*z(i) + y(i+1)*z(i-1)
              + y(i-1)*z(i+1)
  250 continue
```

Figure 11: Sample code where no redundant computations are performed. An exchange of arrays "y" and "z" are required for loop 250 to produce the correct answer on each processor.

```
CSMS$HALO_UPDATE(a<2,2>)
CSMS$HALO_COMP(<1,1>) BEGIN
      do 150 i= 2, IM-1
         y(i) = a(i) - a(i+1) - a(i-1)
         z(i) = a(i) - (a(i+1) - a(i-1)) * 0.5
  150 continue
CSMS$HALO_COMP END
      do 250 i= 2, IM-1
         x(i) = y(i)*z(i) + y(i+1)*z(i-1)
              + y(i-1)*z(i+1)
  250 continue
```

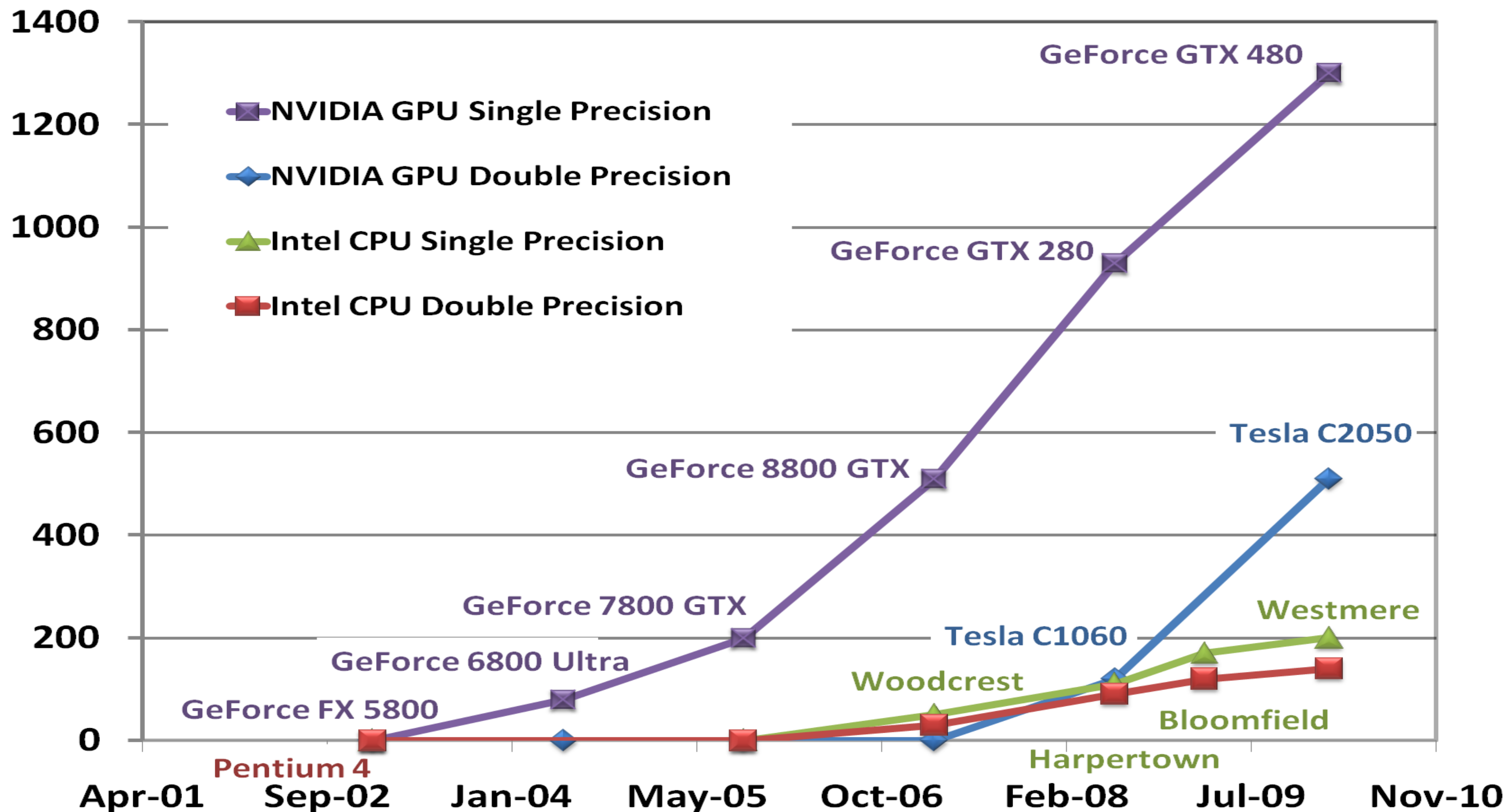
Figure 12: A version of the same code that uses redundant computation. Since "y" and "z" are computed one step into the halo region, their halo regions are up to date after loop 150. Consequently, the exchanges of "y" and "z" after loop 150 can be eliminated.

This strategy is a technique that can yield significant performance benefits. In recent tests

17 of 26

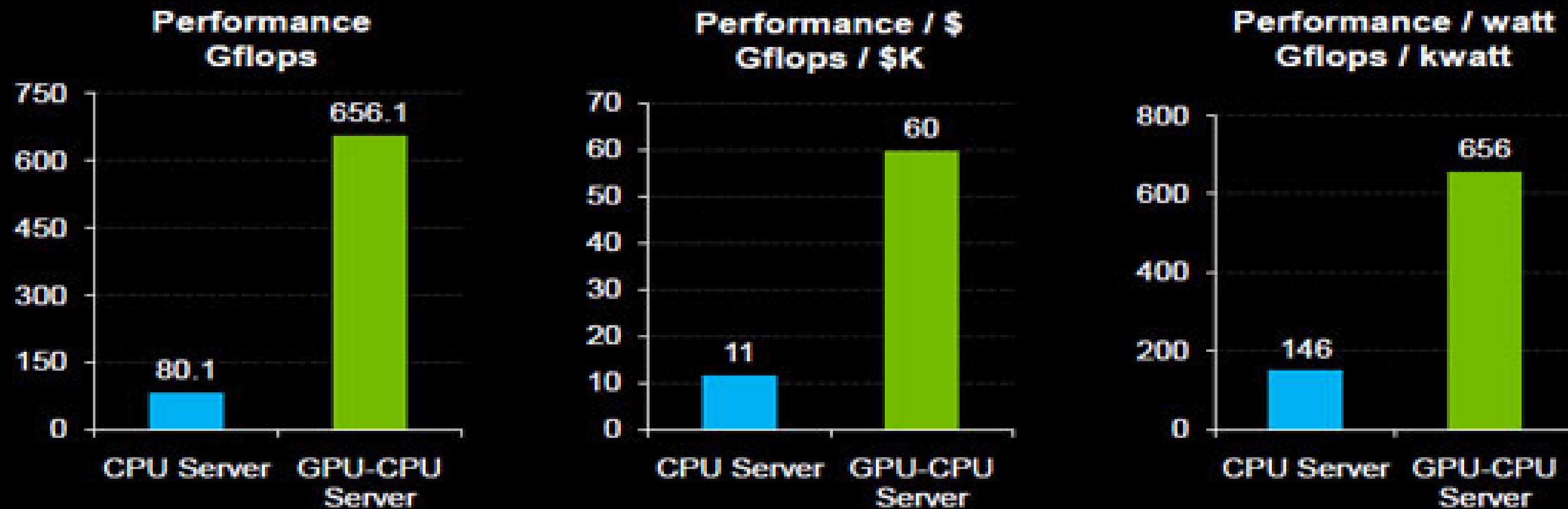
The GPU explosion

- A huge amount of computing power: exponential growth with respect to "standard" multicore CPUs



The GPU explosion

- Affordable price/performance and performance/watt systems (the so called “Green Computing”)



Jazz Fermi GPU Cluster at CASPUR

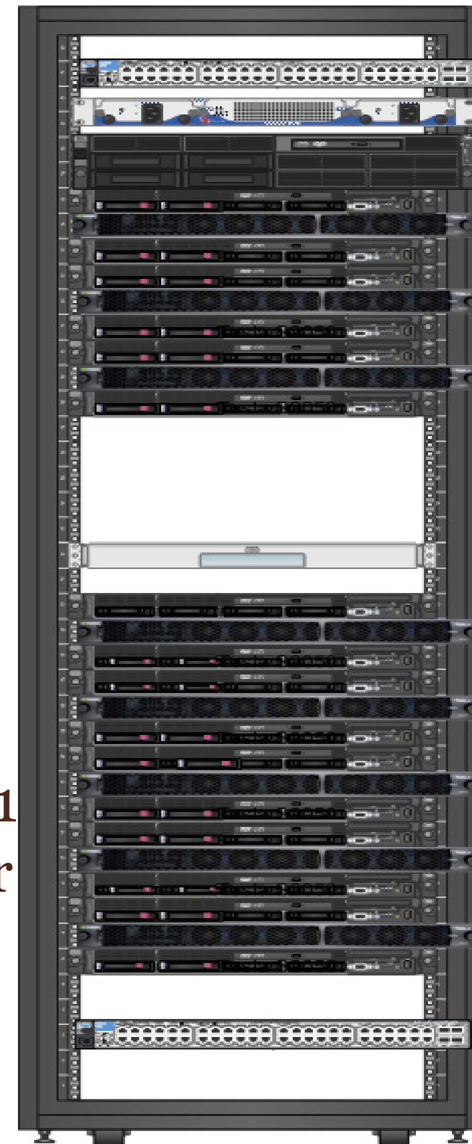
785 MFlops/W

192 cores Intel X5650@2.67 Ghz
14336 cores on 32 Fermi C2050
QDR IB interconnect
1 TB RAM
200 TB IB storage

14.3 Tflops Peak

CASPUR awarded as CUDA Research Center for 2010-2011
Jazz cluster is actually number 5 of Little Green List

10.1 Tflops Linpack



DL360 G7

The New Problem:Porting large Fortran codes to **GPU** systems

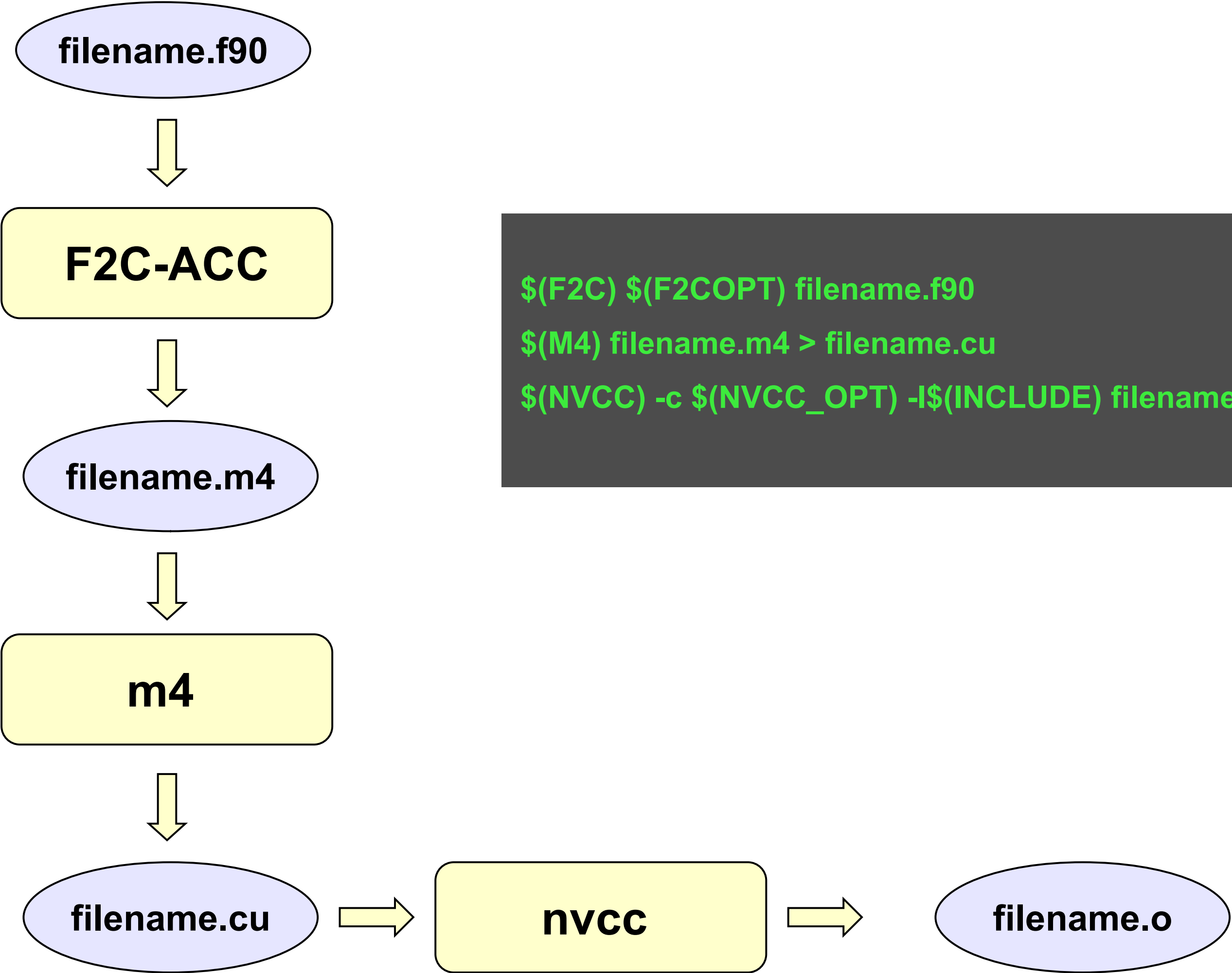
- CUDA is the “de-facto” standard for efficiently programming GPU clusters...
- ….but at this moment the standard is targeted for C (now C++ also) application
- **How to port large Fortran legacy codes?**
  - Rewriting the application in C/CUDA (easy?)
  - Using the PGI Accelerator (efficient?) or CUDA Fortran (is free?)
  - **F2C-ACC Compiler**

F2C-ACC compiler

- F2C-ACC was developed at NOAA by the same team of SMS (Mark Govett et al.)...
- ….in order to reduce the porting-time of a legacy Fortran code to GPUs
- It works well with Fortran 77 codes plus some extension towards Fortran 95 (most of them!)
- F2C-ACC is an “open” project. Actual release is 3.0

http://www.esrl.noaa.gov/gsd/ab/ac/F2C-ACC.html

How F2C-ACC partecipates “in make”




```
$(F2C) $(F2COPT) filename.f90
$(M4) filename.m4 > filename.cu
$(NVCC) -c $(NVCC_OPT) -I$(INCLUDE) filename.cu
```

F2C-ACC: Fortran source code

```
subroutine accdata(vol,flx)

implicit none
integer k,ipn
integer nz,nip
parameter (nz=5,nip=10)
real   ,intent (IN)    :: vol  (nz,nip)
real   ,intent (INOUT) :: flx(nz,nip)

! the "in" data argument indicates the data should be copied to the gpu
! all arguments used in the acccelerated region will be copied based on the intent
! of the variable
!ACC$REGION(<nz>,<nip>,<flx:in>,<vol:none>) BEGIN
!acc$do parallel
do ipn=1,nip
!acc$do vector
  do k=1,nz
    flx(k,ipn) = flx(k,ipn)/vol(k,ipn)
  end do
end do
!ACC$REGION END

! overrides the INOUT default designed in the routine declaration
!ACC$REGION(<nz>,<nip>,<flx:inout>,<vol:none>) BEGIN
!acc$do parallel
do ipn=1,nip
!acc$do vector
  do k=1,nz
    flx(k,ipn) = flx(k,ipn)/vol(k,ipn)
  end do
end do
!ACC$REGION END

end subroutine accdata
```

F2C-ACC: CUDA (parsed)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cutil.h>
#include "ftocmacros.h"
#define TRUE 1
#define FALSE 0
#define nz 5
#define nip 10
//!ACC$REGION(<nz>,<nip>,<flx:in>,<vol:none>) BEGIN
__global__ void accdata_Kernel1(float *vol,float *flx) {
 int ipn;
 int k;
//!acc$do parallel
  ipn = blockIdx.x+1;
//  for (ipn=1;ipn<=nip;ipn++) {
//!acc$do vector
    k = threadIdx.x+1;
//    for (k=1;k<=nz;k++) {
      flx[FTNREF2D(k,ipn,nz,1,1)] = flx[FTNREF2D(k,ipn,nz,1,1)] /
vol[FTNREF2D(k,ipn,nz,1,1)];
//    }
//  }
  return;
}
//!ACC$REGION END
//!ACC$REGION(<nz>,<nip>,<flx:inout>,<vol:none>) BEGIN
__global__ void accdata_Kernel2(float *vol,float *flx) {
 int ipn;
 int k;
//!acc$do parallel
  ipn = blockIdx.x+1;
//  for (ipn=1;ipn<=nip;ipn++) {
//!acc$do vector
    k = threadIdx.x+1;
//    for (k=1;k<=nz;k++) {
      flx[FTNREF2D(k,ipn,nz,1,1)] = flx[FTNREF2D(k,ipn,nz,1,1)] /
vol[FTNREF2D(k,ipn,nz,1,1)];
//    }
```

```
//! ACC$REGION
//!   Variables must be defined either as input arguments, or as
constants.
//!
extern "C" void accdata_ (float *vol,float *flx) {
  static float runtime = 0.0;
//! the "in" data argument indicates the data should be copied to
the gpu
//! all arguments used in the acccelerated region will be copied
based on the intent
//! of the variable
  dim3 cuda_threads1(nz);
  dim3 cuda_grids1(nip);
      // create the timer
  unsigned int timer = 0;
  extern float *d_vol;
  float *d_flx;
  cudaMalloc((void **) &d_flx,((nz)*(nip))*sizeof(float));

cudaMemcpy(d_flx,flx,((nz)*(nip))*sizeof(float),cudaMemcpyHostT
oDevice);
  cutCreateTimer(&timer);
  cutStartTimer(timer);
  // setup execution parameters
  // execute the kernel
  accdata_Kernel1<<< cuda_grids1, cuda_threads1
>>>(d_vol,d_flx);
  cudaThreadSynchronize();
  // check if kernel execution generated an error
  CUT_CHECK_ERROR("Kernel execution failed");
  cutStopTimer(timer);
  float timer__G = cutGetTimerValue(timer);
  cutDeleteTimer(timer);
  runtime += timer__G;
  //printf("accdata kernel execution time: %f \n",runtime/1000);
//! overrides the INOUT default designed in the routine declaration
  dim3 cuda_threads2(nz);
  dim3 cuda_grids2(nip);
      // create the timer
```

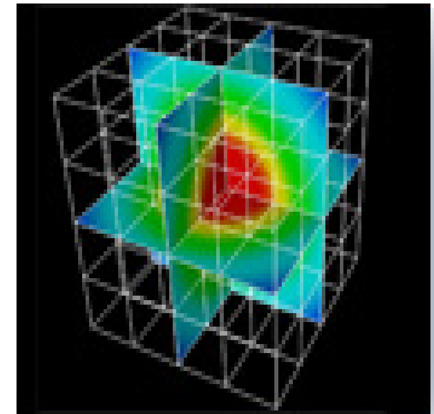
F2C-ACC Workflow

- F2C-ACC translates Fortran code, with user added directives, in CUDA (relies on m4 library for interlanguages dependencies)
- **Some hand coding could be needed (see results)**
- Debugging and optimization Tips (e.g. Thread, block synchronization, out of memory, coalesce, occupancy....) are to be done manually

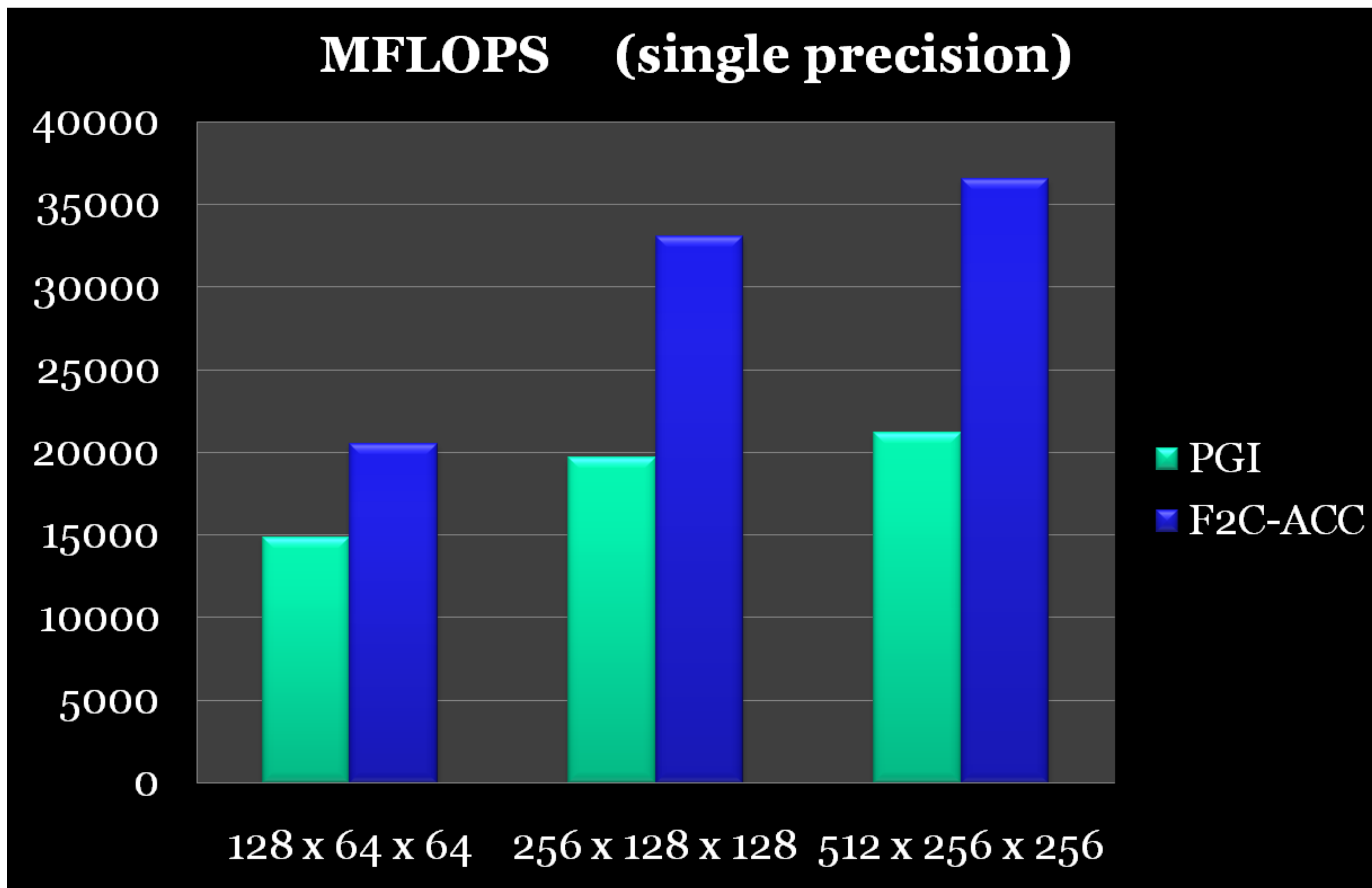
Compile and linking using CUDA libraries to create an executable to run

Himeno Benchmark

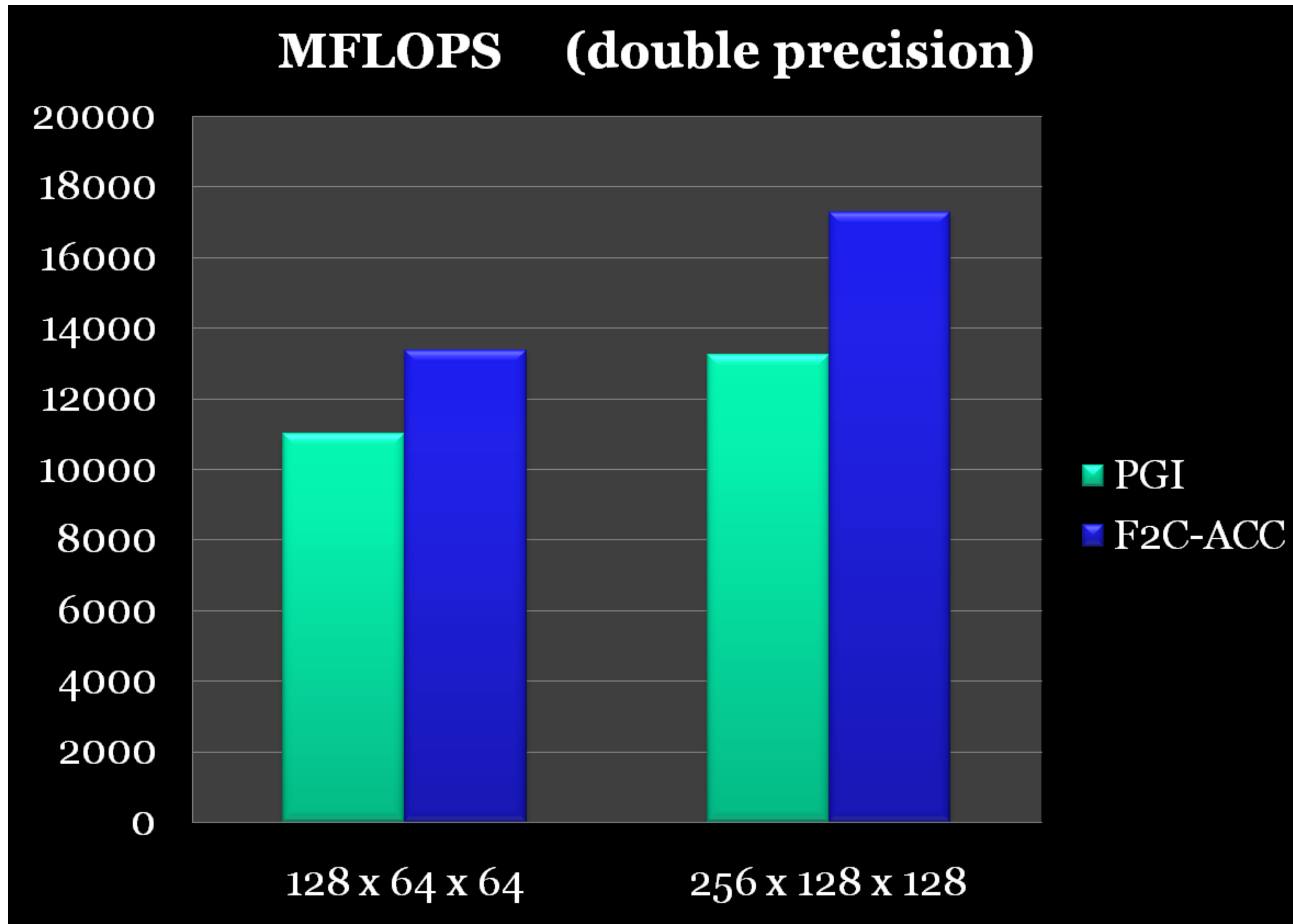
- Developed by Dr. Ryutaro Himeno
- Implement a sort of 3D Poisson Solver using an iterative scheme to converge
- Measures the performance in FLOPS (different grid size from Small to XLarge)
- **Ported to GPUs (PGI Accelerator)**



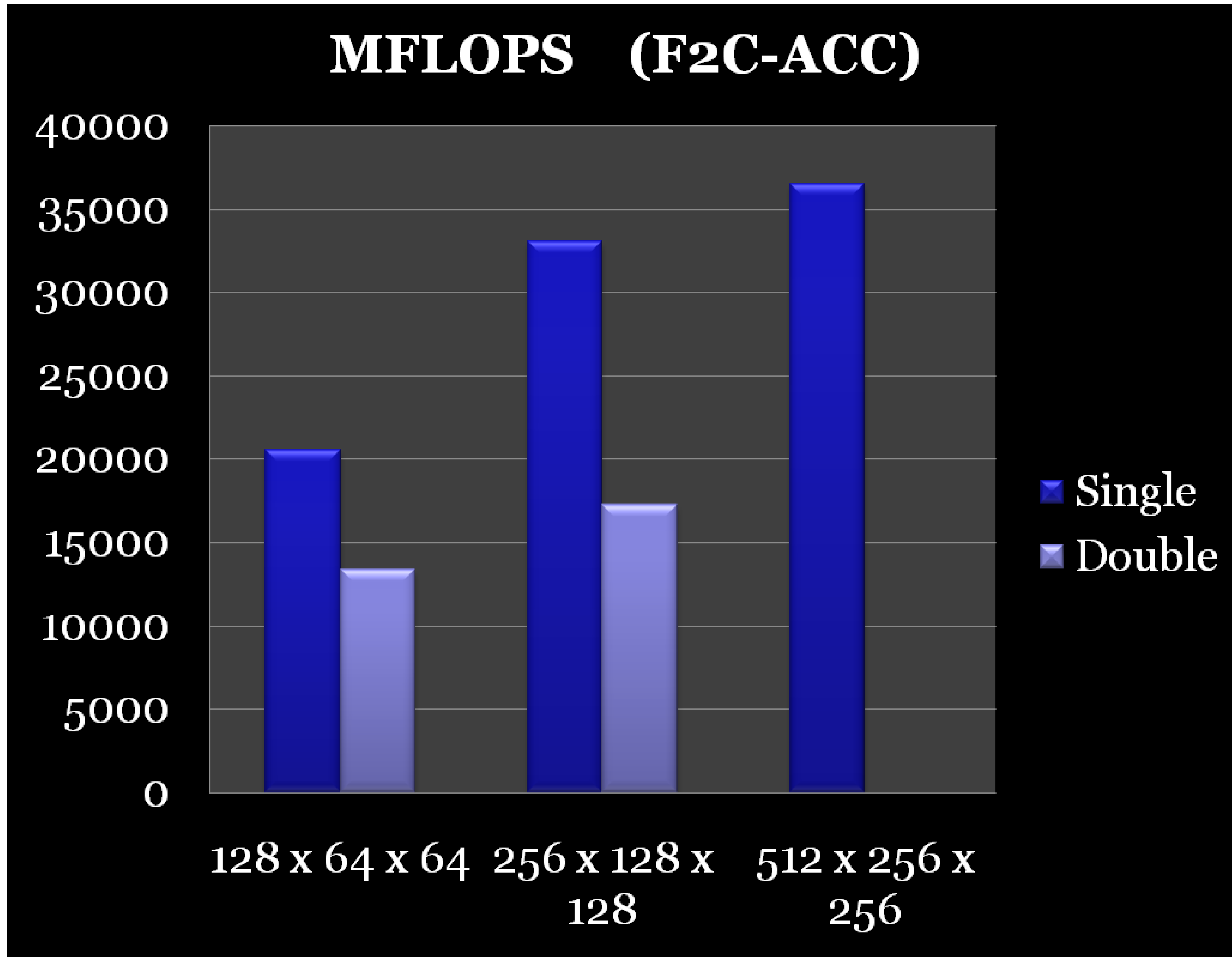
Himeno Benchmark



Himeno Benchmark



Himeno Benchmark



Himeno Benchmark

Time (s)	128 x 64 x 64	256 x 128 x 128	512 x 256 x 256
Serial	20.4014	173.5288	1598.4759
F2C-ACC	4.0147	20.7427	153.0812

GOSA	128 x 64 x 64	256 x 128 x 128	512 x 256 x 256
Serial	1.1000150E-06	1.5260409E-04	3.3477170E-04
F2C-ACC	1.0992105E-06	1.5201638E-04	3.2989052E-04

Himeno Benchmark: some comment

- Very good results expecially for single precision F2C-ACC
- PGI Accelerator is able to solve sum reduction (GOSA) and it generates efficient parallel code
- We did some CUDA hand coding to help F2C-ACC to fix this problem (same numerical results): inherently serial portion of code but no performance lost
- The size of the FERMI device (3GB) is a limiting factor for the GPU computation of the test cases L and XL

Porting Himeno to MultiGPUs

- Preliminary results of the Himeno benchmark using F2C-ACC are encouraging
- MultiGPU and multinode processing are key factors in order to run Himeno benchmark on cluster
- I/O, data movement (CPU-CPU and GPU-CPU communications) are not negligible for the MPI application

Himeno Benchmark: MPI version

- Developed without accelerator
- Simple use of MPI calls to allow the comunication of exchange boundaries (halo comp)
- Domain decomposition only in the third dimension:
  - 2 blocks (i=512 j=256 k=128)
  - 4 blocks (i=512 j=256 k=64)
  - 8 blocks (i=512 j=256 k=32)

Himeno Benchmark: MPI version

- The MPISENDRECV call allow the communication of the right and left layers in the loop of time iteration before the accelerated region
- There are too much data communications between host and device (2 complete layers every time iteration)
- So…

Himeno Benchmark: MPI version

- For the MPI version we implement the *double buffering* technique
- we change the MPI calls to communicate only the left slide of the the left layer and the right slide of the right layer
- In this way we reduce the time for the communication that is completely saturated by the time of the computation on GPU

Himeno Benchmark: MPI version

- A sketch of the code:
  - Copy from host to device needed data (accelerated)
  - Begin iteration loop
    - MPI communication reduced to slices (double buffering)
    - Copy from host to device others data of layer (accelerated)
    - Accelerated Region
    - Copy from device to host others data of layer (accelerated)
  - End iteration loop
  - Copy from device to host needed data (accelerated)

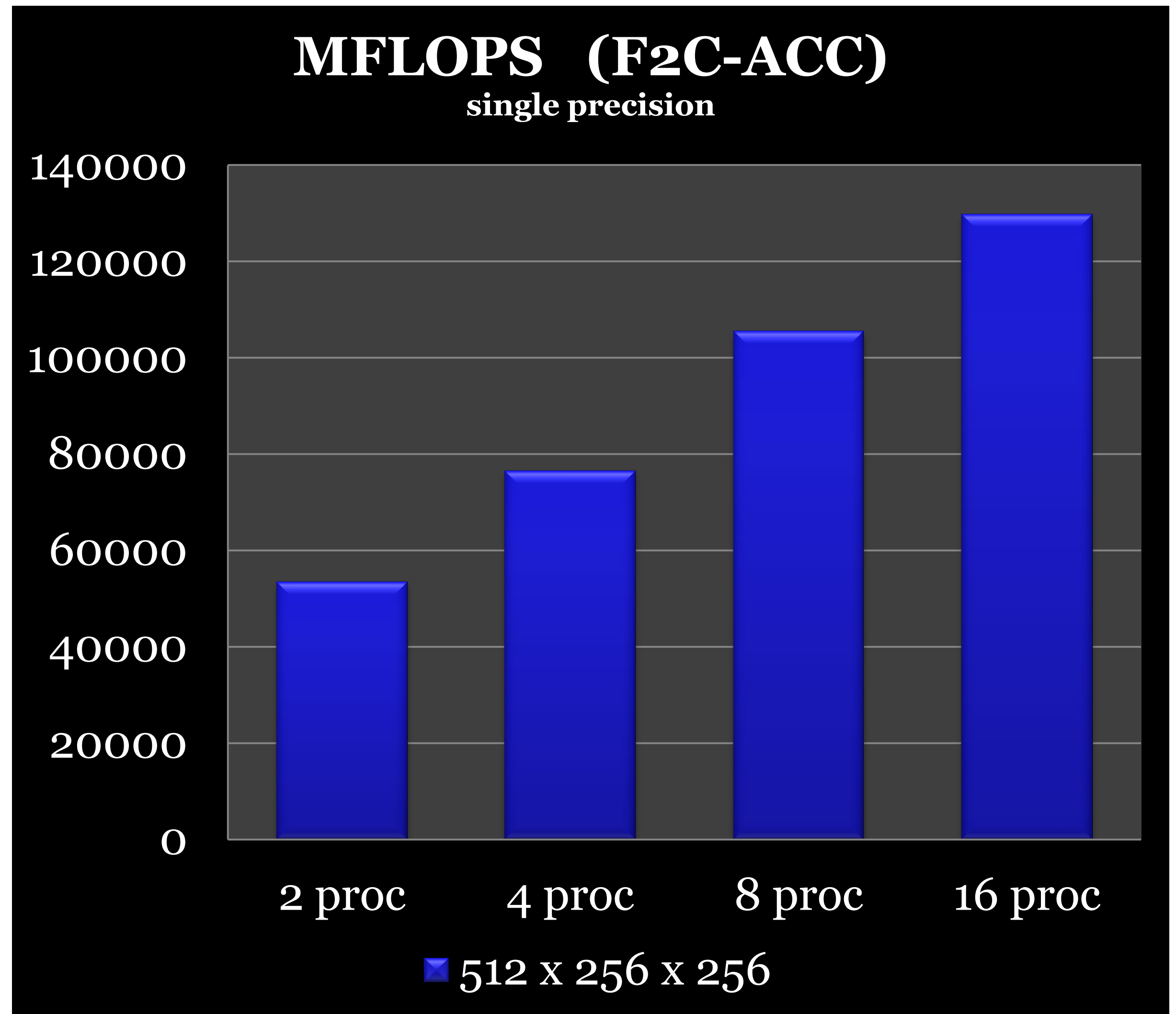
Himeno Benchmark: MPI version

- For sake of simplicity and performances, we use only F2C-ACC accelerator
- Computation kernels are the same of the serial version
- Data are in single precision
- Runs on the grid i=512 j=256 k=256

Himeno Benchmark: MPI version

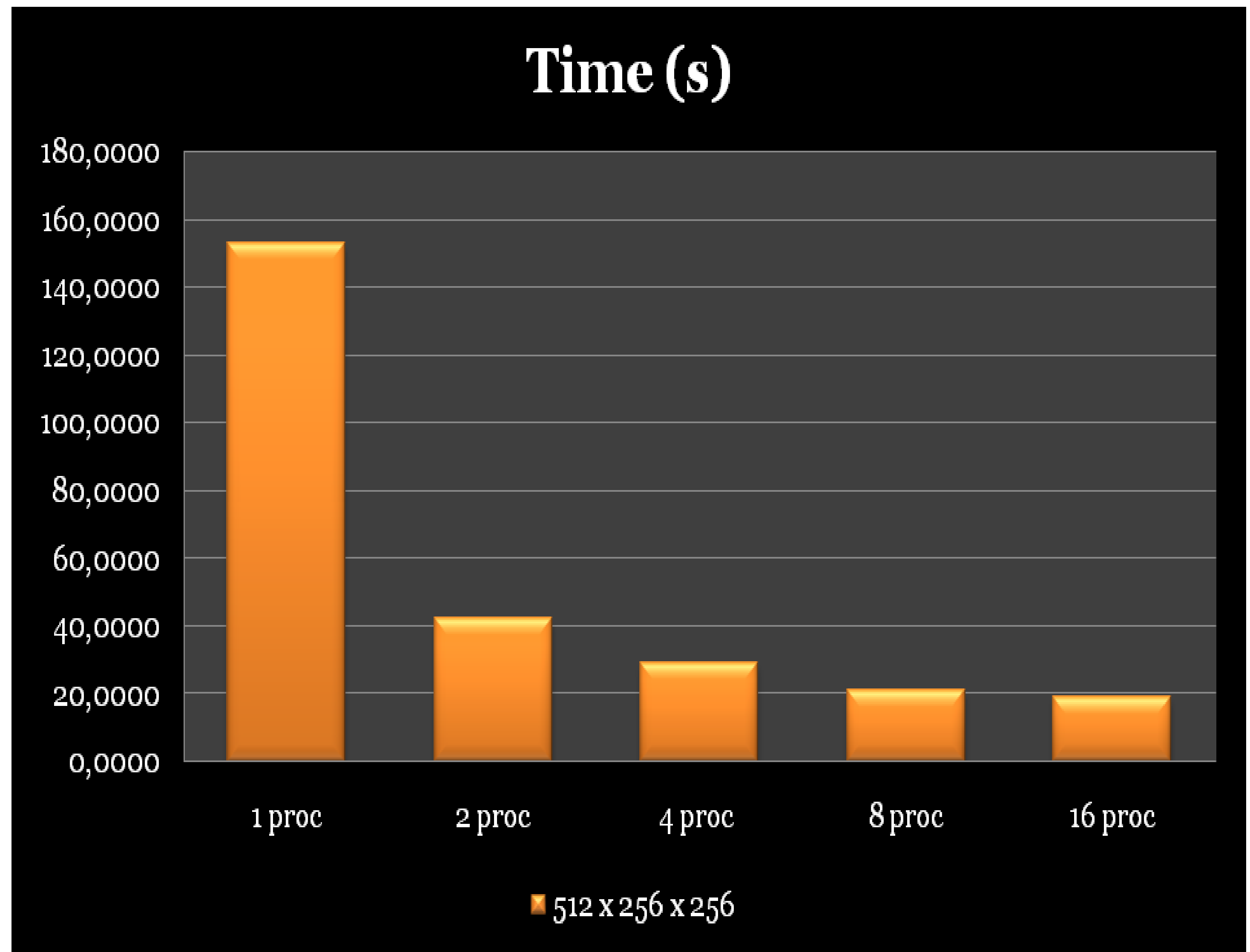
1 Process - 1 GPU

2 Process
512 x 256 x 128
4 Process
512 x 256 x 64
8 Process
512 x 256 x 32
16 Process
512 x 256 x 16



Himeno Benchmark: MPI version

512 x 256 x 256
153,0812 (s)
512 x 256 x 128
42,2807 (s)
512 x 256 x 64
29,2211 (s)
512 x 256 x 32
21,1991 (s)
512 x 256 x 16
19,1053 (s)

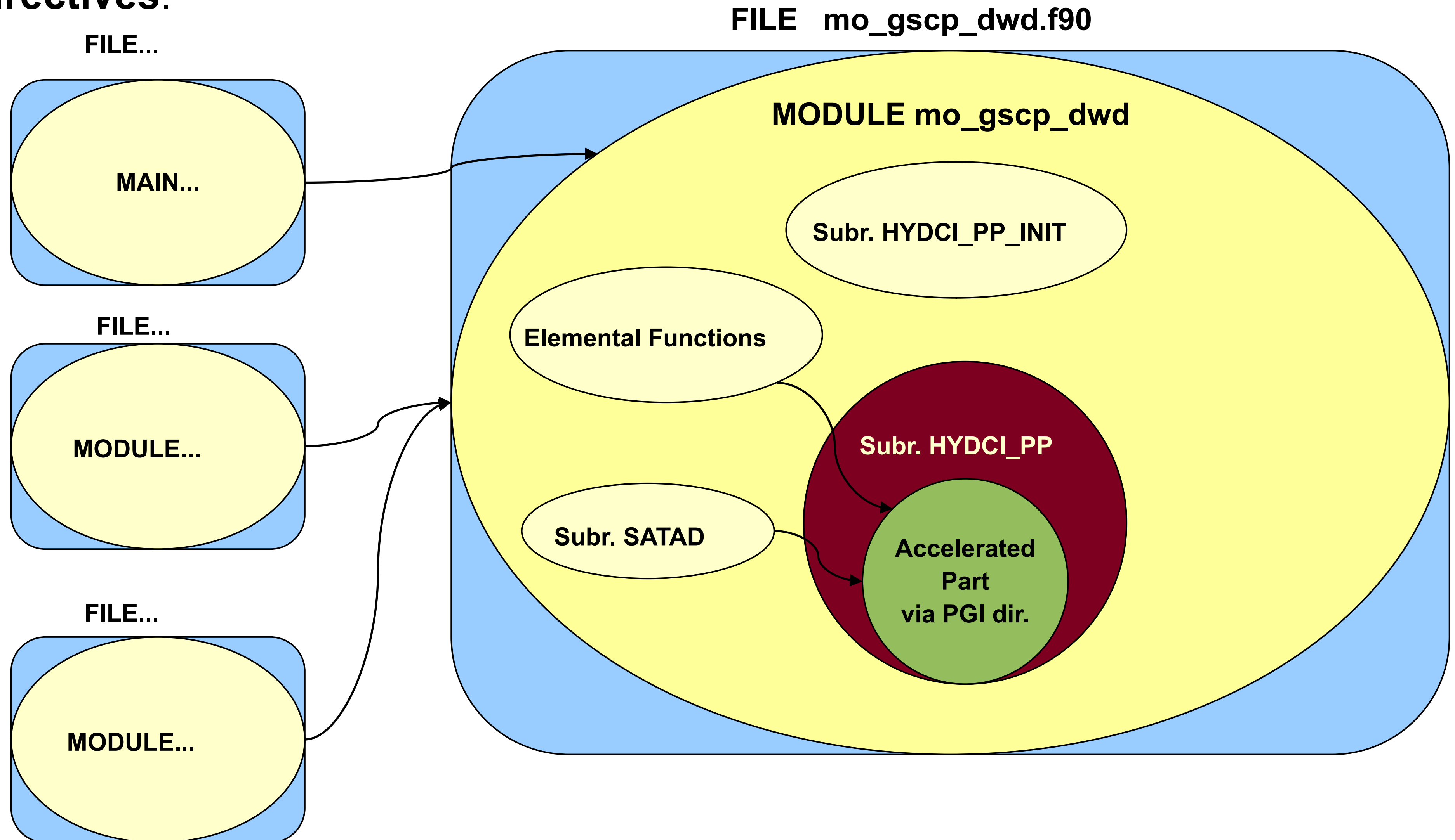


Porting COSMO Microphysics

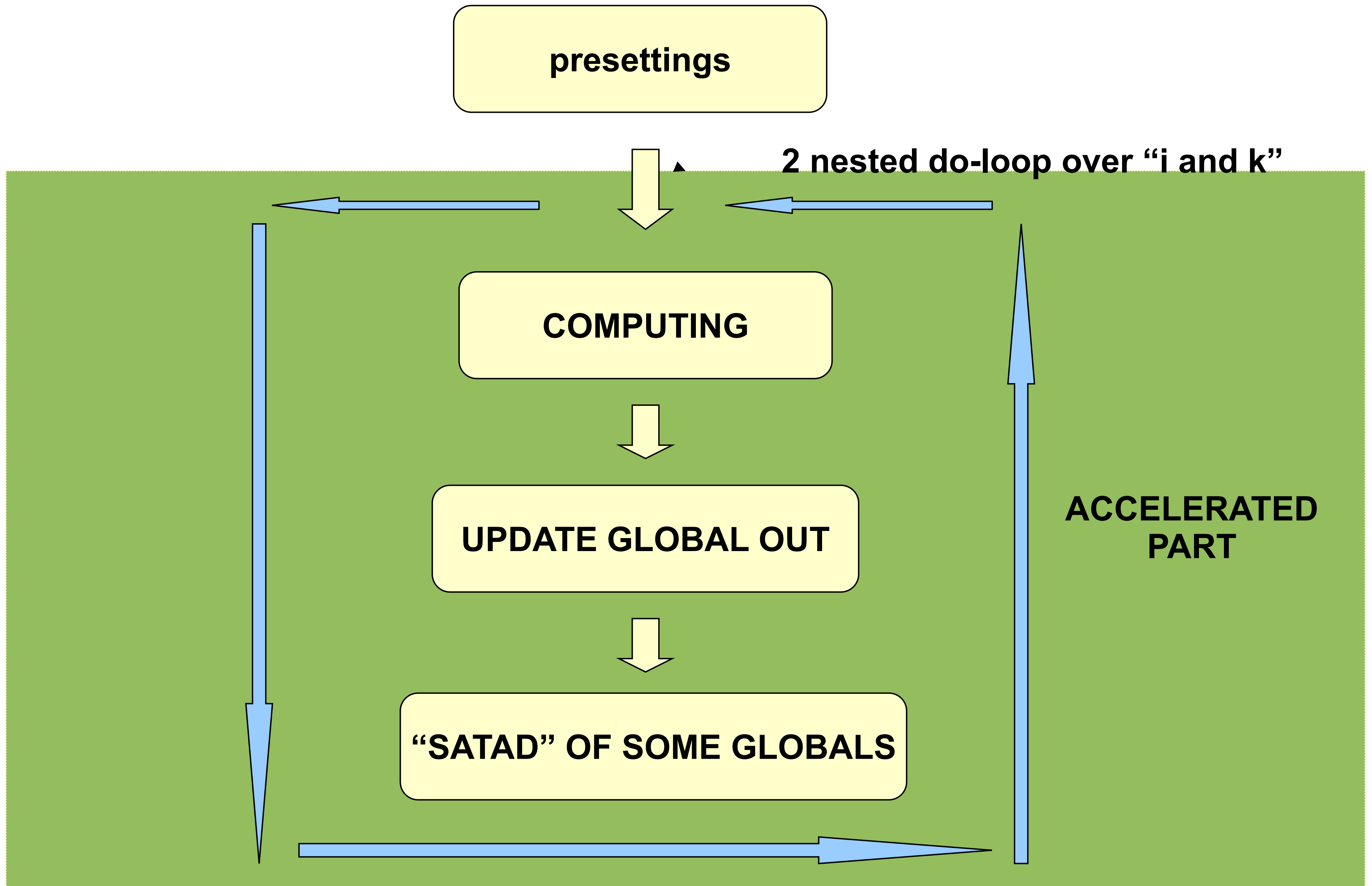
- POMPA COSMO p.p. is exploring “the possibilities of a simple porting of specific physics or dynamics kernels to GPUs”.
- Two different approaches emerged to deal with the problem: one based on **PGI Accelerator** directives and the other one based on the **F2C-ACC** tool.
- The study has been done on the **Microphysics stand alone** program optimized by Xavier Lapillonne for GPU with PGI Accel tool, and referred on the HPCforge site.

Reference Code Structure

In **microphysics** program the two nested do-loop over space inside the subroutine **hydci_pp** has been individuated as the part to be accelerated via **PGI directives**.



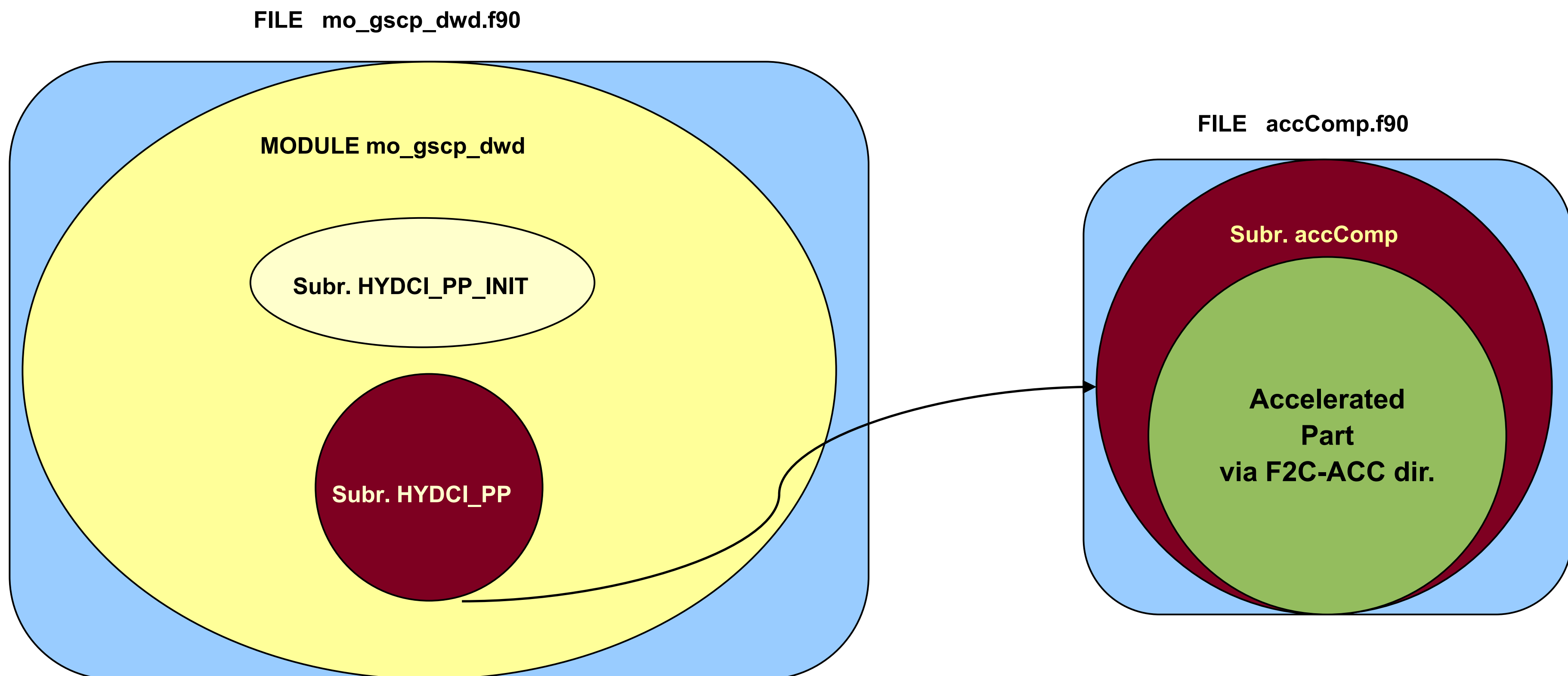
Reference Code



...

Modified Code Structure

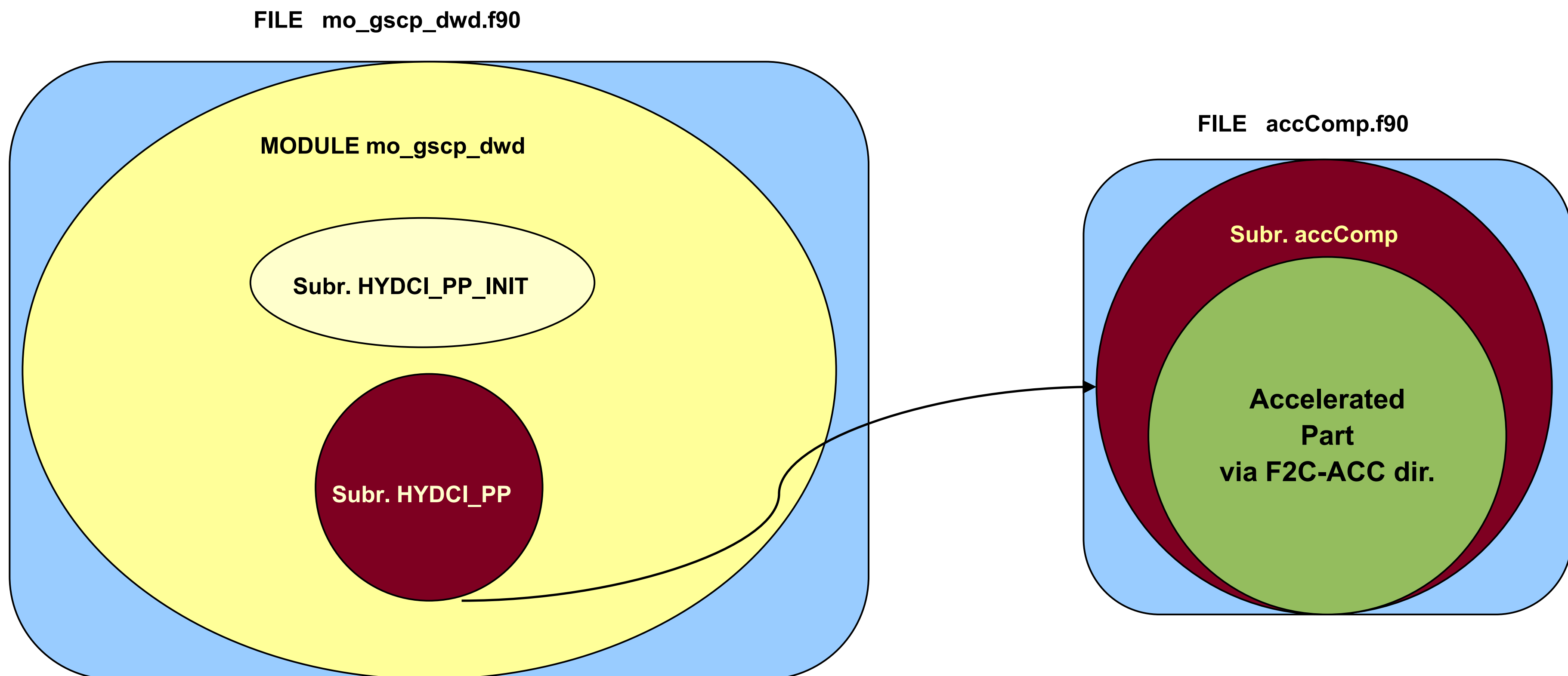
- We proceeded to accelerate **the same part** of the code via **F2C-ACC directives**.
- Due to current release limitations of F2C-ACC **the code structure has been partly modfied**, while the **workflow has been leaved unchanged**.
- The part of the code to be accelerated remain the same but this has been extracted from hydci_pp subroutine and a file apart containing a **new subroutine** as been created for it: **accComp.f90**.



Modified Code Structure: why ?

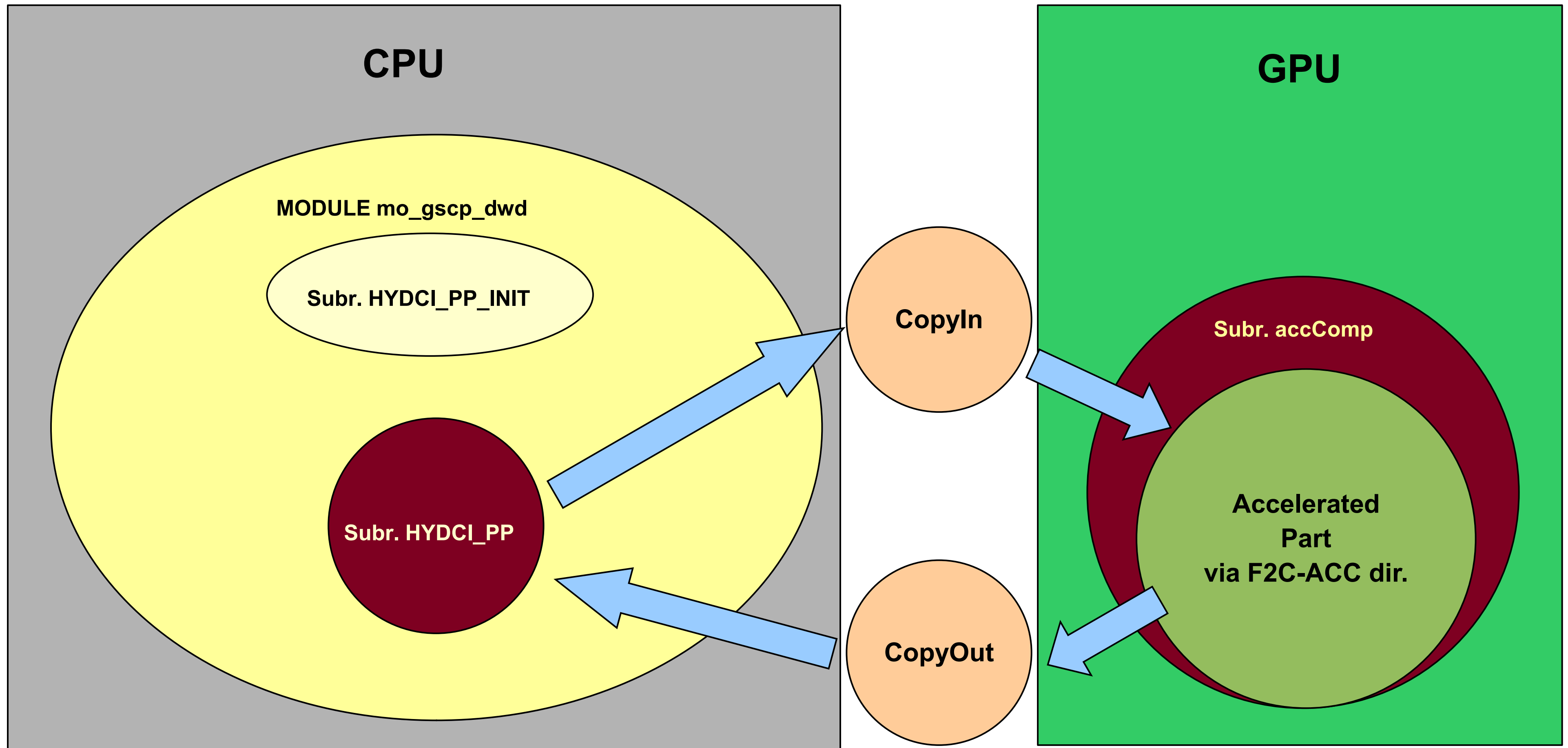
Major limitations have driven the changing in the code are:

- Modules are (for now) not supported → necessary variables passed to the called subroutines and called subroutines/functions included into the file.
- F2C-ACC "--kernel" option isn't carefully tested → elemental functions and subroutines ("satad") inlined.

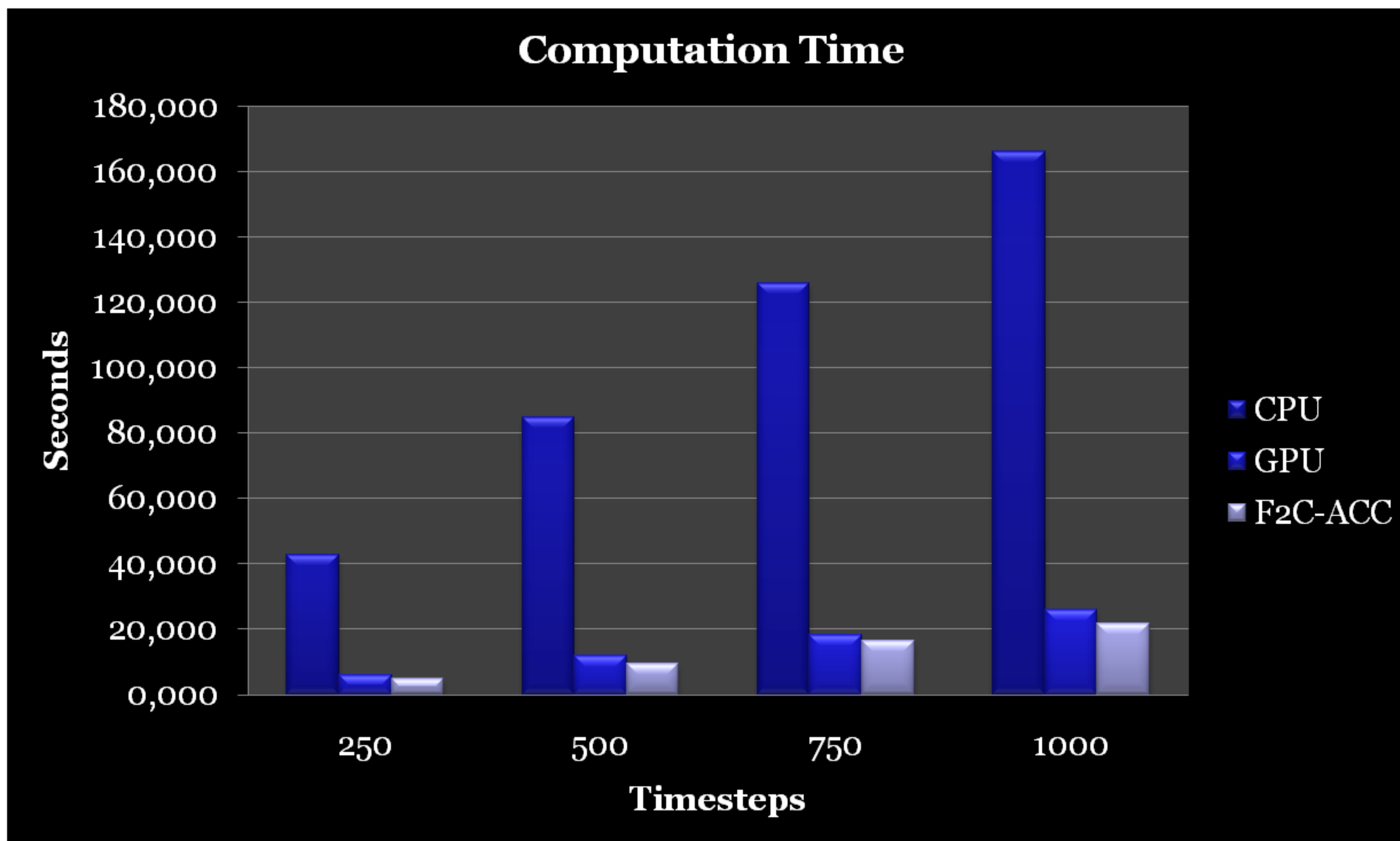


Modified Code Structure

Host / Device View



Preliminary results



Timesteps	250	500	750	1000
CPU	42,685	84,951	125,973	166,206
GPU	5,952	11,819	18,389	26,081
F2C-ACC	4,814	9,634	16,650	21,843

Experience Applying Fortran GPU Compilers to Numerical Weather Prediction

Tom Henderson

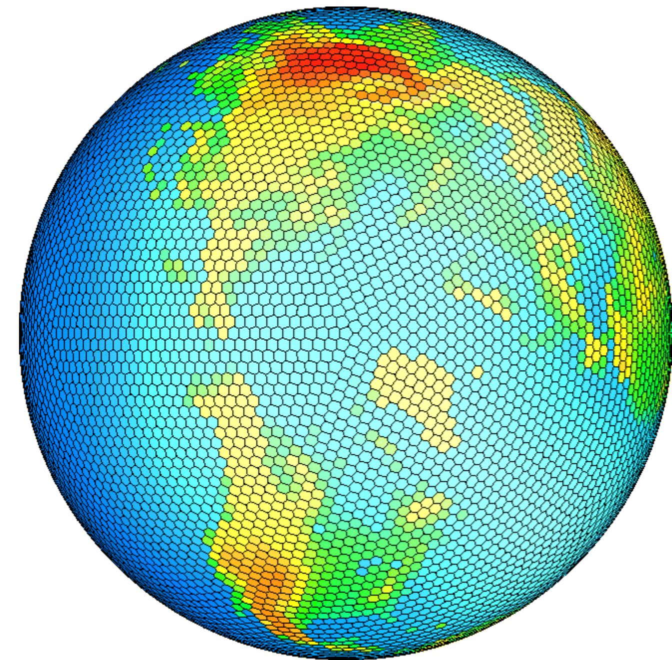
NOAA Global Systems Division

Thomas.B.Henderson@noaa.gov

Mark Govett, Jacques Middlecoff

Paul Madden, James Rosinski,

Craig Tierney



slide courtesy Dr. Tom Henderson NOAA

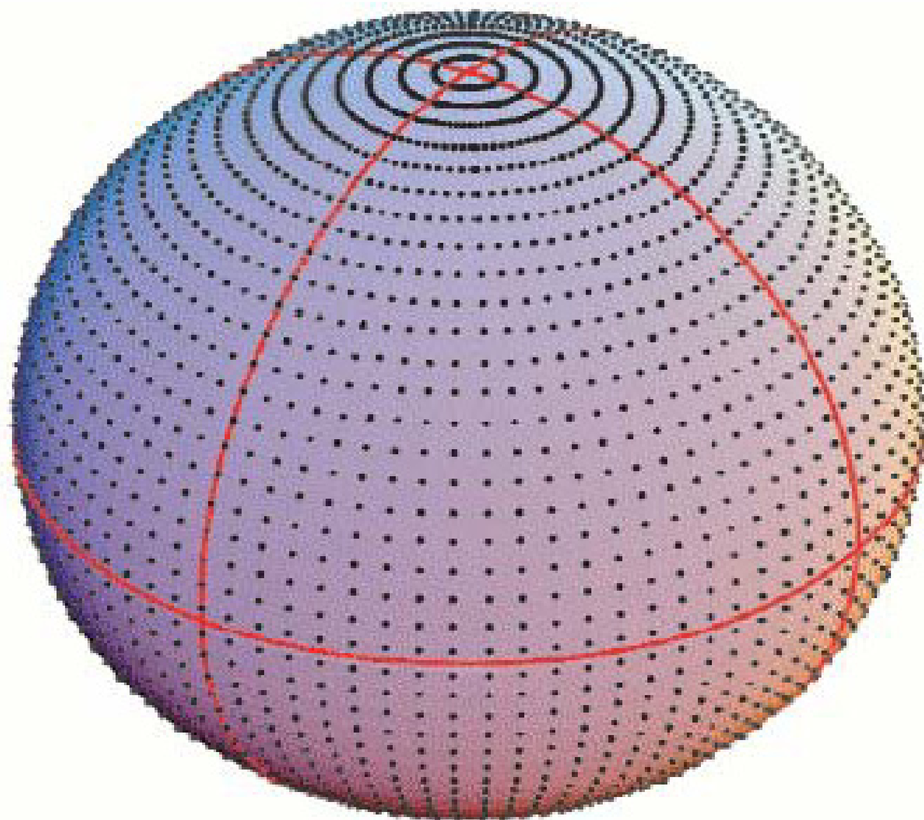
NIM NWP Dynamical Core

- NIM = "Non-Hydrostatic Icosahedral Model"
  - New NWP dynamical core
  - Target: global "cloud-permitting" resolutions ~3km (42 million columns)
  - Rapidly evolving code base
- Single-precision floating-point computations
- Computations structured as simple vector ops with indirect addressing and inner vertical loop
- "GPU-friendly", also good for CPU
- Coarse-grained parallelism via Scalable Modeling System (SMS)
  - Directive-based approach to distributed-memory parallelism

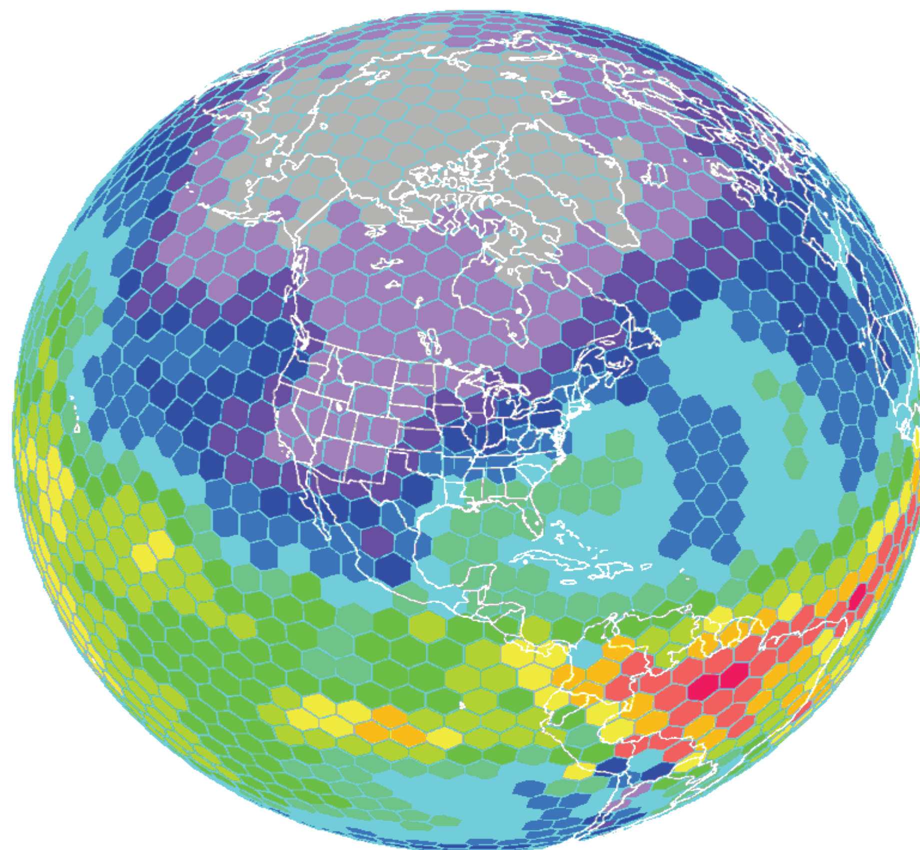
slide courtesy Dr. Tim Henderson NOAA

Icosahedral (Geodesic) Grid: A Soccer Ball on Steroids

Lat/Lon Model



Icosahedral Model



slide courtesy Dr. Tim Henderson NOAA

Early Work With Multi-GPU Runs

- F2C-ACC + SMS directives
  - Identical results using different numbers of GPUs
  - Poor scaling because compute has speed up but communication has not
  - Working on communication optimizations
- **Demonstrates that single source code can be used for single/multiple CPU/GPU runs**
- Should be possible to mix HMPP/PGI directives with SMS too

slide courtesy Dr. Tim Henderson NOAA

Thanks!