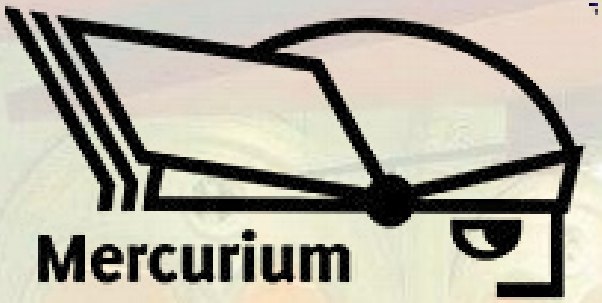


Design Decisions for a Source-2-Source Compiler

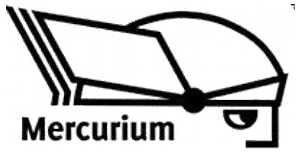
Roger Ferrer, Sara Royuela, Diego Caballero,
Alejandro Duran, **Xavier Martorell** and Eduard Ayguadé
Barcelona Supercomputing Center
and Universitat Politècnica de Catalunya

Cetus Users and Compiler Infrastructures Workshop
in conjunction with PACT'11

Galveston, TX, USA, October 10th, 2011

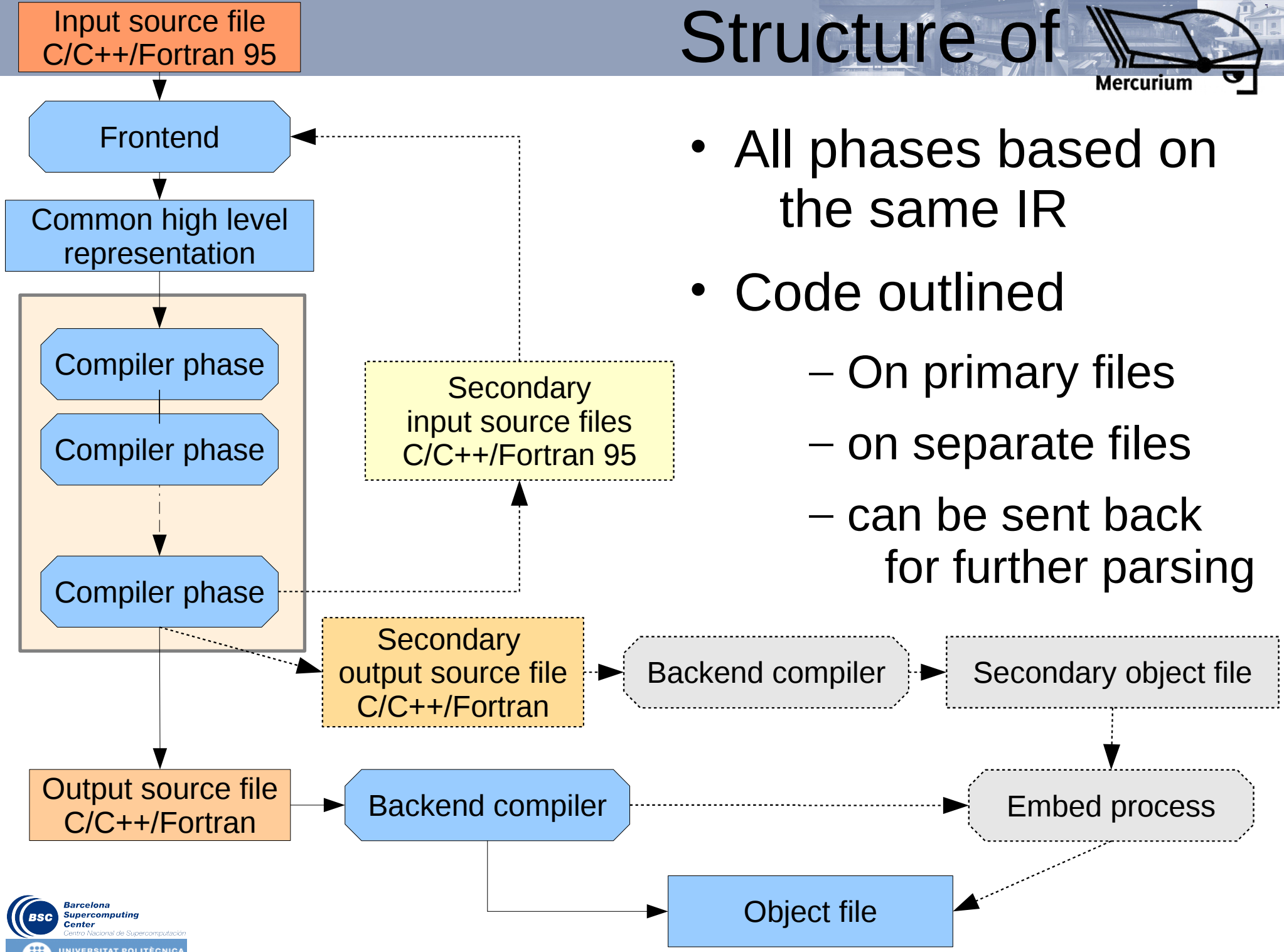
Barcelona Supercomputing Center
Centro Nacional de Supercomputación
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Outline

- **Structure of the compiler**
- **A little of history**
- Design decisions for 

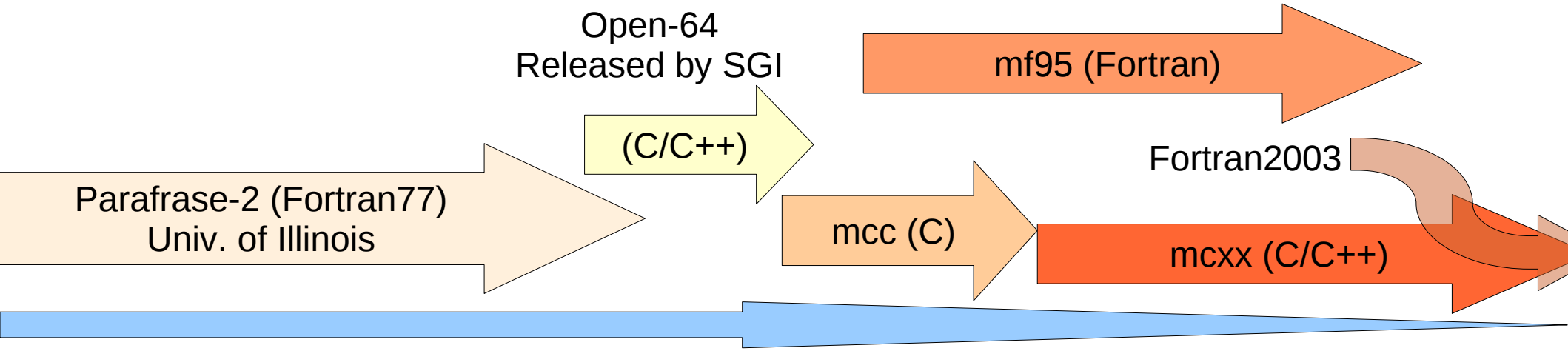
- Developments
- Conclusions and future work

Structure of Mercurium



- All phases based on the same IR
- Code outlined
  - On primary files
  - on separate files
  - can be sent back for further parsing

The Mercurium story



96 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012

NANOS
POP
SARC
ENCORE
INTONE
ACOTES
TERAFLUX
TEXT
MONT BLANC

penMP 1.0 Fortr

OpenMP 1.0 - C/C++

OpenMP 2.0 - Fortr

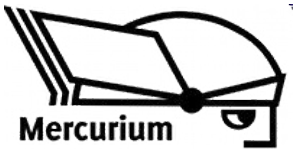
OpenMP 2.0 - C/C++

OpenMP 2.5 - merging Fortran & C/C++

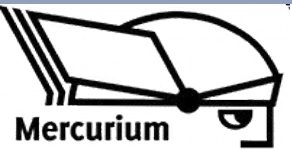
OpenMP 3.0 (tasking)

OpenMP 3.1

Outline

- Structure of the compiler
- A little of history
- **Design decisions for** 

- Developments
- Conclusions and future work



design decisions

- Extended parsing **&&** later disambiguation
- Write source **&&** subparsing
- Generic driver **&&** plug-ins
- Drop-in replacement **&&** driver compatibility
- Have multi-file support **&&** secondary output files
- Common representation **&&** FORTRAN/C/C++

Extended parsing

- Add new language features
  - Types, built-in functions
- Extend directives/pragmas
  - Directive registration
- Directives are broken into tokens and lists of symbols
  - Meaning is built at a later pass
    - **Later disambiguation**

Write source

- Source datatype
  - Embed the code to be generated in your compiler phases

```
Source TL::Nanox::common_parallel_code(const std::string& outline_name,
                                       Source num_threads, ...)
{
   device_provider->do_replacements(data_environ_info, parallel_code, ...);
   device_provider->create_outline(outline_name, struct_arg_type_name, ...);

   result
      << "{"
      <<   "unsigned int _nanos_num_threads = " << num_threads << ";"
      <<   "nanos_team_t _nanos_team = (nanos_team_t)0;"
      <<   "nanos_thread_t _nanos_threads[_nanos_num_threads];"
      <<   "nanos_err_t err;"
      <<   "err = nanos_create_team(&_nanos_team, (nanos_sched_t)0, &_nanos_num_threads,"
      <<            "(nanos_constraint_t*)0, /* reuse_current */ 1, _nanos_threads);"
       ...
}
```

Write source

- “Source” is later parsed
  - Incorporating it to the IR
- Early compiler phase can generate directives for later phases

```
Source new_pragma_construct_src;
new_pragma_construct_src
   << "#line " << construct.get_ast().get_line() << " \""
                << construct.get_ast().get_file() << "\"\n"
   << device_line
         << "#pragma omp task " << clauses << "\n"
         << ";"
         ;
```

- **Subparsing**: multiple parsing starting points in grammar
  - Statement, declaration, function, directive...

Generic driver

- Driver is controlled by a configuration file
  - Allows conditional execution of compiler phases
    - shared libraries

```
...
# if --instrument is given, activate the internal
#                                compiler variable indicating so
{instrument} options = --variable=instr:1

# load the proper compiler plug-in
{instrument} compiler_phase = libtlinstr.so

# and link against the proper (instrumented) libraries
{instrument} linker_options = \
                                -L@NANOX_LIBS@/instrumentation -lnanox
...
```

Compiler plugin

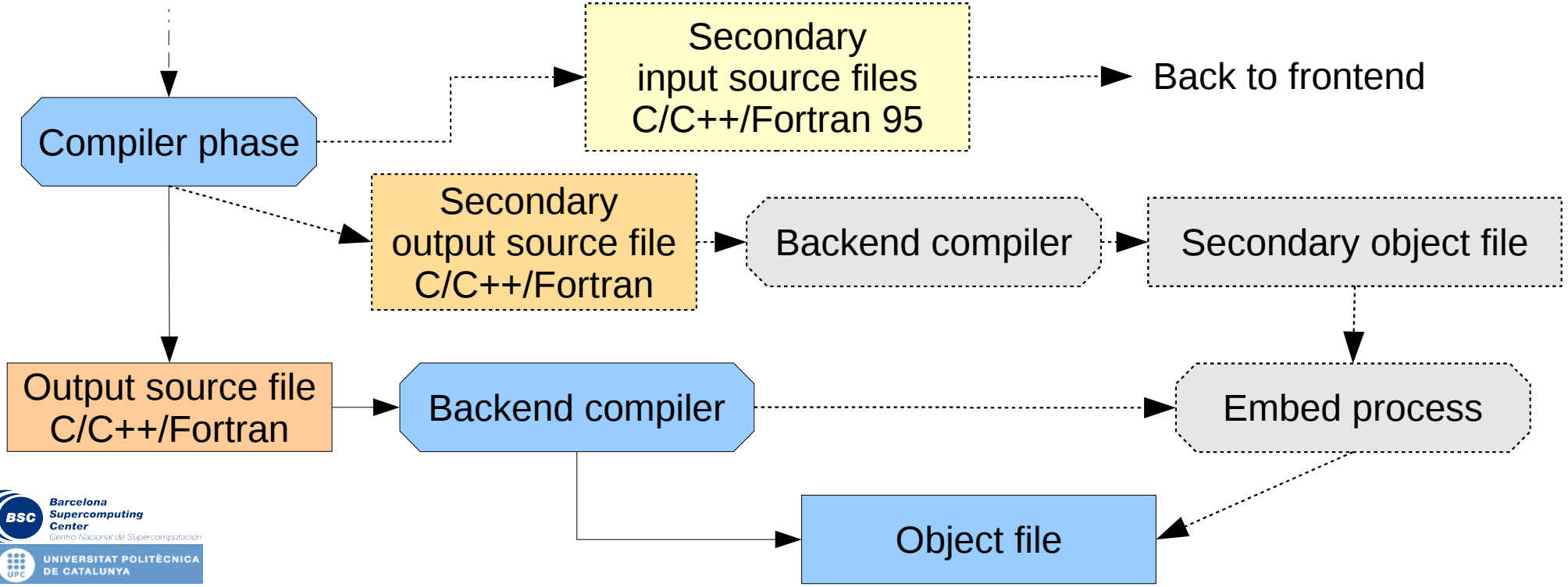
Drop-in replacement

- Autoconf, CMake, Makefiles should work out of the box
- Offer the flavor of just any other compiler
  - Be compatible with the way of invoking GCC
- Functionalities
  - Generate object files, and executable files
  - Link objects into the executable
  - Embed GPU or SPU binaries into the main host executable

Driver compatibility

Have multi-file support

- Portions of code identified as “accelerator code”
  - Are **outlined** to separate files
  - Compiled with the **native accelerator** compiler



Common representation

- Use a common IR among FORTRAN/C/C++

cxx03.y

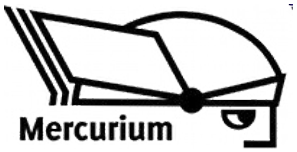
```
FOR '(' for_init_statement condition_opt ';' expression_opt ')' statement
{
    AST loop_control = ASTMake3(AST_LOOP_CONTROL, $3, $4, $6, $1.token_file, $1.token_line, NULL);
   $$ = ASTMake3(AST_FOR_STATEMENT, loop_control, $8, NULL, $1.token_file, $1.token_line, “c++”);
}
```

fortran03.y

```
loop_control : comma_opt do_variable '=' int_expr ',' int_expr comma_int_expr_opt
{
    AST assig = ASTMake2(AST_ASSIGNMENT, $2, $4, ASTFileName($2), ASTLine($2), NULL);
   $$ = ASTMake3(AST_LOOP_CONTROL, assig, $6, $7, ASTFileName($2), ASTLine($2), "fortran");
}

label_do_stmt : labeldef name_colon_opt TOK_DO label loop_control eos
{
    $$ = ASTMake5Label(AST_FOR_STATEMENT, $1, $5, NULL, NULL, $4, $3.token_file,
                                                                  $3.token_line, NULL);
}
```

Outline

- Structure of the compiler
- A little of history
- Design decisions for 

- **Developments**
- Conclusions and future work

Developments

- OpenMP 3.0 tasking
  - tasks, standard since 2008
  - prototypes for taskgroups
    - Not included in the standard
- Prototyping User Defined Reductions (UDRs)
- StarSs and OmpSs: input/output/inout extensions
  - Data dependences among tasks
  - Copy in/out to/from accelerator memories

OmpSs: OpenMP+StarSs

- Objective
  - New set of annotations for data dependence analysis and movement

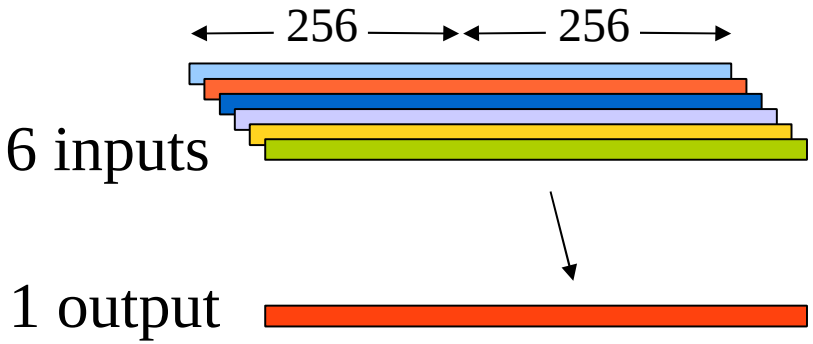
```
float a[N];        // want them copied to accelerator
float b[N];        // as automatically as possible
float c[N];

for (i=0; i<N; i++) {
#pragma omp target device(cuda) copy_deps
#pragma omp task input (a, b) output (c)
  {
    c[i] = a[i] + b[i];     // want it run in the accelerator
  }
}
```

OmpSs: OpenMP+StarSs

- BlackScholes annotated

```
for (i=0; i<array_size; i+=local_work_group_size*vector_width) {
  int limit = ((i+local_work_group_size)>array_size) ?
                    array_size - i : local_work_group_size;
  uint * cpflag_f = &cpflag_fptr[i];
  float * S0_f    = &S0_fptr[i];
  float * K_f     = &K_fptr[i];
  float * r_f     = &r_fptr[i];
  float * sigma_f = &sigma_fptr[i];
  float * T_f     = &T_fptr[i];
  float * answer_f = &answer_fptr[i];
```

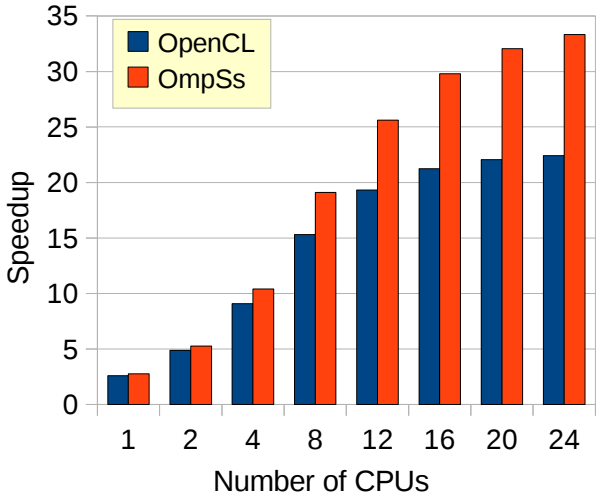



```
#pragma omp target device(cuda) copy_deps
#pragma omp task shared(cpflag_f,S0_f,K_f,r_f,sigma_f,T_f,answer_f) \
                 input ( \
                       [global_work_group_size]   cpflag_f, \
                       [global_work_group_size]   S0_f, \
                       [global_work_group_size]   K_f, \
                       [global_work_group_size]   r_f, \
                       [global_work_group_size]   sigma_f, \
                       [global_work_group_size]   T_f) \
                 output ([global_work_group_size]   answer_f)
    {
        // kernel code
    }
}
```

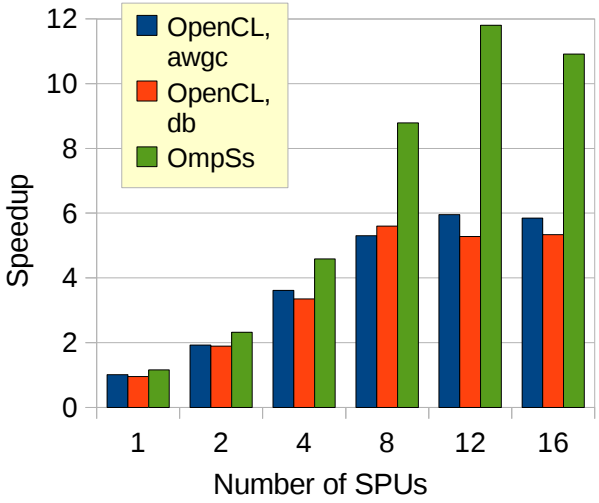
OmpSs: OpenMP+StarSs

- BlackScholes evaluation
  - 1.4x in SMP
  - 2x performance increase in Cell/B.E. vs. OpenCL
  - Equivalent in GTX 285, no scaling due to inputs

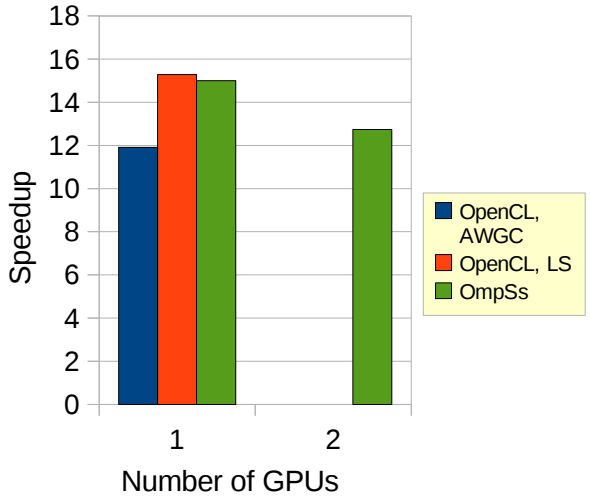
Intel Xeon server



Cell processor



Nvidia GPU GTX 285

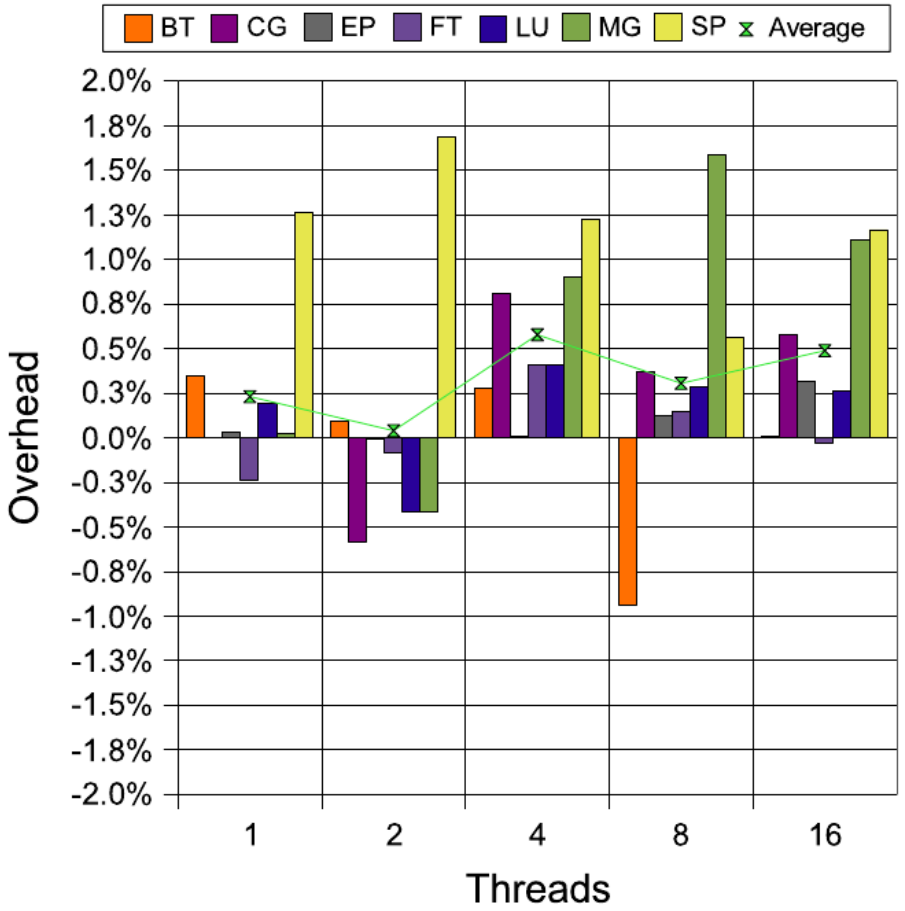


Error handling for OpenMP

- Generic error management and user error handlers

```
#pragma omp parallel onerror(OMP_SEVERE_ERROR : OMP_ABORT, \
                             OMP_MEDIUM_ERROR : my_error_handler, arg)
{
  // parallel region
}
```

- Evaluated with the NAS benchmarks



User-driven vectorization

- High-level transformation directive indicating...
  - A loop is vectorizable
    - and the variables that should be vectorized
  - A function is vectorizable
  - Still, user needs to take care of proper alignment!

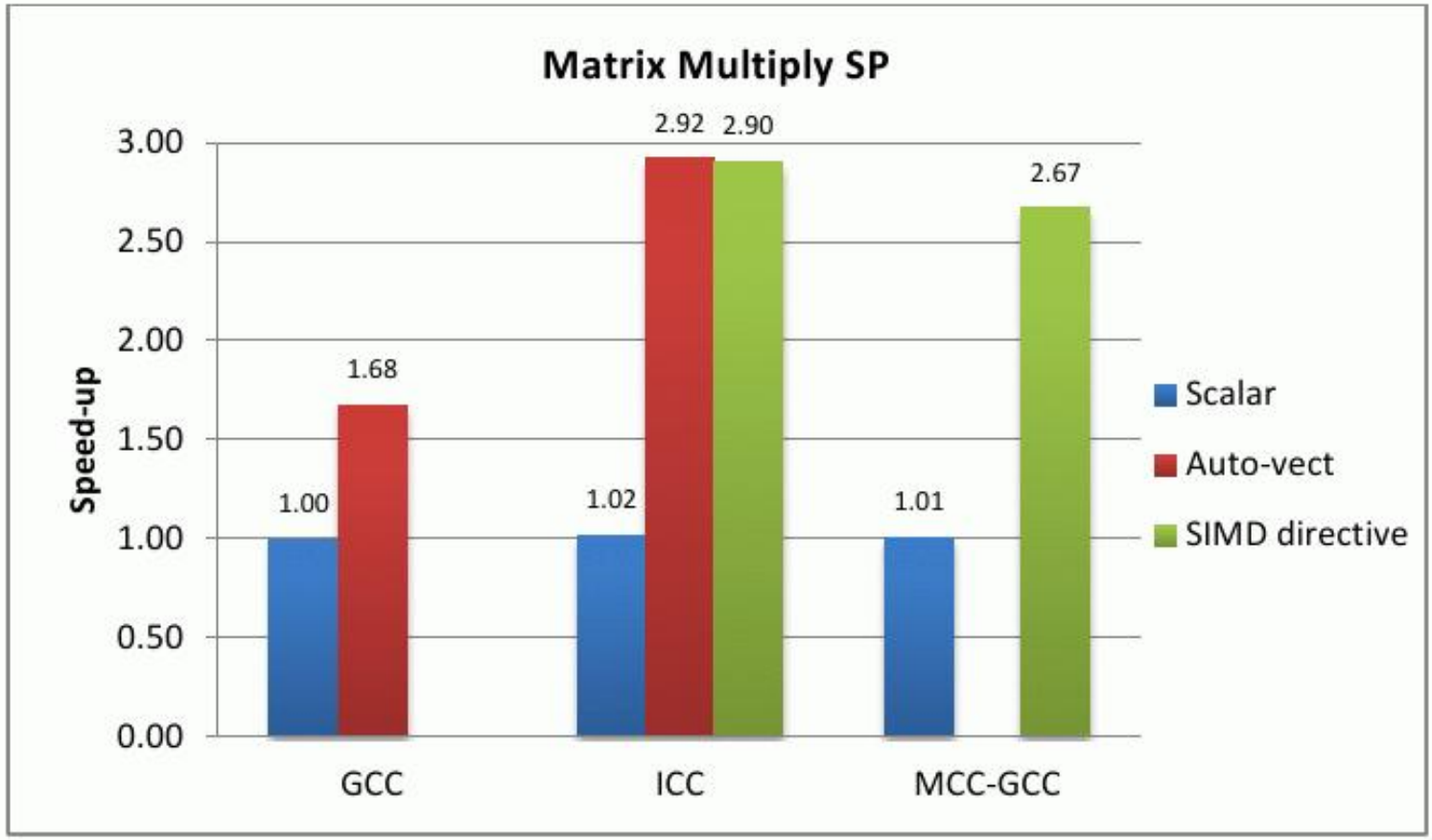
```
#pragma hlt simd
float myfunc(float X)
{
...
}
```

```
void main(int args, char * argv[])
{
float a [N], b[N], c[N];
...
#pragma hlt simd(a, b, c)
   for (i=0; i < N; i++)
   {
      c[i] = a[i] + b[i] + myfunc(a[i]);
   }
...
}
```

Inspired on the **Intel SIMD directive** available in ICC

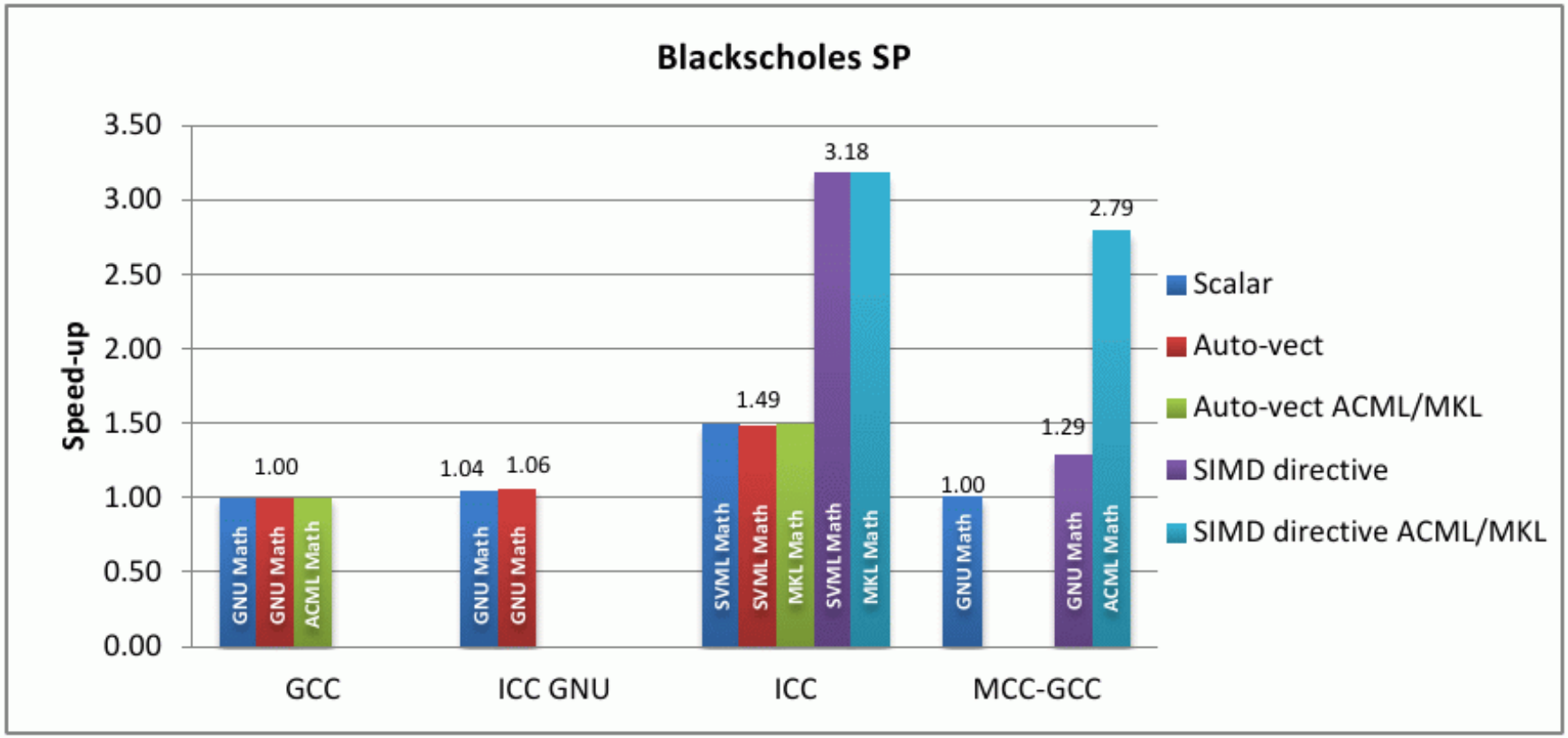
User-driven vectorization

- Results comparable to Intel ICC auto and SIMD



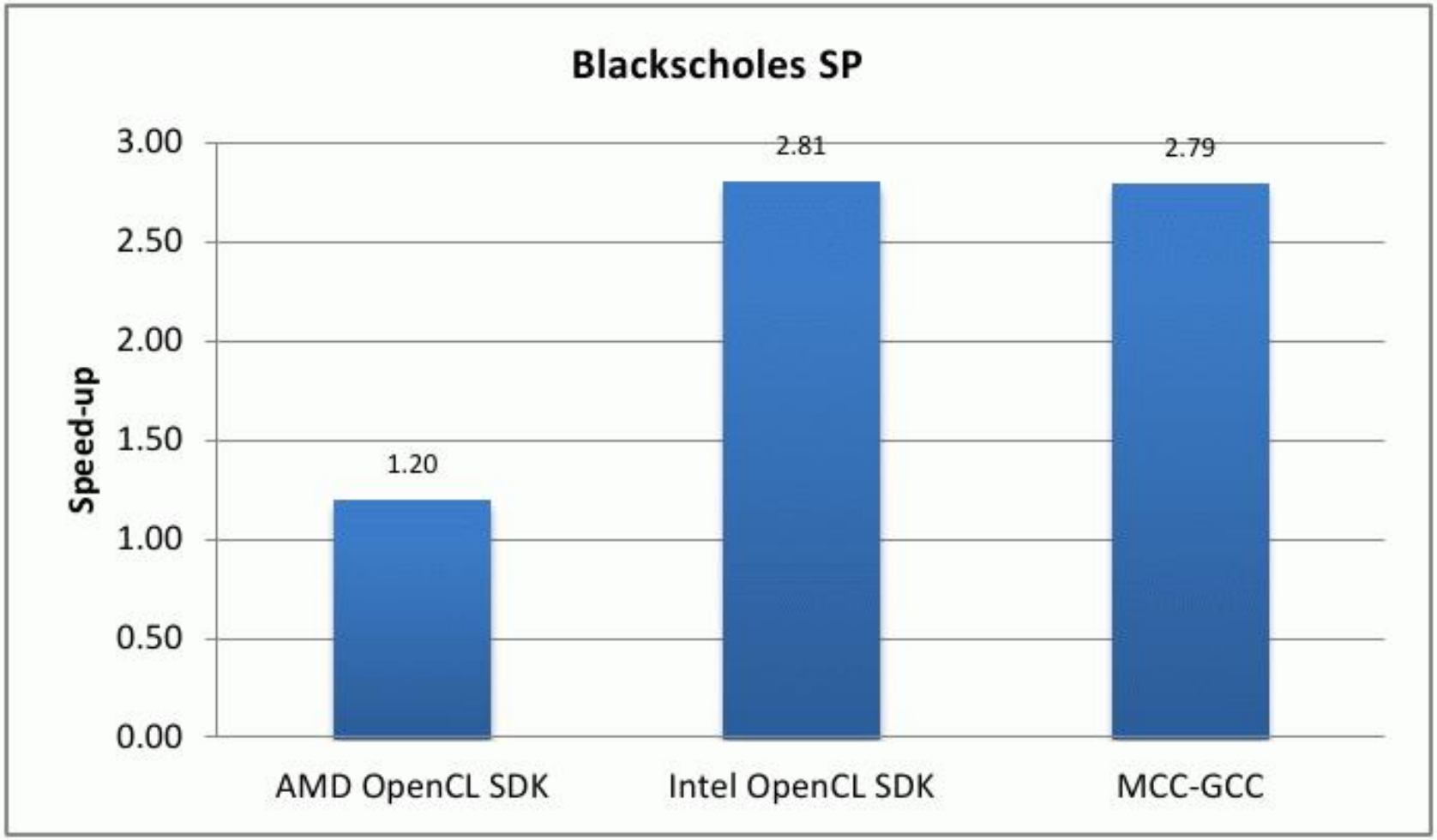
User-driven vectorization

- Results comparable to Intel ICC SIMD with MKL

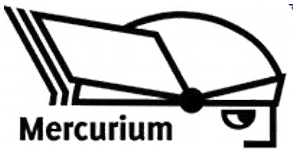


User-driven vectorization

- Results comparable to Intel OpenCL



Outline

- Structure of the compiler
- A little of history
- Design decisions for 

- Developments
- **Conclusions and future work**

Conclusions

- Shown the Mercurium compiler infrastructure
  - C/C++ and Fortran
  - Flexible to be adapted to lots of projects
  - Useful and productive for directive-based program transformations
  - Mostly compatible with existing compilers
  - Support for heterogeneity and local memories

Future work

- Fortran support
  - Recognize directives and generate code
- Analysis phases
  - Currently, life analysis for symbols
- Code inlining, interprocedural analysis
- Prototyping of new OpenMP features

Acknowledgments

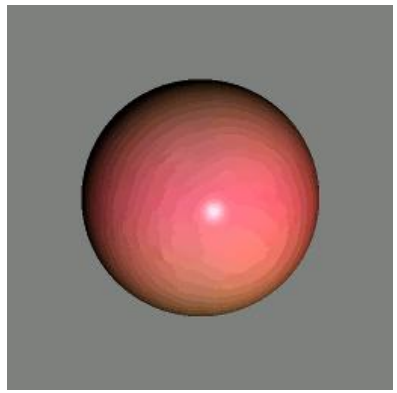
- Parallel programming group @BSC
- Encore, TEXT, TERAFLUX, MONTBLANC
  - European Commission

Mercurium

available at

Barcelona Supercomputing Center

http://pm.bsc.es



Barcelona Supercomputing Center
Centro Nacional de Supercomputación
UNIVERSITAT POLITÈCNICA DE CATALUNYA