PIPS

An Interprocedural, Extensible, Source-to-Source Compiler Infrastructure for Code Transformations and Instrumentations

International Symposium on Code Generation and Optimization
CGO 2011

Corinne Ancourt, Frederique Chaussumier-Silber, Serge Guelton, Ronan Keryell

For the most recent version of these slides, see:
http://www.pips4u.org

Last edited:
April 2, 2011

Whom is this Tutorial for?

- **This tutorial is relevant to people interested in:**
  - GPU or FPGA-based, hardware accelerators, manycores,
  - Quickly developing a compiler for an exotic processor (Larrabee, CEA SCMP...),
  - And more generally to all people interested in experimenting with new program transformations, verifications and/or instrumentations.

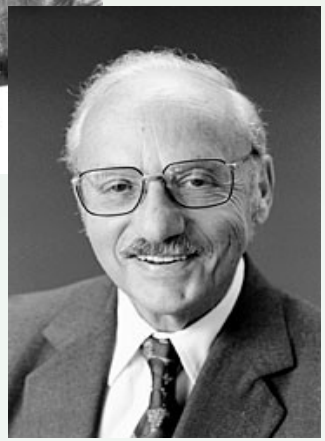
- **This tutorial aims:**
  - To illustrate usage of PIPS analyses and transformations in an interactive demo
  - To give hints on how to implement passes in Pips
  - To survey the functionalities available in PIPS
  - To introduce a few ongoing projects. Code generation for
    - Streaming SIMD Extensions
    - Distributed memory machines: STEP
  - To present the Par4All plateform based on PIPS

Once upon a Time...

- **1823:** J.B.J. Fourier, « Analyse des travaux de l'Académie Royale des Sciences pendant l'année 1823 »

- **1936:** Theodor Motzkin, « Beiträge zur Theorie der linearen Ungleichungen »

- **1947:** George Dantzig, Simplex Algorithm



- Linear Programming, Integer Linear Programming

$$\exists ?\ Q \text{ s.t. } \{x|\ \exists\ y\ P(x,y)\} = \{x|Q(x)\}$$

Once upon a Time...

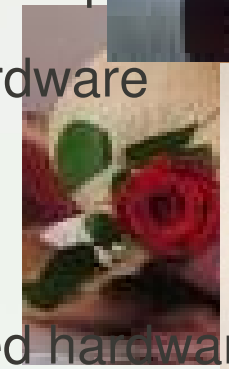
- **1984:** Rémi Triolet, interprocedural parallelization, convex array regions
- **1987:** François Irigoin, tiling, control code generation
- **1988:** PIPS begins...
- **1991:** Corinne Ancourt, code generation for data communication
- **1993:** Yi-qing Yang, dependence abstractions
- **1994:** Lei Zhou, execution cost models
- **1996:** Arnauld Leservot, Presburger arithmetic
- **1996:** Fabien Coelho, HPF compiler, distributed code generation
- **1996:** Béatrice Creusillet, must/exact regions, in and out regions, array privatization, coarse grain parallelization
- **1999:** Julien Zory, expression optimization

Ten years ago...
Why do we need this today?
→ Heterogeneous computing!

~~In the West~~ In France...

- **2002:** Nga Nguyen, array bound check, alias analysis, variable initialization
- **2002:** Youcef Bouchebaba, tiling, fusion and array reallocation
- **2003:** C parser, MMX vectorizer, VHDL code generation
- **2004:** STEP Project: OpenMP to MPI translation
- **2005:** Ter@ops Project: XML code modelization, interactive compilation
- **2006:** CoMap Project, code generation for programmable hardware accelerator
- **2007:** HPC Project startup is born
- **2008:** FREIA Project: heterogeneous computing, FPGA-based hardware accelerators
- **2009:** Par4All initiative + Ronan Keryell: CUDA code generation
- **2010:** OpenGPU Project: CUDA and OpenCL code generation

  SCALOPES, MediaGPU, SMECY, SIMILAN, ...



What is PIPS?

- **Source-to-source Fortran and C compiler, written in C**
  - Maintained by MINES ParisTech, TELECOM Bretagne / SudParis and HPC Project
- **Includes free Flex/Bison-based parsers for C and Fortran**
- **Internal representation with powerful iterators (30K lines)**
- **Compiler passes (300K+ lines and growing)**
  - Static interprocedural analyses
  - Code transformations
  - Instrumentations (dynamic analyses)
  - Source code generation
- **Main drivers of the PIPS effort:**
  - Automatic interprocedural parallelization
  - Code safety
  - Heterogeneous computing

Teams Currently Involved in PIPS

- **MINES ParisTech (Fontainebleau, France)**
  - Mehdi Amini, Corinne Ancourt, Fabien Coelho, Laurent Daverio, Dounia Khaldi, François Irigoin, Pierre Jouvelot, Amira Mensi, Maria Szymczak
- **TELECOM Bretagne (Brest, France)**
  - Stéphanie Even, Serge Guelton, Adrien Guinet, Sébastien Martinez, Grégoire Payen
- **TELECOM SudParis (Evry, France)**
  - Rachid Habel, Alain Muller, Frédérique Silber-Chaussumier
- **HPC Project (Paris, France)**
  - Mehdi Amini, Béatrice Creusillet, Johan Gall, Onil Goubier, Ronan Keryell, Francois-Xavier Pasquier, Raphaël Roosz, Pierre Villalon

Past contributors: CEA, ENS Cachan,...

Why PIPS? (1/2)

- **A source-to-source interprocedural translator, because:**
  - Parallelization techniques tend to be source transformations
  - Outputs of all optimization and compilation steps, can be expressed in C
  - Allows comparison of original and transformed codes, easy tracing and IR debugging
  - Instrumentation is easy, as well as transformation combinations.
- **Some alternatives:**
  - Polaris, SUIF: not maintained any longer
  - GCC has no source-to-source capability; entrance cost; low-level SSA internal representation.
  - Open64's 5 IRs are more complex than we needed
  - PoCC (INRIA)
  - CETUS (Purdue), OSCAR (Waseda), Rose (LLNL)...
  - LLVM (Urbana-Champaign)

Why PIPS? (2/2)

- **A new compiler framework written in a modern language?**
  - High-level Programming
  - Standard library
  - Easy embedding and extension
- **Or a time-proven, feature-rich, existing Fortran and C framework?**
  - Inherit lots of static and dynamic analyses, transformations, code generations
  - Designed as a framework, easy to extend
  - Static and dynamic typing to offer powerful iterators
  - Global interprocedural consistence between analyses and transformations
  - Persistence and Python binding for more extensibility
  - Script and window-based user interfaces

→ Best alternative is to reuse existing time-proven software!

Download and License

I.0.10

- **PIPS is free software**
  - Distributed under the terms of the GNU Public License (GPL) v3+.
- **It is available primarily in source form**
  - http://pips4u.org/getting-pips
  - PIPS has been compiled and run under several kinds of Unix-like (Solaris, Linux).
  - Currently, the preferred environment is **amd64 GNU/Linux**.
  - To facilitate installation, a **setup script** is provided to automatically check and/or fetch required dependencies (eg. the Linear and Newgen libraries)
  - Support is available via irc, e-mail and a Trac site.
- **Unofficial Debian GNU/Linux packages**
  - Source and binary packages for Debian Sid (unstable) on x86 and amd64: http://ridee.enstb.org/debian/info.html
  - Tar.gz snapshots are built (and checked) nightly

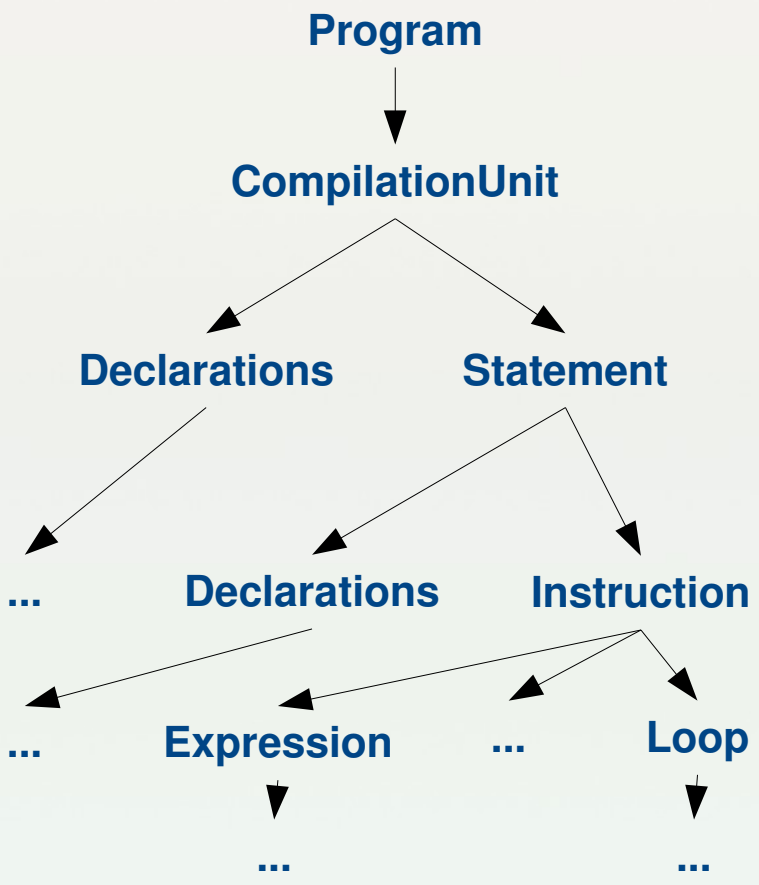
A First Example: Source-to-Source Compilation


```
int
main (void)
{
  int i,j,c,a[100];

  c = 2;
  /* a simple parallel loop */
  for (i = 0;i<100;i++)
    {
      a[i] = c*a[i]+(a[i]-1);
    }
}
```

```
delete intro_example01
create intro_example01 \
    intro_example01.c

apply UNSPLIT

close
quit
```

Explicit destruction of workspace

```
int main(void)
{
   int i, j, c, a[100];

   c = 2;
   /* a simple parallel loop */
   for(i = 0; i <= 99; i += 1)
      a[i] = c*a[i]+a[i]-1;
}
```



Simple Tree-Based IR
As closely associated with original program structure as possible for regeneration of source code

Source-to-Source Parallelization

I.0.12

```
int foo(void)
{
  int i;
  double t, s=0., a[100];
  for (i=0; i<50; ++i) {
    t = a[i];
    a[i+50] = t + (a[i]+a[i+50])/2.0;
    s = s + 2 * a[i];
  }
  return s;
}
```

```
delete intro_example02
create intro_example02 intro_example02.c

setproperty PRETTYPRINT_SEQUENTIAL_STYLE "do"

apply PRIVATIZE_MODULE[foo]
apply INTERNALIZE_PARALLEL_CODE
apply OMPIFY_CODE[foo]
display PRINTED_FILE[foo]

quit
```

```
int foo(void)
{
   int i;
   double t, s = 0., a[100];
#pragma omp parallel for private(t)
   for(i = 0; i <= 49; i += 1) {
      t = a[i];
      a[i+50] = t+(a[i]+a[i+50])/2.0;
   }
#pragma omp parallel for reduction(+:s)
   for(i = 0; i <= 49; i += 1)
      s = s+2*a[i];
   return s;
}
```

Oops, low level.
Encapsulation needed!

Q: Garbage Out? A: Garbage In!


```
int foo(void)
{
   int i;
   double t, s, a[100];
#pragma omp parallel for private(t)
   for(i = 0; i <= 49; i += 1) {
      t = a[i];
      a[i+50] = t+(a[i]+a[i+50])/2.0;
   }
#pragma omp parallel for private(s)
   for(i = 0; i <= 49; i += 1)
      s = s+2*a[i];
   return 0;
}
```

private(s)?

```
int foo(void)
{
  int i;
  double t, s, a[100];
  for (i=0; i<50; ++i) {
    t = a[i];
    a[i+50] = t + (a[i]+a[i+50])/2.0;
    s = s + 2 * a[i];
  }
  return 0;
}
```

```
int foo(void)
{
   return 0;
}
```

Example: Array Bound Checking


```
real function sum(n, a)
real s, a(100)
s  = 0.
do i = 1, n
  s = s + 2. * a(i)
enddo
sum = s
end
```

```
delete intro_example03
create intro_example03 intro_example03.f

setproperty PRETTYPRINT_STATEMENT_NUMBER FALSE
activate MUST_REGIONS

apply ARRAY_BOUND_CHECK_TOP_DOWN
apply UNSPLIT

close
quit
```

or: ARRAY_BOUND_CHECK_BOTTOM_UP

```
!!
!! file for intro_example03.f
!!
      REAL FUNCTION SUM(N, A)
      REAL S, A(100)
      IF (101.LE.N) STOP 'Bound violation:, READING,  array SUM:A, upper
     & bound, 1st dimension'
      S = 0.
      DO I = 1, N
        S = S+2.*A(I)
      ENDDO
      SUM = S
      END
```

Test hoisted out of the loop

User Interfaces

I.0.15

- **Scripting:**
  - tpips: standard interface, used in previous examples
  - ipyps: Python-powered interactive shell
- **Shell command:**
  - pipscc
  - Pips, Init, Display, Delete,...
- **GUI:**
  - paws: under development
  - wpips, epips, jpips, gpips: not useful for real work
- **Programming + Scripting:**
  - PyPS: API to build new compilers, e.g. used in `p4a`

Scripting PIPS: tpips

I.0.16

- **tpips can be interactive or scripted**
- **With tpips, you can:**
  - Manage workspaces
    - create, delete, open, close
  - Set properties
  - Activate rules
  - Apply transformations
  - Display resources
  - Execute shell commands
  - ...

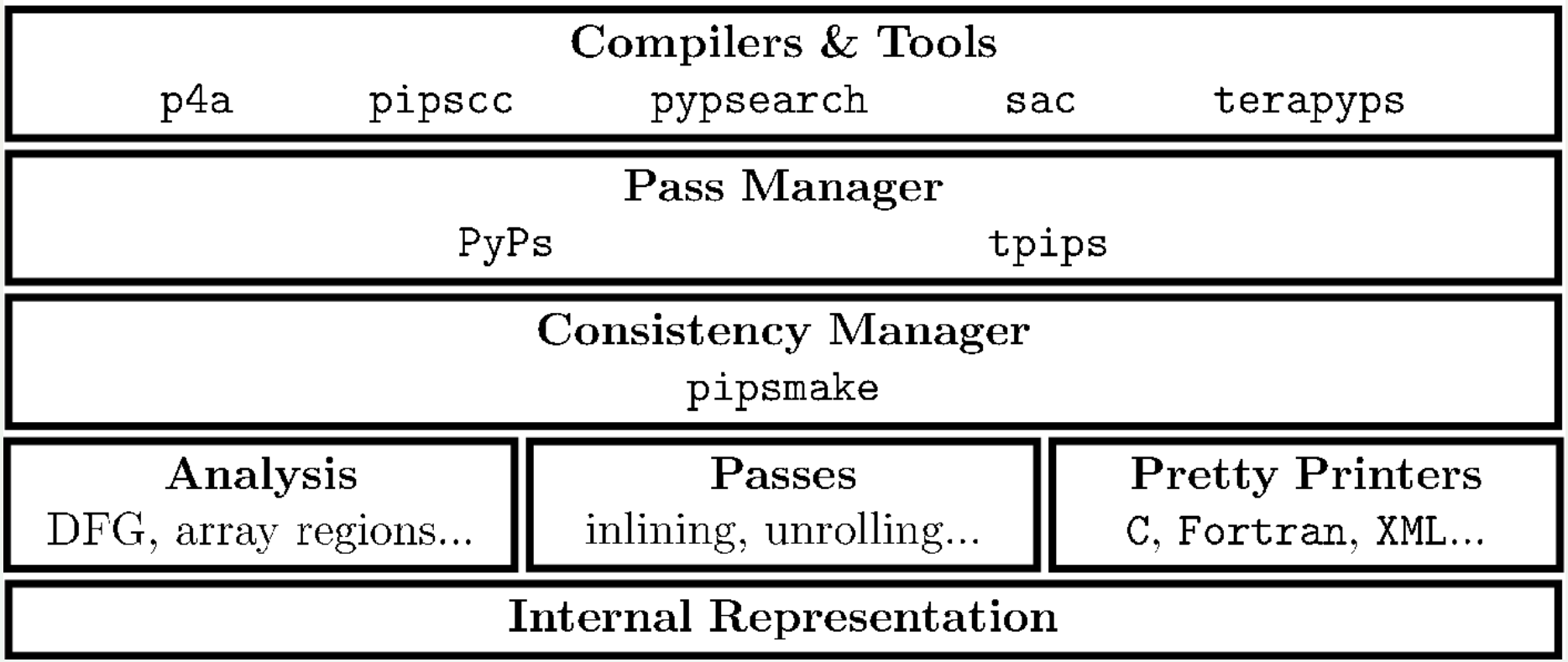
```
delete intro_example03
create intro_example03 intro_example03.f

setproperty PRETTYPRINT_STATEMENT_NUMBER FALSE
activate MUST_REGIONS

apply ARRAY_BOUND_CHECK_TOP_DOWN
apply UNSPLIT

sh cat intro_example03.database/Src/intro_example03.f

close
quit
```

- **All internal pieces of information can be displayed**
- **tpips User Manual:**
  - See http://pips4u.org/doc/manuals (HTML or PDF)

II. Diving into Pips: from Python to C

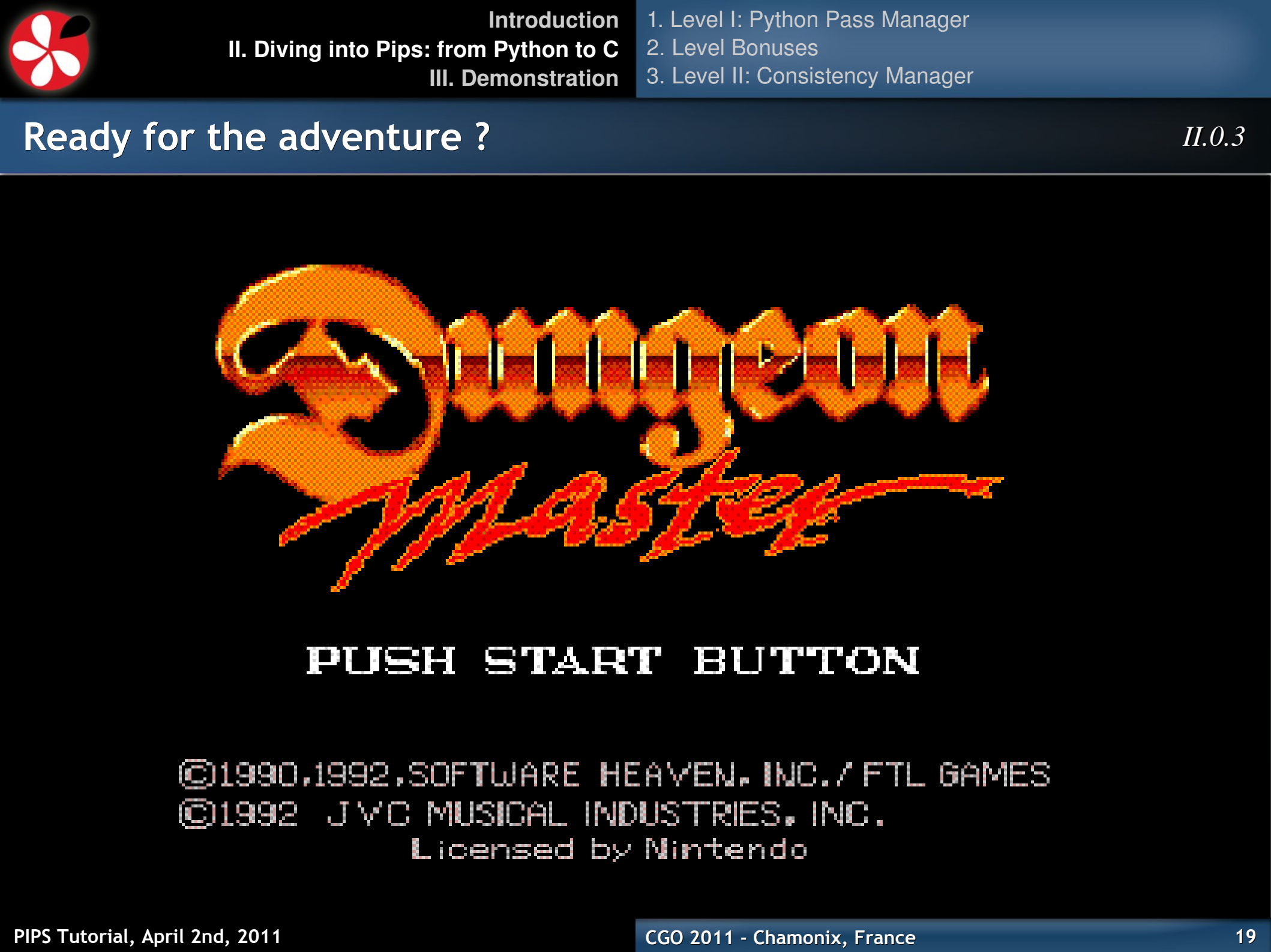
II. Diving into Pips: from Python to C

Pips Overview



Ready for the adventure ?

II.0.3



Choose your weapons !

II.0.4

Level I: ~~kill rats~~ Python Pass Manager

- **Goals:**
  - Make Pass Manager more flexible (python > shell)
  - Develop generic modules (no hard-coded values, enforce resuability)
  - Easier high-level extensions to PIPS using high-level modules
- **Why Python?**
  - Scripting language, Natural syntax
  - Rich ecosystem
  - Easy C binding using swig
- **Be nice with new developers!** **(Plenty of pythonic tasks)**
  - ipython integration
  - PyPS As a Web Service (PAWS)
- **Attract (lure?) users!**
  - Combine transformations easily
  - Develop high-level tools based on PIPS

Pass Manager Example


```
from pyps import *
import re

launcher_re = re.compile("^p4a_kernel_launcher_.*")
def launcher_filter(module):
        return launcher_re.match(module.name)

w = workspace("jacobi.c","p4a_stubs.c",deleteOnClose=True)

w.all.loop_normalize(one_increment=True,lower_bound=0,skip_index_s
ide_effect=True)
w.all.privatize_module()

w.all.display(activate=module.print_code_regions)

w.all.coarse_grain_parallelization()
w.all.display()

w.all.gpu_ify()

# select only some modules from the workspace
launchers=w.all(launcher_filter)
# manipulate them as first level objects
launchers.kernel_load_store()
launchers.display()

launchers.gpu_loop_nest_annotate()
launchers.inlining()
...
```

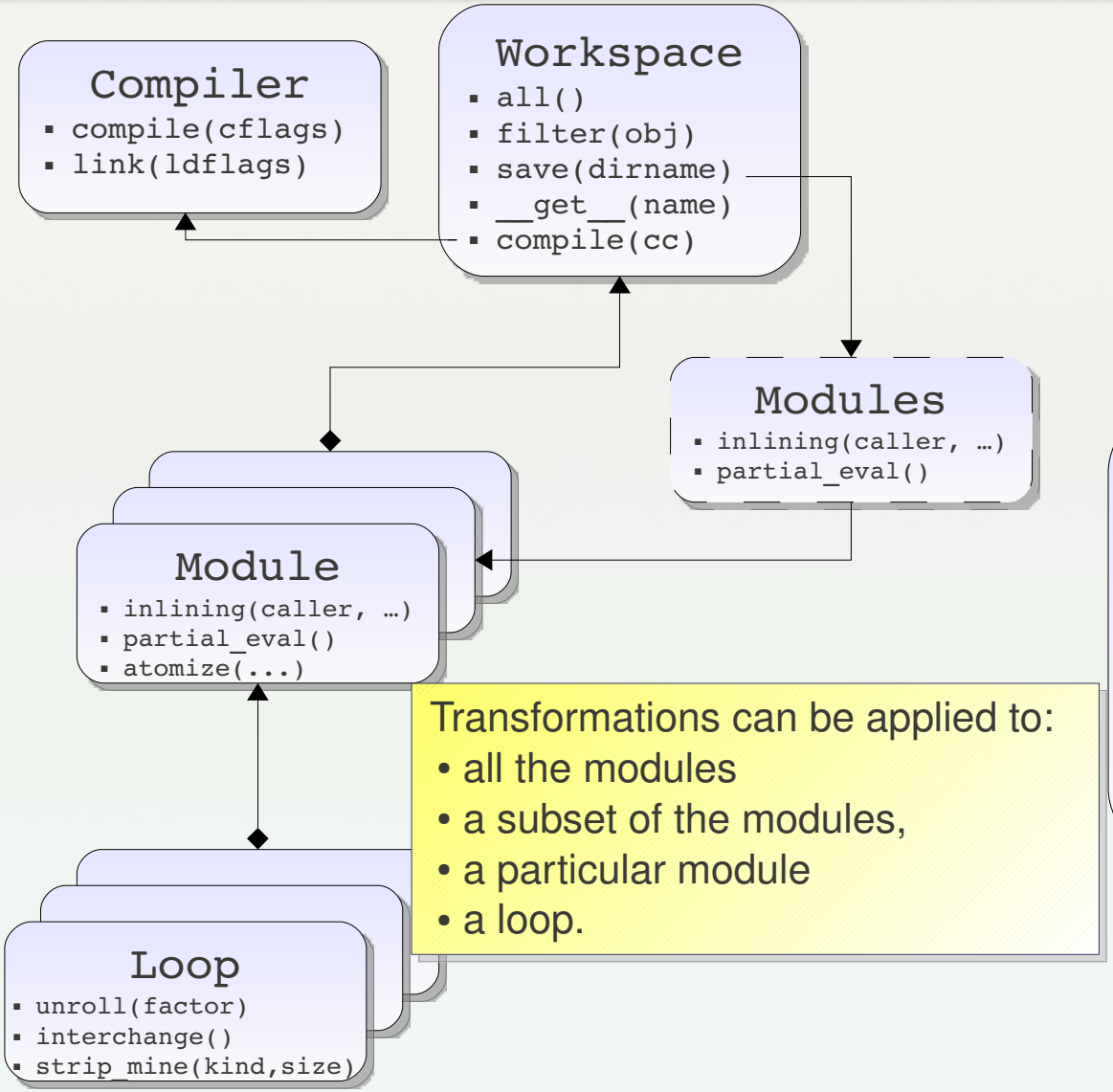
Reuse !

OOP

Interact

Abstract

Interface: PyPs Class Hierarchy



Transformations can be applied to:
- all the modules
- a subset of the modules,
- a particular module
- a loop.

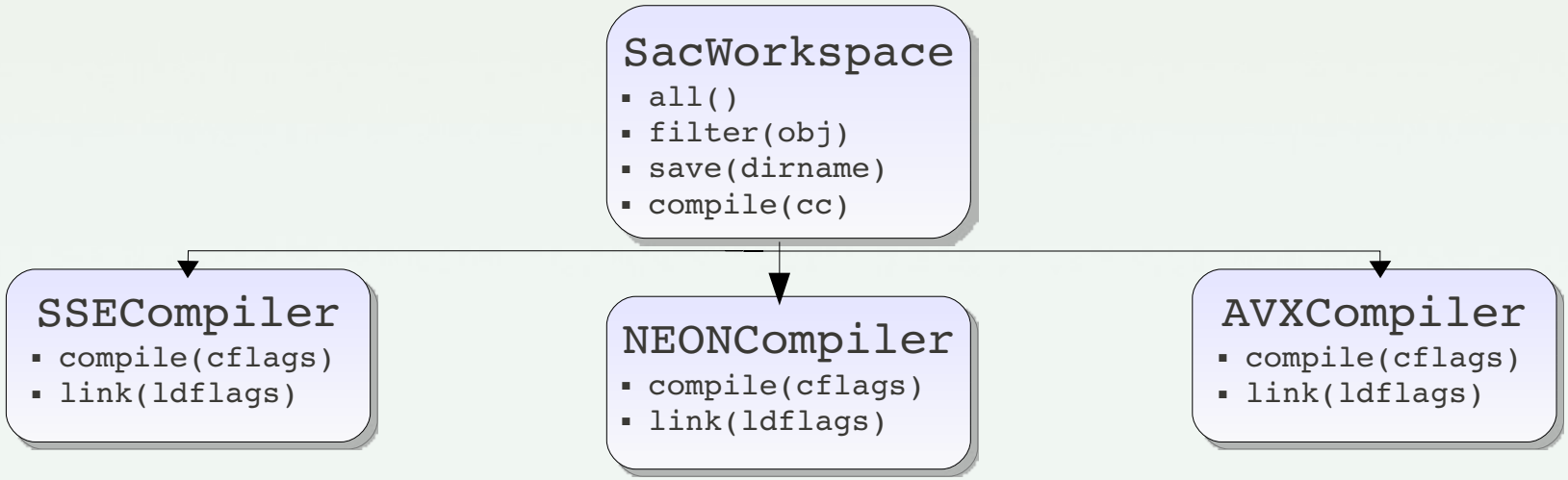
- Programs, Modules and Loops are first-level objects
- Collection of modules have the same interface as single modules

- Transformation extension through inheritance
- Transformation chaining with new methods
- Workspace hook through inheritance
- PostProcessing through compiler inheritence

```
$ sudo apt-get install python-pips
$ pydoc pyps
```

Level Bonuses: *sac*

- ***Simd Architecture Compiler (SAC)*:**
  - Reuse existing loop-level transformations such as tiling, unrolling etc
  - Combine it with Superword Level Parallelism (SLP)
  - Meta-Multimedia Instruction Set for multi target
- **Implementation:**
  - A generic compilation scheme implemented as a new workspace parametrized by the register length
  - A new compiler *per* backend with hook for generic to specific instruction conversion



Level Bonuses: Iterative Compilation

- **Goal:**
  - "Transformation space exploration": find a good transformation set for a given application
- **How:**
  - Explore the possibilities using a genetic algorithm
  - Use PyPS to dynamically
    - create workspaces
    - apply transformation sets
    - generate new source files
    - benchmark them
- **Extensions:**
  - Use it as a "fuzzer"
  - Use RPC ("Pyro") for distributed exploration

Developed in partnership with

Level II: Consistency Manager

- Automate interprocedural pass chaining
- Ensure analysis consistency
- Choose among analysis implementation (performance / accuracy tradeoff)

PIPS Consistency Manager: Pipsmake

II.3.2

```
\subsection{Detect Computation Intensive Loops}

\begin{PipsPass}{computation_intensity}
Generate a pragma on each loop that seems to be computation intensive according to a simple
cost model.
\end{PipsPass}

The computation intensity is derived from the complexity and the memory footprint.
It assumes the cost model:
$$execution\_time = startup\_overhead + \frac{memory\_footprint}{bandwidth} +
\frac{complexity}{frequency}$$
A loop is marked with pragma \PipsPropRef{COMPUTATION_INTENSITY_PRAGMA} if the
communication costs are lower
than the execution cost as given by \PipsPassRef{uniform_complexities}.
\begin{PipsMake}
computation_intensity > MODULE.code
    < MODULE.code
    < MODULE.regions
    < MODULE.complexities
\end{PipsMake}
```

Short description used for Python help

Long description For the manual

Cross references For pass parameters

Pass Dependency For automatic managment

Level III: Write the Code

Iterate over the Hierarchical Control Flow graph using `newgen`

```
computation_intensity_param p;
init_computation_intensity_param(&p);
gen_context_recurse(get_current_module_statement(),&p,
    statement_domain,do_computation_intensity,gen_null);
```

Collaborate with the consistency manager using `pipsdbm`

```
set_complexity_map( (statement_mapping)
db_get_memory_resource(DBR_COMPLEXITIES, module_name, true));
set_cumulated_rw_effects((statement_effects)db_get_memory_resource(DBR_REGIONS,
module_name, true));
```

Use the result of analysis as annotations

```
list regions =
load_cumulated_rw_effects_list(s);
complexity comp =
load_statement_complexity(s);
```

Level IV: linearlibs

Compute region memory usage

```
FOREACH(REGION,reg,regions) {
   Ppolynome reg_footprint= region_enumerate(reg);
   // may be we should use the rectangular hull ?
   polynome_add(&transfer_time,reg_footprint);
   polynome_rm(&reg_footprint);
}
```

Execution time estimation

```
Ppolynome instruction_time = polynome_dup(complexity_polynome(comp));
polynome_scalar_mult(&instruction_time,1.f/p->frequency);
...
polynome_negate(&transfer_time);
polynome_add(&instruction_time,transfer_time);
int max_degree = polynome_max_degree(instruction_time);
```

PIPS Technical View

At low level:

- Autotool-based build system
- C99 core libraries, Python extensions
- Litterate Programing everywhere
- `newgen` DSL
- `linear` Sparse algebra

At Higher level:

- A rich transformation toolbox
- Manipulated through high-level abstractions
- Use multiple inheritance to compose abstractions
- Use RPC to launch several instance of the compiler
- Leverage errors through exception mechanism

III. Demonstration

Goal: Generate and Benchmark Code for OpenMP + SSE

- Interact with PIPS through PyPS
- Chain program transformations
- Choose among various analyses and settings
- Reuse existing workspaces
- Edit intermediate textual representation

IV. Using PIPS

Using PIPS

- **Interprocedural static analyses**
  - Semantic
  - Memory effects
  - Dependences
  - Array Regions
- **Transformations**
  - Loop transformations
  - Code transformations
    - Restructuration, Cleaning,..
  - Memory re-allocations
    - Privatization, Scalarisation,..
- **Instrumentation**
  - Array bound checking
  - Alias checking
  - Variable initialization
- **Source code generation**
  - OpenMP
  - MPI

- Property verification: buffer overflow,...
- Optimization
- Parallelization
- Maintenance
- Reuse

A variety of goals: well beyond parallelization!

- Debugging
- Conformance to standards
- Heterogeneous computing: GPU/CUDA
- Visual programming
- Interactive compilation

- **Code modelling**
- **Prettyprint**
  - Source code [with analysis results]
  - Call tree, call graph
  - Interprocedural control flow graph

Key Concepts by Example


```
void bar(int n, double a[n], b[n], int i)
```

```
void foo(int n, double a[n],
double b[n])
{
  int j = 1;
  // precondition: j=1
  if(j<n) {
    // precondition: j=1 ^ j<n
    for(i=1; i<n-1; i++)
      // precondition: j=1 ^
j<n ^ 0<=i<n
      bar(n, a, b, i);


  }
}
```

Proper Read	Cumul. Read	Proper Written	Cumul. Written	
b[i]	b[*]	a[i]	a[*]	{a[i]=b[i]*b[i];}
a[i], b[i]	a[*], b[*]	a[i]	a[*]	{a[i]=a[i]+b[i];}
a[i-1], b[i]	a[*], b[*]	a[i]	a[*]	{a[i]=a[i-1]+b[i];}
b[i-1:i+1]	b[*]	a[i]	a[*]	{a[i] = b[i-1]+b[i]+b[i+1];}

a[i], b[0:i]	a[*], b[*]	a[i]	a[*]	{ int k;
				a[i]=0;
				for(k=0; k<= i; k++)
				a[i] += b[k]; }

∅	∅	a[i-1:i+1]	a[*]	{a[i-1]=-1.; a[i] = 0.; a[i+1] = 1;}

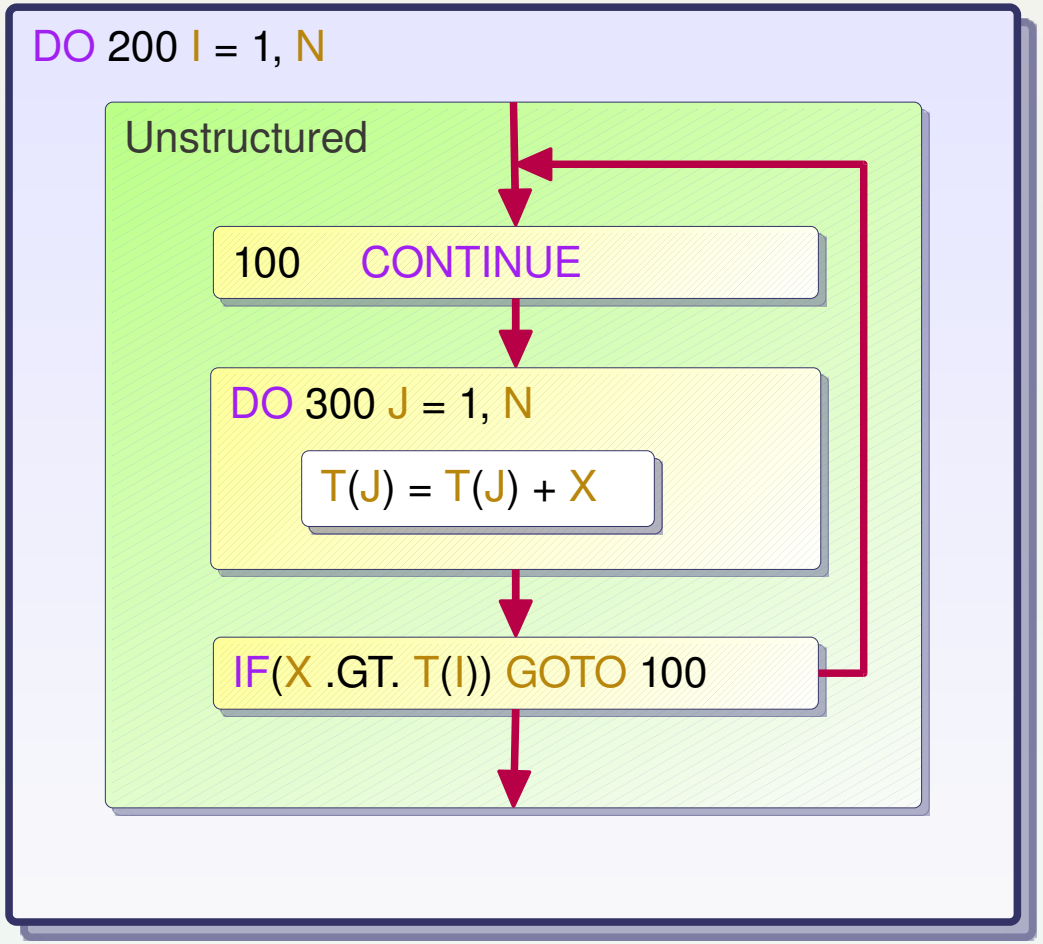
Key Concepts by Example (cont.)


```
      DO 200 I = 1, N
100     CONTINUE
        DO 300 J = 1, N
          T(J) = T(J) + X
300     CONTINUE
        IF(X .GT. T(I)) GOTO 100
200   CONTINUE
```

HCFG enables structural induction over AST:

$$\mathcal{F}(s1;s2) = C(\mathcal{F}(s1), \mathcal{F}(s2))$$

Hierarchical Control Flow Graph (HCFG)



Internal Representation: Newgen declarations

Excerpt from $PIPS_ROOT/src/Documentation/newgen/ri.tex :

- **statement** = label:entity
  x number:int x ordering:int
  x comments:string
  x **instruction**
  x declarations:entity*
  x decls_text:string x extensions;
- **instruction** = sequence + test
  + loop + whileloop
  + goto:statement
  + **call**
  + unstructured + multitest
  + forloop + expression;
- **call** = function:entity
  x arguments:expression*;

- **Newgen syntax:**
  - x : build a **structure**
  - + : build a **union**
  - * : build a **list**
  - string, int, float, ...: **basic types**
  - Also set **{}**, array **[]** and mapping **->**
- **Documentation:**
  - http://pips4u.org/doc/manuals
    (ri.pdf, ri_C.pdf)

In French: *Représentation Interne*, hence the many “ri”

The Internal Representation Interface: ri

- **Excerpt from** $PIPS_ROOT/include/ri.h:

```
#define statement_undefined ((statement)gen_chunk_undefined)
#define statement_undefined_p(x) ((x)==statement_undefined)

extern statement make_statement(entity, intptr_t, intptr_t, string, instruction, list, string,
extensions);
extern statement copy_statement(statement);
extern void free_statement(statement);

extern statement check_statement(statement);
extern bool statement_consistent_p(statement);
extern bool statement_defined_p(statement);

extern list gen_statement_cons(statement, list);

extern void write_statement(FILE*, statement);
extern statement read_statement(FILE*);

// gen_context_multi_recurse(obj, context, [domain, filter, rewrite,] * NULL);
```

Automatically generated by Newgen

Memory management

Debugging
Dynamic type checking

Typed lists

ASCII Serialization

Iterators

Static Analyses

Semantics:

- Transformers
  - Predicate about state transitions
- Preconditions
  - Predicate about state

Memory Effects:

- Read/Write effects
- In/Out effects
- Read/Write convex array regions
- In convex array regions
- Out convex array regions

Dependences:

- Use/def chains
- Region-based use/def chains
- Dependences (levels, cones)

Experimental Analyses:

- Flow-sensitive, context-insensitive pointer analysis
- Complexity
- Total preconditions

Principle: Each Function is Analyzed Once
Summaries must be built

Preconditions

- Affine predicates on scalar variables
  - Integer, float, complex, boolean, string
- Options:
  - Trust array references or Transformer in context, ...
- Innovative affine transitive closure operators

Includes symbolic ranges

```
//  P() {}
int main()
{
//  P() {}
  float a[10][10], b[10][10], h;
//  P(h) {}
  int i, j;
//  P(h,i,j) {}
  for(i = 1; i <= 10; i += 1)
//  P(h,i,j) {1<=i, i<=10}
    for(j = 1; j <= 10; j += 1)
//  P(h,i,j) {1<=i, i<=10, 1<=j, j<=10}
      b[i][j] = 1.0;
//  P(h,i,j) {i==11, j==11}
  h = 2.0;
//  P(h,i,j) {2.0==h, i==11, j==11}
  func1(10, 10, a, b, h);
//  P(h,i,j) {2.0==h, i==11, j==11}
  for(i = 1; i <= 10; i += 1)
//  P(h,i,j) {2.0==h, 1<=i, i<=10}
    for(j = 1; j <= 10; j += 1)
//  P(h,i,j) {2.0==h, 1<=i, i<=10, 1<=j, j<=10}
      fprintf(stderr, "a[%d] = %f \n", i, a[i][j]);
}
```

Preconditions (cont.)

- Interprocedural analysis:
  - Summary transformer, summary precondition
  - Top-down analysis

Summary Precondition

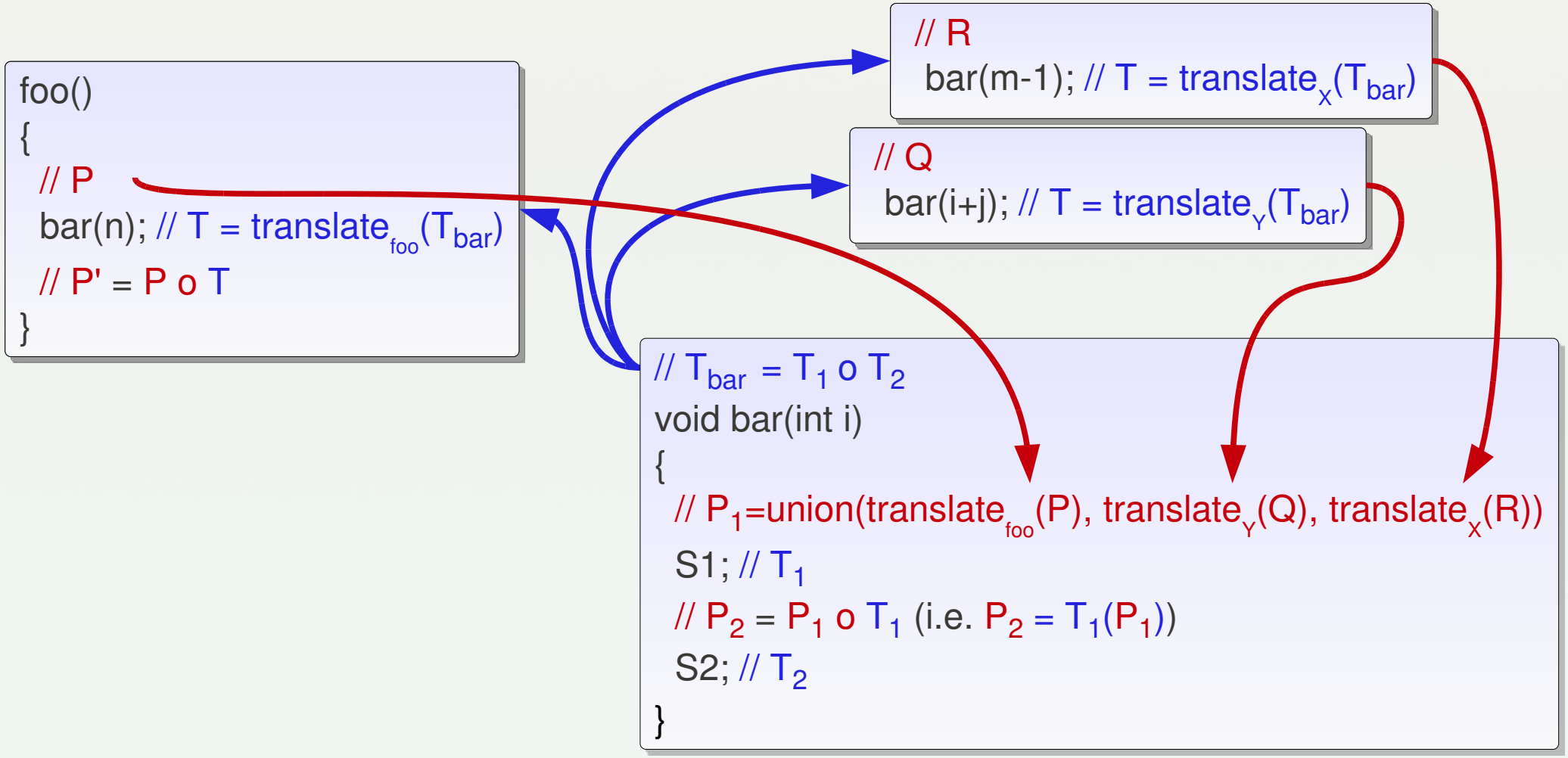
```
// P() {2.0==h, m==10, n==10}
void func1(int n, int m, float a[n][m], float b[n][m], float h)
{
// P() {2.0==h, m==10, n==10}
  float x;
// P(x) {2.0==h, m==10, n==10}
  int i, j;
// P(i,j,x) {2.0==h, m==10, n==10}
  for(i = 1; i <= 10; i += 1)
// P(i,j,x) {2.0==h, m==10, n==10, 1<=i, i<=10}
    for(j = 1; j <= 10; j += 1) {
// P(i,j,x) {2.0==h, m==10, n==10, 1<=i, i<=10, 1<=j, j<=10}
      x = i*h+j;
// P(i,j,x) {2.0==h, m==10, n==10, 1<=i, i<=10, 1<=j, j<=10}
      a[i][j] = b[i][j]*x;
    }
}
```

```
// P() {}
int main()
{
// P() {}
  float a[10][10], b[10][10], h;
// P(h) {}
  int i, j;
// P(h,i,j) {}
  for(i = 1; i <= 10; i += 1)
// P(h,i,j) {1<=i, i<=10}
    for(j = 1; j <= 10; j += 1)
// P(h,i,j) {1<=i, i<=10, 1<=j, j<=10}
      b[i][j] = 1.0;
// P(h,i,j) {i==11, j==11}
  h = 2.0;
// P(h,i,j) {2.0==h, i==11, j==11}
  func1(10, 10, a, b, h);
// P(h,i,j) {2.0==h, i==11, j==11}
  for(i = 1; i <= 10; i += 1)
// P(h,i,j) {2.0==h, 1<=i, i<=10}
    for(j = 1; j <= 10; j += 1)
// P(h,i,j) {2.0==h, 1<=i, i<=10, 1<=j, j<=10}
      fprintf(stderr, "a[%d] = %f \n", i, a[i][j]);
}
```

Call site

Affine Transformers, Preconditions and Summarization

- **Abstract store: precondition $P(\sigma_0, \sigma)$ or range($P(\sigma_0, \sigma)$)**
- **Abstract command: transformer $T(\sigma, \sigma')$**



Memory Effects

- **Used and def variables**
  - Read or Written
  - May or Exact
  - Proper, Cumulated or Summary

```
//        <must be read    >: n
//        <must be written>: i
  for(i = 1; i <= n; i += 1) {

//        <must be read    >: m n
//        <must be written>: j
    for(j = 1; j <= m; j += 1) {

//        <must be read    >: h i j m n
//        <must be written>: x
      x = i*h+j;

//        <must be read    >: b[i][j] i j m n x
//        <must be written>: a[i][j]
      a[i][j] = b[i][j]*x;
}}
```

Proper

```
//              <may be read    >: b[*][*] h
//              <may be written >: a[*][*]
//              <must be read   >: m n
func1(int n, int m, float a[n][m], float b[n][m], float h)
{
  float x;
  int i,j;
//              <may be read    >: b[*][*] h i j m x
//              <may be written >: a[*][*] j x
//              <must be read   >: n
//              <must be written>: i
  for(i = 1; i <= n; i += 1)
    for(j = 1; j <= m; j += 1) {
      x = i*h+j;
      a[i][j] = b[i][j]*x;
    }}
```

Summary

Cumulated

Convex Array Regions

- Bottom-up refinement of effects for array elements
- Polyhedral approximation of referenced array elements

```
//  <a[PHI1][PHI2]-W-EXACT-{1<=PHI1, PHI1<=n, 1<=PHI2, PHI2<=m, m==10, n==10}>
//  <b[PHI1][PHI2]-R-EXACT-{1<=PHI1, PHI1<=n, 1<=PHI2, PHI2<=m, m==10, n==10}>
void func1(int n, int m, float a[n][m], float b[n][m], float h)
{
  float x;
  int i,j;

//  <a[PHI1][PHI2]-W-EXACT-{1<=PHI1, PHI1<=n, 1<=PHI2, PHI2<=m, m==10, n==10}>
//  <b[PHI1][PHI2]-R-EXACT-{1<=PHI1, PHI1<=n, 1<=PHI2, PHI2<=m, m==10, n==10}>
  for(i = 1; i <= n; i += 1)

//  <a[PHI1][PHI2]-W-EXACT-{PHI1==i, 1<=PHI2, PHI2<=m, m==10, n==10, 1<=i, i<=n}>
//  <b[PHI1][PHI2]-R-EXACT-{PHI1==i, 1<=PHI2, PHI2<=m, m==10, n==10, 1<=i, i<=n}>
    for(j = 1; j <= m; j += 1) {
      x = i*h+j;

//  <a[PHI1][PHI2]-W-EXACT-{PHI1==i, PHI2==j, m==10, n==10, 1<=i, i<=10, 1<=j, j<=10}>
//  <b[PHI1][PHI2]-R-EXACT-{PHI1==i, PHI2==j,m==10, n==10, 1<=i, i<=10, 1<=j, j<=10}>
      a[i][j] = b[i][j]*x;
    }}
```

Interprocedural preconditions are used

A triangular iteration space could be used as well

Convex Array Regions: Use Transformers and Preconditions

- **Regions: Functions from stores σ to sets of elements φ for arrays A, …**
- **Functions $\varphi=r_A(\sigma)$ or function graphs $R_A(\varphi, \sigma)$**
- **Approximation: MAY, MUST, EXACT**
- **Use transformers $T(\sigma, \sigma')$ and preconditions $P(\sigma)=range(P(\sigma_0, \sigma))$**
  - Note: σ_0 is the function initial state

Assume S does not use nor define elements of A

```
// P(σ)

// r_A(σ) : σ → { φ | R_A(φ, σ) }

S: i++; // T(σ, σ')

// r_A(σ') : σ' → { φ | R'_A(φ, σ') }

S': a[i] = ...; // T(σ', σ'')
```

Exact quantifier elimination?

Quantifier elimination

$$R_A(\varphi, \sigma) = \{ (\varphi, \sigma) \mid \exists \sigma' \;\; T(\sigma, \sigma') \wedge R'_A(\varphi, \sigma') \wedge P(\sigma) \}$$

IN and OUT Convex Array Regions

- **IN convex array region for Statement S**
  - Memory locations whose values are used by S before they are defined
- **OUT convex array region for S**
  - Memory locations defined by S, and whose values are used later by the program
  - Sometimes surprising... when no explicit continuation exists: garbage in, garbage out

Non convex regions?

```
//  <b[PHI1][PHI2]-IN-EXACT-{ 1<=PHI1, PHI1<=n,  1<=PHI2, PHI2<=m,
//        m==10, n==10}>

//  <a[PHI1][PHI2]-OUT-EXACT-{ 1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10,
//        m==10, n==10}>

S: for(i = 1; i <= n; i += 1)
     for(j = 1; j <= m; j += 1) {
       x = i*h+j;
      a[i][j] = b[i][j]*x;
     }
```

Requires non-monotonic operators: MUST or EXACT regions
IN(S1;S2) = IN(S1) U (READ(S2) – WRITE(S1))

Data Dependence

- **Several dependence test algorithms:**
  - Fourier-Motzkin with different information:
    - rice_fast_dependence_graph
    - rice_full_dependence_graph
    - rice_semantics_dependence_graph
  - Properties
    - Read-read dependence arcs
- **Dependence abstractions:**
  - Dependence level
  - Dependence cone
    - Includes uniform dependencies
- **Prettyprint dependence graph:**
  - Use-def chains
  - Dependence graph

My parallel loop is still sequential:

Why?

Dependence test?

Look at the dependence graph?

My parallel loop is still sequential:

Why?

Array Privatization?

Complexity

- **Symbolic approximation of execution cost: polynomials**

```
//                                                    1721 (SUMMARY)
void func1(int n, int m, float a[n][m], float b[n][m], float h)
```

```
//                                                    17*m.n + 3*n + 2 (SUMMARY)
void func1(int n, int m, float a[n][m], float b[n][m], float h)
{
  float x;
  int i, j;
//                                                    17*m.n + 3*n + 2 (DO)
  for(i = 1; i <= n; i += 1)
//                                                    17*m + 3 (DO)
    for(j = 1; j <= m; j += 1) {
//                                                     6 (STMT)
      x = i*h+j;
//                                                    10 (STMT)
      a[i][j] = b[i][j]*x;
    }
}
```

Application: complexity comparison before and after constant propagation.

P() {m==10, n==10}

Based on a parametric cost table

Loop Transformations

- Loop Distribution
- Index set splitting
- Loop Interchange
- Hyperplane method
- Loop Normalization
- Strip Mining
- Tiling
- Full/Partial Unrolling
- Parallelizations

```
apply PARTIAL_EVAL[convol]
apply LOOP_TILING[convol]
apply FULL_UNROLL[convol]
apply PARTIAL_EVAL[convol]
apply SCALARIZATION[convol]
display PRINTED_FILE[convol]
```

- Tiling example with convol

```
void convol(int isi, int isj, float new_image[isi][isj], float image[isi][isj],
int ksi, int ksj, float kernel[ksi][ksj])
{
  int i, j, ki, kj;
  int i_t, j_t; float __scalar__0;        //PIPS generated variables
l400:
  for(i_t = 0; i_t <= 3; i_t += 1)
    for(j_t = 0; j_t <= 3; j_t += 1)
      for(i = 1+128*i_t; i <= MIN(510, 128+128*i_t); i += 1)
        for(j = 1+128*j_t; j <= MIN(128+128*j_t, 510); j += 1) {
          __scalar__0 = 0.;
l200:     __scalar__0 = __scalar__0+image[i-1][j-1]*kernel[0][0];
          __scalar__0 = __scalar__0+image[i-1][j]*kernel[0][1];
          __scalar__0 = __scalar__0+image[i-1][j+1]*kernel[0][2];
          __scalar__0 = __scalar__0+image[i][j-1]*kernel[1][0];
          __scalar__0 = __scalar__0+image[i][j]*kernel[1][1];
          __scalar__0 = __scalar__0+image[i][j+1]*kernel[1][2];
          __scalar__0 = __scalar__0+image[i+1][j]*kernel[2][1];
          __scalar__0 = __scalar__0+image[i+1][j+1]*kernel[2][2];
          __scalar__0 = __scalar__0/9;
          new_image[i][j] = __scalar__0; }}
```

Loop Parallelization

- **Allen & Kennedy**
- **Coarse grain**
- **Nest parallelization**

```
      PROGRAM NS
      PARAMETER (NVAR=3,NXM=2000,NYM=2000)
      REAL PHI(NVAR,NXM,NYM),PHI1(NVAR,NXM,NYM)
      REAL PHIDES(NVAR,NYM)
      REAL DIST(NXM,NYM),XNOR(2,NXM,NYM),SGN(NXM,NYM)
      REAL XCOEF(NXM,NYM),XPT(NXM),YPT(NXM)


!$OMP  PARALLEL DO PRIVATE(I,PX,PY,XCO)
      DO J = 2, NY-1

!$OMP  PARALLEL DO PRIVATE(PX,PY,XCO)
        DO I = 2, NX-1
          XCO = XCOEF(I,J)
          PX = (PHI1(3,I+1,J)-PHI1(3,I-1,J))*H1P2
          PY = (PHI1(3,I,J+1)-PHI1(3,I,J-1))*H2P2
          PHI1(1,I,J) = PHI1(1,I,J)-DT*PX*XCO
          PHI1(2,I,J) = PHI1(2,I,J)-DT*PY*XCO
        ENDDO
      ENDDO
      END
```

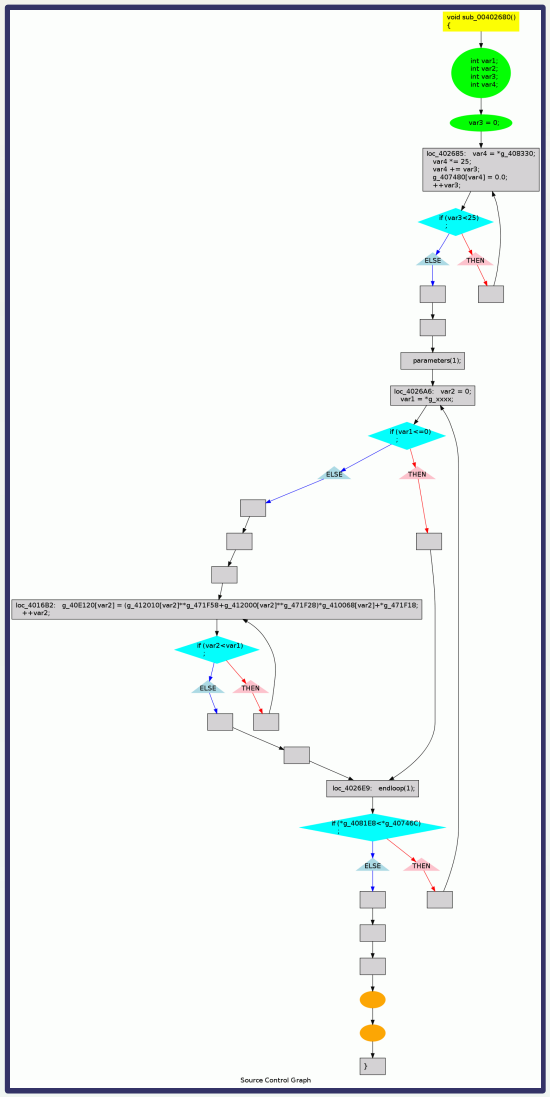
Code Transformation Phases (1)

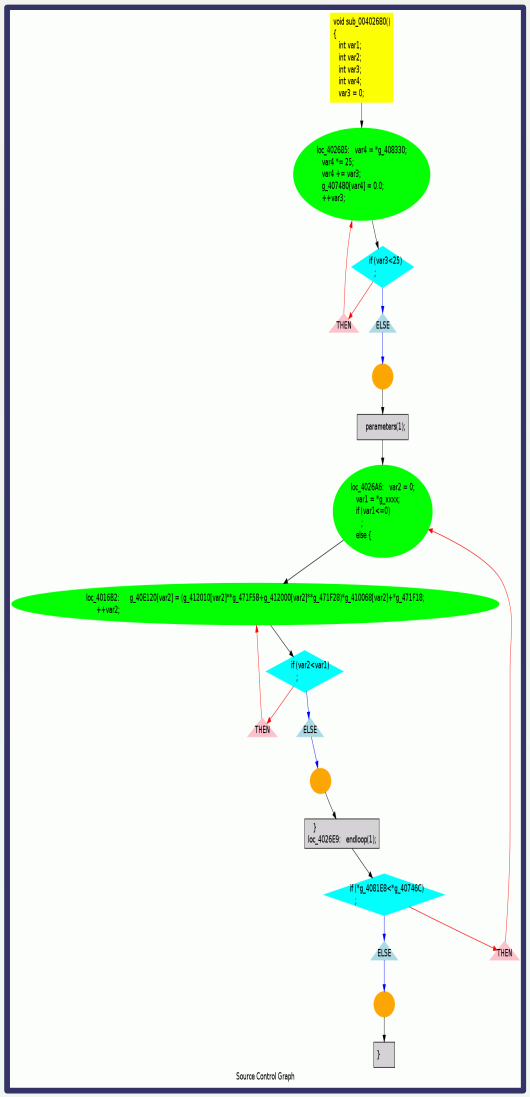
- **Three-address code**
  - *Atomizers*
  - Two-address code
- **Reduction recognition**
- **Expression optimizations:**
  - Common subexpression elimination
  - Forward substitution
  - Invariant code motion
  - Induction variable substitution
- **Restructuring**
  - Restructure control
  - Split initializations
- **Memory optimizations:**
  - Scalar privatization
  - Array privatization from regions
  - Array/Scalar expansion
  - Scalarization

Code Transformation Phases (2)

- **Cloning**
- **Inlining**
- **Outlining**
- **Partial evaluation from preconditions**
  - Constant propagation + evaluation
- **Dead code elimination**
- **Control simplification**
- **Control restructurations**
  - Hierarchization
  - if/then/else restructuring
  - Loop recovery
  - For- to do-loop

A hierarchization example :





Inlining and Outlining


```
void convol(int n, int a[n][n], int b[n][n], int
kernel[3][3])
{
  int i, j;
  for(i = 0; i <= n-1; i += 1)
    for(j = 0; j <= n-1; j += 1) {
      int k, l;
      b[i][j] = 0;
      for(k = 0; k <= 2; k += 1)
        for(l = 0; l <= 2; l += 1)
          b[i][j] += a[i+k-1][j+l-1]*kernel[k][l];
    }
}
```

```
void convol_outlined(int n, int i, int a[n][n], int
b[n][n], int kernel[3][3])
{
  //PIPS generated variable
  int j;
l99996:
  for(j = 0; j <= n-1; j += 1) {
    int k, l;
    b[i][j] = 0;
l99997:
    for(k = 0; k <= 2; k += 1)
l99998:
      for(l = 0; l <= 2; l += 1)
        b[i][j] += a[i+k-1][j+l-1]*kernel[k][l];
  }
}
```

```
void convol(int n, int a[n][
int kernel[3][3])
{
  int i, j;
l99995:
  for(i = 0; i <= n-1; i += 1)
l99996:     convol_outlined(n, i,
kernel);
}
```

```
apply UNFOLDING[convol]
apply FLAG_LOOPS[convol]
setproperty OUTLINE_LABEL "l99996"
setproperty OUTLINE_MODULE_NAME
"convol_outlined"
apply OUTLINE[convol]
```

Cloning (+ Constant Propagation + Dead Code Elimination)


```
# 1
int clone01(int n, int s)
{
  int r = n;
  if(s<0)
    r = n-1;
  else if(s>0)
    r = n+1;
  return r;
}
```

Imprecise summary transformer

```
int clone01_2(int n, int s)
int clone01_1(int n, int s)
int clone01_0(int n, int s)
{
   // PIPS: s is assumed a
constant reaching value
   return 0;
}
```

```
//  P() {}
int main()
{
//  P() {}
   int i = 1;
//  P(i) {i==1}
   i = clone01(i, -1);
//  P(i) {0<=i, i<=2}
   i = clone01(i, 1);
//  P(i) {0<=i+1, i<=3}
   i = clone01(i, 0);
}
```

Imprecise preconditions

Exact preconditions

```
//  P() {}
int main()
{
//  P() {}
   int i = 1;
//  P(i) {i==1}
   i = clone01_0(i, -1);
//  P(i) {i==0}
   i = clone01_1(i, 1);
//  P(i) {i==1}
   i = clone01_2(i, 0);
}
```

Dead Code Elimination (1)

- **Control Simplification:**
  - Redundant test elimination
  - Use preconditions to eliminate tests and simplify zero- and one-trip loops
- **Partial evaluation**
  - Interprocedural constant propagation
- **Use-def elimination**

```
int clone01_1(int n, int s)
{
  // PIPS: s is assumed a constant reaching value
  if (s!=1)
    exit(0);
  {
    int r = n;
    if (s<0)
      r = n-1;
    else if (s>0)
      r = n+1;
    return r;
  }
}
```

```
int clone01_1(int n, int s)
{
  // PIPS: s is assumed a constant reaching value
  if (1!=1)
    exit(0);
  {
    int r = 0;
    if (1<0)
      r = n-1;
    else if (1>0)
      r = 1;
    return 1;
  }
}
```

```
int clone01_1(int n, int s)
{
  // PIPS: s is assumed a constant reaching value
  ;
  return 1;
}
```

```
int clone01_1(int n, int s)
{
  // PIPS: s is assumed a constant reaching value
  return 1;
}
```

Dead Code Elimination (2)

- **Partial eval**
- **Control simplification**
- **Use-def elimination**

```
int clone02_1(int n, int s)
{
  // PIPS: s is assumed
a constant reaching
value
  if (s!=1)
    exit(0);
  {
    int r = n;
    if (s<0)
      r = n-1;
    else if (s>0)
      r = n+1;
    return r;
  }
}
```

Cloning warning

```
int clone02_1(int n, int s)
{
  // PIPS: s is assumed a
constant reaching value
  if (1!=1)
    exit(0);
  {
    int r = 0;
    if (1<0)
      r = n-1;
    else if (1>0)
      r = 1;
    return 1;
  }
}
```

```
int clone02_1(int n,
int s)
{
  // PIPS: s is
assumed a constant
reaching value
  int r = 0;
  r = 1;
  return 1;
}
```

```
int clone02_1(int n, int s)
{
  // PIPS: s is assumed a
constant reaching value
  ;
  return 1;
}
```

Maintenance and Debugging: Dynamic Analyses

- **Uninitialized variable detection (used before set, UBS)**
- **Fortran type checking**
- **Declarations: cleaning**
- **Array resizing**
- **Fortran alias detection**
- **Array bound checking**

```
!!
!! file for scalar02.f
!!
      PROGRAM SCALAR02
      INTEGER X,Y,A,B
      EXTERNAL ir_isnan,id_isnan
      LOGICAL*4 ir_isnan,id_isnan
      STOP 'Variable SCALAR02:Y is used before set'
      STOP 'Variable SCALAR02:B is used before set'
      X = Y
      A = B
      PRINT *, X, A
      B = 1
      RETURN
      END
```

Prettyprint

- **Fortran 77**
  - + OpenMP directives
  - + Fortran 90 array expressions
- **Fortran 77: a long history...**
  - + HPF directives
  - + DOALL loops
  - + Fortran CRAY
  - + CMF
- **C**
  - + OpenMP directives
- **XML**
  - Code modelling
  - Visual programming
- **Graphs**
  - Call tree, call graph
  - Use-Def chains
  - Dependence graph
  - Interprocedural control flow graph

The results of all PIPS analyses can be prettyprinted and visualized with the source code:

activate PRINT_CODE_PRECONDITIONS
display PRINTED_FILE

Source Code Generation

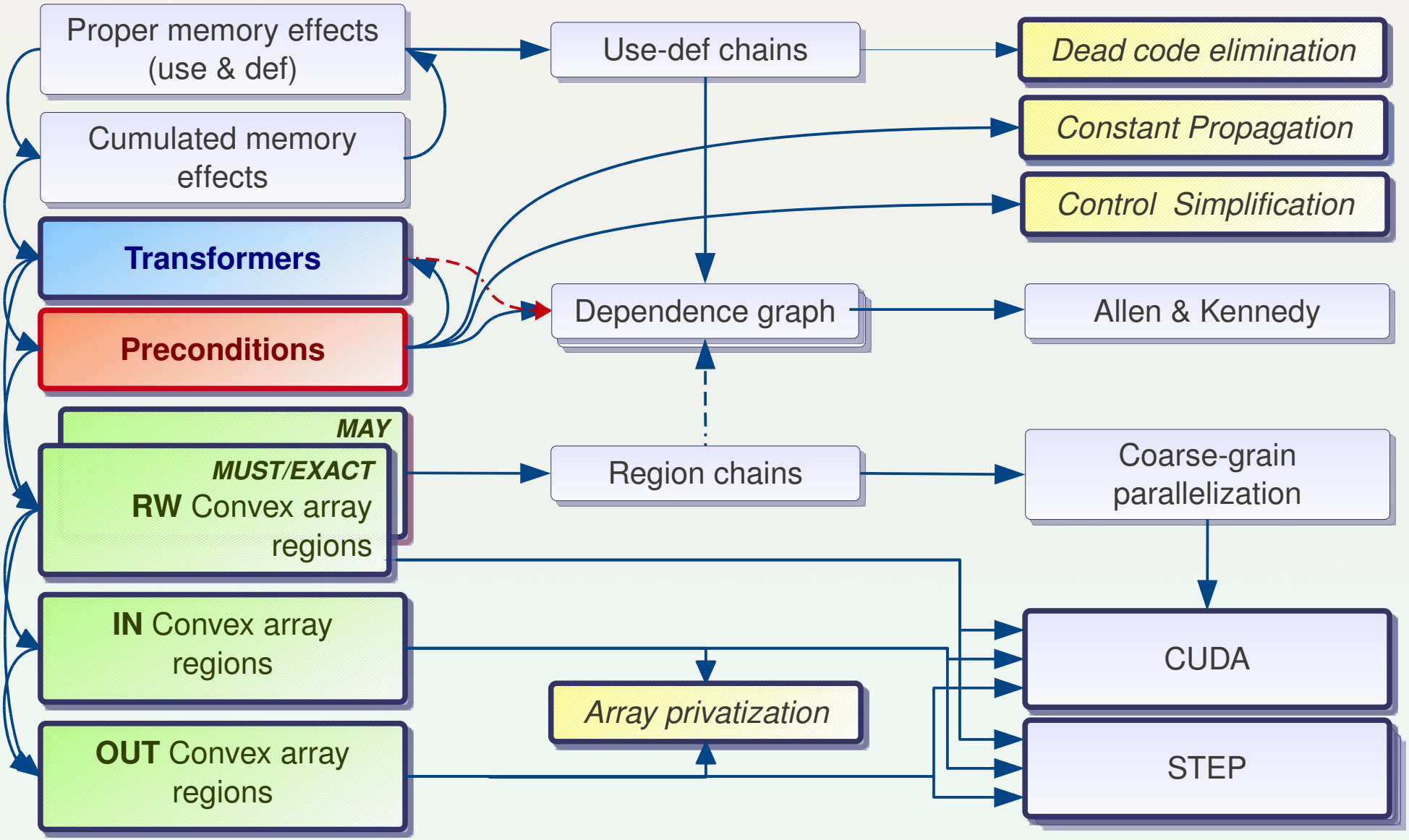
- **HPF**
  - MPI
  - PVM
- **OpenMP → MPI**
- **GPU/CUDA**
- **SSE**
- **Ongoing:**
  - OpenCL
  - FREIA

Excerpt of an image alphablending function

```
....
  SIMD_LOAD_GENERIC_V4SF(v4sf_vec1, alpha, alpha, alpha, alpha);
  SIMD_LOAD_CONSTANT_V4SF(v4sf_vec4, 1, 1, 1, 1);
  LU_IND0 = LU_IB0+MAX(INT((LU_NUB0-LU_IB0+3)/4), 0)*4;
  SIMD_SUBPS(v4sf_vec3, v4sf_vec4, v4sf_vec1);
  for(LU_IND0 = LU_IB0; LU_IND0 <= LU_NUB0-1; LU_IND0 += 4) {
    SIMD_LOAD_V4SF(v4sf_vec2, &src1[LU_IND0]);
    SIMD_MULPS(v4sf_vec0, v4sf_vec1, v4sf_vec2);
    SIMD_LOAD_V4SF(v4sf_vec8, &src2[LU_IND0]);
    SIMD_MULPS(v4sf_vec6, v4sf_vec3, v4sf_vec8);
    SIMD_ADDPS(v4sf_vec9, v4sf_vec0, v4sf_vec6);
    SIMD_SAVE_V4SF(v4sf_vec9, &result[LU_IND0]);
  }
  SIMD_SAVE_GENERIC_V4SF(v4sf_vec0, &F_03, &F_02, &F_01, &F_00);
  SIMD_SAVE_GENERIC_V4SF(v4sf_vec3, &F_13, &F_12, &F_11, &F_10);
  SIMD_SAVE_GENERIC_V4SF(v4sf_vec6, &F_23, &F_22, &F_21, &F_20);
}
```

Assembly level code

Relationships: Analyses, Transformations & Code Generation



Using PIPS: Wrap-Up

- **Analyze...**
  - to decide what parts of code to optimize
  - to detect parallelism
- **Transform...**
  - to simplify, optimize locally
  - to adjust code to memory constraints and parallel components
- **Generate code for a target architecture**
  - SSE
  - CUDA

- Interprocedural analyses
  - Preconditions, array regions, dependences, complexity
- Transformations
  - Constant propagation, loop unrolling,
  - Expression optimization, privatization, scalarization,
  - Loop parallelization, tiling, inlining, outlining,
- Prettyprints
  - OpenMP

V. Ongoing Projects Based on PIPS

V.0.1

V. Ongoing Projects Based on PIPS

V. Ongoing Projects Based on PIPS

V.0.2

- **What can you do by combining basic analyses and transformations?**
  - Heterogeneous code optimization for a hardware accelerator: **FREIA / SpoC** (ANR Project)
  - Generic vectorizer for SIMD instructions
  - OpenMP to MPI: the STEP phase (ParMA European Project)
  - GPU / CUDA
  - OpenCL (FUI OpenGPU Project)
  - Code generation for hardware accelerators (SCALOPES European Project)

STEP

V.1.1

- **STEP: Transformation System for Parallel Execution**
- **Use a single program to run both on shared-memory and distributed-memory architectures**
- **Parallelism specified via OpenMP directives**
- **A shared-memory OpenMP program is translated into a MPI program to run on distributed-memory machines**

OpenMP Directives

V.1.2

Parallel construct

Worksharing on loop i

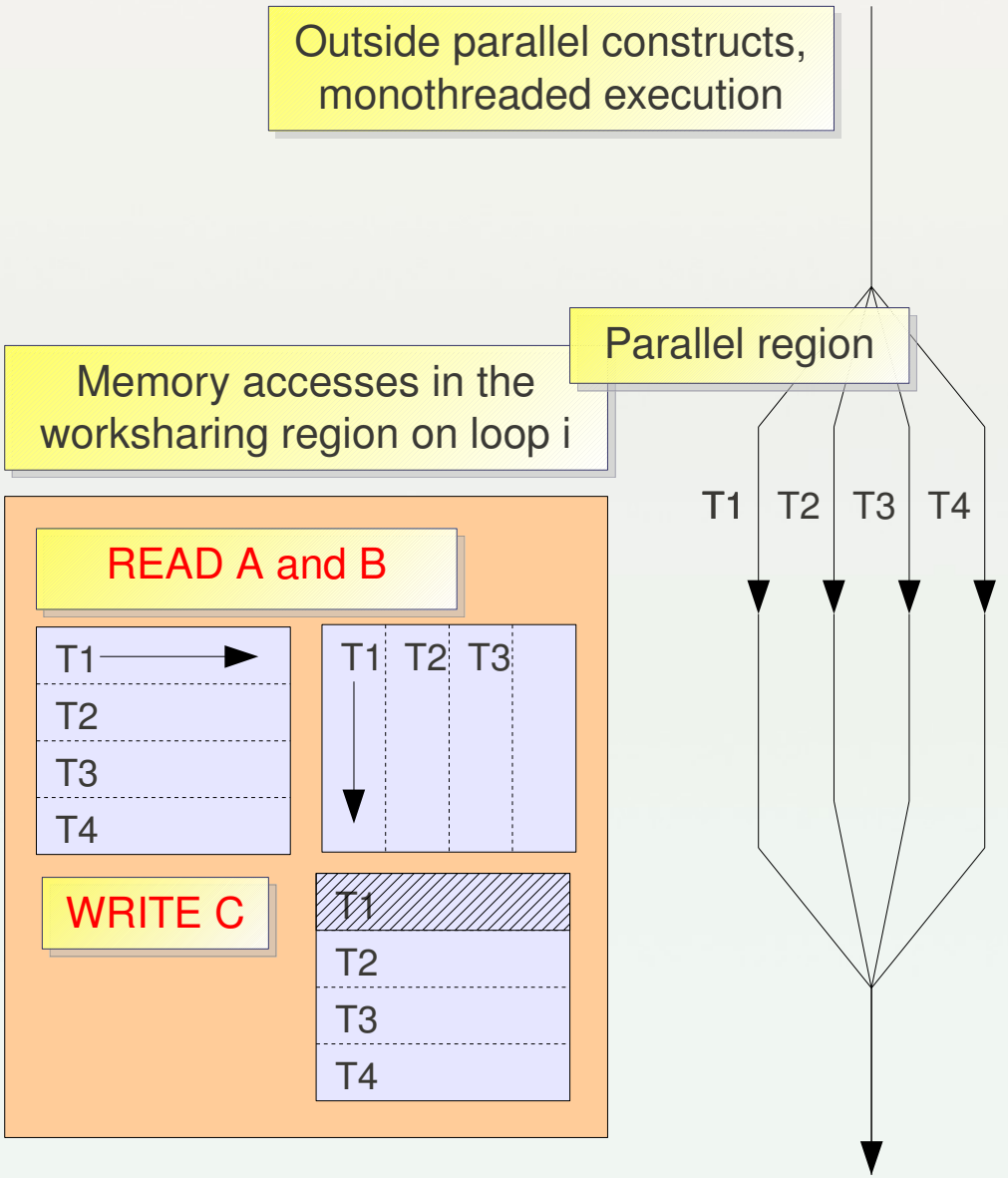
```
#pragma omp parallel for shared(A, B, C)
private(i, j, k)
for (i = 0; i < N; i++) {
 for (j = 0; j < N; j++) {
  for (k = 0; k < N; k++) {
    C[i][j] = C[i][j] + A[i][k] * B[k][j];
 } } }
```

Using OpenMP:

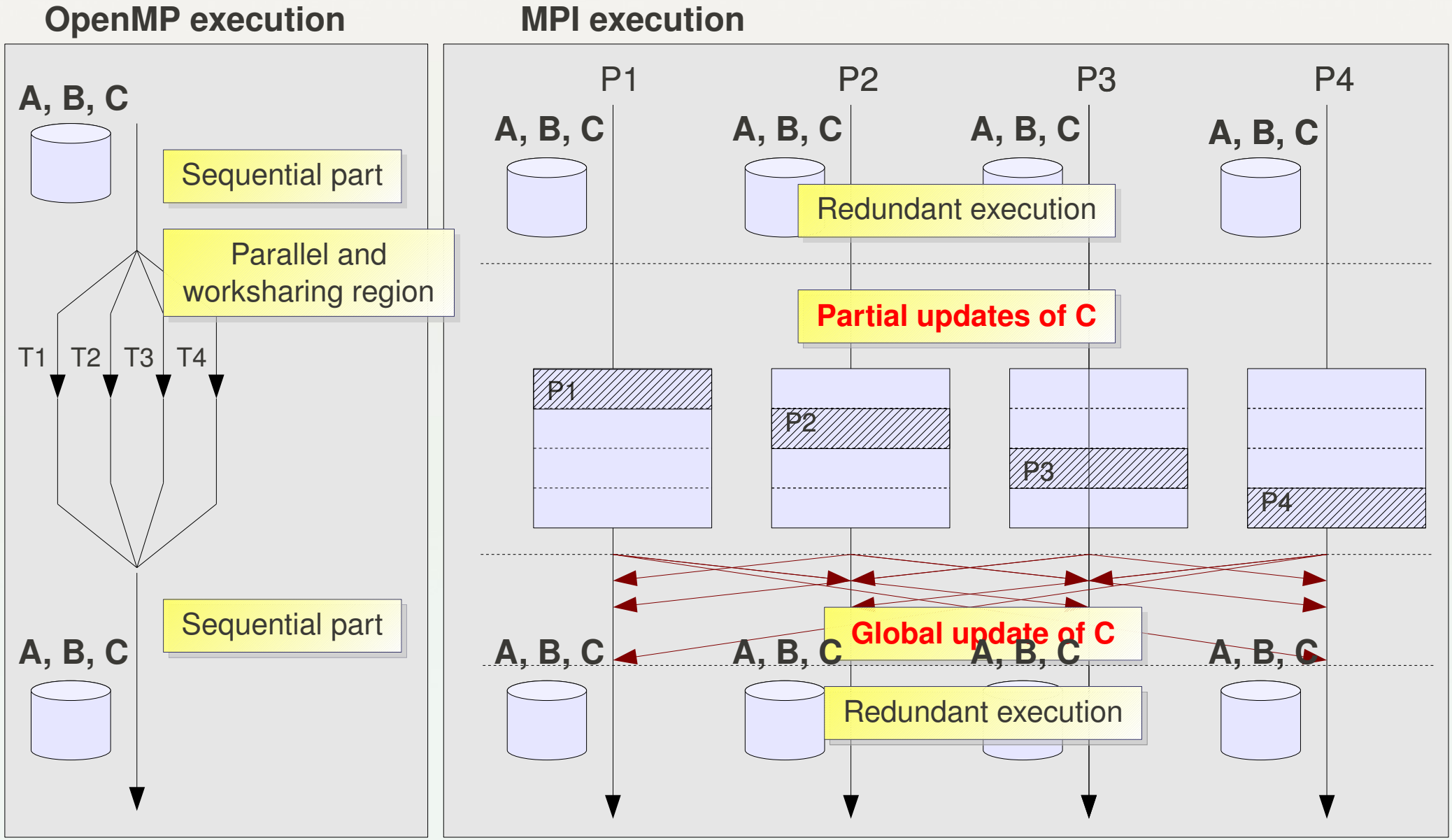
- **The programmer** must guarantee that the code is correct
- … and avoid concurrent write access

Based on relaxed-consistency memory:

- **Update main memory** at specific points
- Explicit synchronisation primitives such as *flush*



From a Shared-Memory to a Distributed-Memory Execution Model V.1.3



From OpenMP to MPI: three main phases

V.1.4

```
#pragma omp parallel for shared(A, B, C)
private(i, j, k)
for (i = 0; i < N; i++) {
 for (j = 0; j < N; j++) {
  for (k = 0; k < N; k++) {
    C[i][j] = C[i][j] + A[i][k] * B[k][j];
 } } }
```

1) Identify parallel constructs and compute worksharing

2) Global update: all2all communication

- Determine **modified data** inside the worksharing region for each process

- Find which process needs which data

3) Generate MPI code

SPMD message-passing programming

```
/*
  Explicit worksharing
  depending on process ID
*/

nbrows = N / nbprocs;
i_low = myrank * nbrows;
i_up =  (myrank + 1) * nbrows;

for (i = i_low; i < i_up; i++) {
  for (j = 0; j < N; j++) {
    for (k = 0; k < N; k++) {
      C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
  }
}

/* Explicit data update */
All2all_update(C);
```

Using PIPS for STEP

V.1.5

- **Interprocedural analyses**
  - Array regions as convex polyhedra
  - EXACT, MAY approximations
  - IN, OUT, READ, WRITE
- **PIPS as a workbench**
  - Intermediate representation
  - Program manipulation
  - Pretty-printer
  - Source-to-source transformation

Program Example

V.1.6

```
      PROGRAM MATMULT
      implicit none
      INTEGER  N, I, J, K
      PARAMETER (N=1000000)
      REAL*8 A(N,N), B(N,N), C(N,N)

      CALL INITIALIZE(A, B, C, N)

C     Compute matrix-matrix product
!$OMP PARALLEL DO
      DO 20 J=1, N
        DO 20 I=1, N
          DO 20 K=1, N
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
 20       CONTINUE
!$OMP END PARALLEL DO

      CALL PRINT(C, N)
      END
```

Three PIPS modules:
- INITIALIZE
- PRINT
- MATMUL

One parallel loop in the MATMUL program

First PIPS phase: « STEP_DIRECTIVES »

V.1.7

- **Parse the OpenMP program:**
  - Recognize OpenMP directives
  - Outline OpenMP constructs in separate procedures

Output resources

```
step_directives   > PROGRAM.directives
                  > PROGRAM.outlined
                  > MODULE.code
                  > MODULE.callees
  ! MODULE.directive_parser
  < PROGRAM.entities
  < PROGRAM.outlined
  < PROGRAM.directives
  < MODULE.code
  < MODULE.callees
```

Input resources

```
create myworkspace matmul.f


apply STEP_DIRECTIVES[%ALL]


close
```

"On all modules"

« STEP_DIRECTIVES » results

V.1.8

MATMULT.f

```
      PROGRAM MATMULT
! MIL-STD-1753 Fortran extension not in PIPS
!     implicit none
      INTEGER  N, I, J, K
      PARAMETER (N=1000000)
      REAL*8 A(N,N), B(N,N), C(N,N)

      CALL INITIALIZE(A, B, C, N)
C     !$omp parallel do
      CALL MATMULT_PARDO20(J, 1, N, I, N, K, C, A, B)

      CALL PRINT(C, N)
      END
```

Initial MATMUL calling the outlined function

New module containing the parallel DO loop

MATMULT_PARDO20.f

```
      SUBROUTINE MATMULT_PARDO20(J, J_L, J_U, I, N, K, C, A, B)
      INTEGER J, J_L, J_U, I, N, K
      REAL*8 C(1:N, 1:N), A(1:N, 1:N), B(1:N, 1:N)
      DO 20 J = J_L, J_U
        DO 20 I = 1, N
          DO 20 K = 1, N
            C(I,J) = C(I,J)+A(I,K)*B(K,J)
20          CONTINUE
      END
```

Second PIPS Phase: « STEP_ANALYSE »

V.1.9

- **Parse the OpenMP program containing outlined functions**
- **For each outlined module corresponding to a OpenMP construct:**
  - Apply PIPS analyses: IN, OUT, READ, WRITE array regions
  - Compute SEND array regions describing data that have been modified by each process

```
create myworkspace matmul.f
activate MUST_REGIONS
activate TRANSFORMERS_INTER_FULL



apply STEP_DIRECTIVES[%ALL]
apply STEP_ANALYSE[%ALL]


close
```

MUST_REGIONS for the most precise analysis

TRANSFORMERS for accurate analysis
(translation of linear expressions...)

```
step_analyse            > PROGRAM.step_analyses
  < PROGRAM.entities
  < PROGRAM.directives
  < PROGRAM.step_analyses
  < MODULE.code
  < MODULE.summary_regions
  < MODULE.in_summary_regions
  < MODULE.out_summary_regions
```

Input/output resources

We ask for PIPS summary READ, WRITE, IN and OUT regions to be computed HERE!

« STEP_ANALYSE » Results

V.1.10

```
C  <C(PHI1,PHI2)-W-EXACT-{1<=PHI1, PHI1<=N, J_L<=PHI2, PHI2<=J_U}>
C  <C(PHI1,PHI2)-OUT-EXACT-{1<=PHI1, PHI1<=1000000, 1<=PHI2,
C   PHI2<=1000000, J_L==1, J_U==1000000, N==1000000}>

      SUBROUTINE MATMULT_PARDO20(J, J_L, J_U, I, N, K, C, A, B)
      INTEGER J, J_L, J_U, I, N, K
      REAL*8 C(1:N, 1:N), A(1:N, 1:N), B(1:N, 1:N)

      DO 20 J = J_L, J_U
        DO 20 I = 1, N
          DO 20 K = 1, N
            C(I,J) = C(I,J)+A(I,K)*B(K,J)
20          CONTINUE
      END
```

Print WRITE and OUT summary regions

WRITE regions

OUT regions

Compute SEND regions depending on loop bounds:
WRITE ∩ OUT

```
C  <C(PHI1,PHI2)-write-EXACT-{1<=PHI1, PHI1<=N, PHI1<=1000000,
C   J_LOW<=PHI2, 1<=PHI2, PHI2<=J_UP, PHI2<=1000000}>
```

WRITE and OUT regions for array C
PHI1 (first dimension) is modified on all indices
PHI2 (second dimension) is modified between J_LOW and J_UP
SEND regions correspond to blocks of C rows

Third PIPS Phase: « STEP_COMPILE »

V.1.11

- **For each OpenMP directive**
  - Generate MPI code in outlined procedures (when necessary)

```
create myworkspace matmul.f
activate MUST_REGIONS
activate TRANSFORMERS_INTER_FULL



apply STEP_DIRECTIVES[%ALL]
apply STEP_ANALYSE[%ALL]
apply STEP_COMPILE[%MAIN]

close
```

```
step_compile      > PROGRAM.step_status
                  > MODULE.code
                  > MODULE.callees
  ! CALLEES.step_compile
  < PROGRAM.entities
  < PROGRAM.outlined
  < PROGRAM.directives
  < PROGRAM.step_analyses
  < PROGRAM.step_status
  < MODULE.code
```

Input/output resources

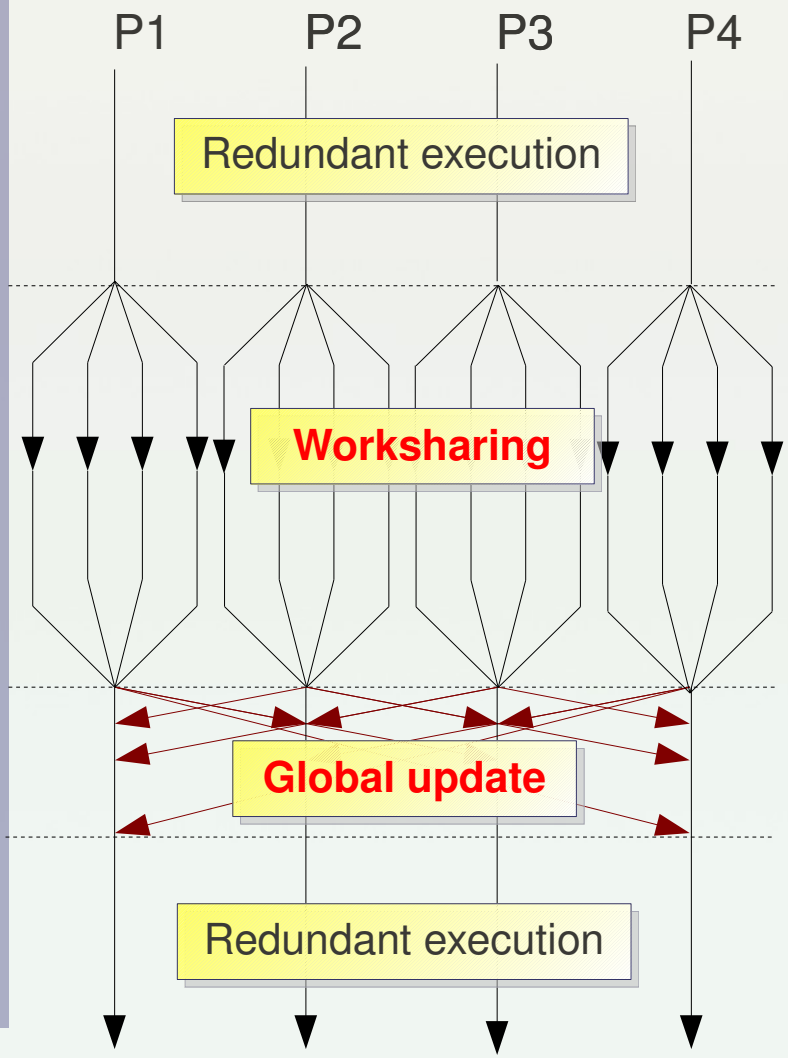
« STEP_COMPILE »: results

V.1.12

```
      SUBROUTINE MATMULT_PARDO20_HYBRID(J, J_L, J_U, I,
N, K, C, A, B)
C       Some declarations
      CALL STEP_GET_SIZE(STEP_LOCAL_COMM_SIZE_)
      CALL STEP_GET_RANK(STEP_LOCAL_COMM_RANK_)

      CALL STEP_COMPUTELOOPSLICES(J_LOW, J_UP, ...)
C     Compute SEND regions for array C
      STEP_SR_C(J_LOW,1,0) = 1
      STEP_SR_C(J_UP,1,0) = N
      ...
C     Where work is done...
      J_LOW = STEP_J_LOOPSLICES(J_LOW, RANK+1)
      J_UP = STEP_J_LOOPSLICES(J_UP,RANK_+1)
      CALL MATMULT_PARDO20_OMP(J, J_LOW, J_UP, I, N, K, C,
A, B)

!$omp master
      CALL STEP_ALLTOALLREGION(C, STEP_SR_C, ...)
!$omp end master
!$omp barrier
      END
```

3 different All2all: NONBLOCKING, BLOCKING1, BLOCKING2

Hybrid execution



Using STEP

V.1.13

Full *tpips* file

```
create myworkspace matmul.f
activate MUST_REGIONS
activate TRANSFORMERS_INTER_FULL
setproperty STEP_DEFAULT_TRANSFORMATION "HYBRID"
setproperty STEP_INSTALL_PATH  " "

apply STEP_DIRECTIVES[%ALL]
apply STEP_ANALYSE[%ALL]
apply STEP_COMPILE[%MAIN]
apply STEP_INSTALL
close
```

"run_step.script" to run STEP on your source files

Get the transformed source in the Src directory

```
$ run_step.script matmul.f

$ ls matmul/matmul.database/Src
Makefile
matmul.f
MATMULT_PARDO20_HYBRID.f
MATMULT_PARDO20_OMP.f
MATMULT_PARDO20.f
STEP.h
steprt_f.h
step_rt/
```

Properties to tune STEP

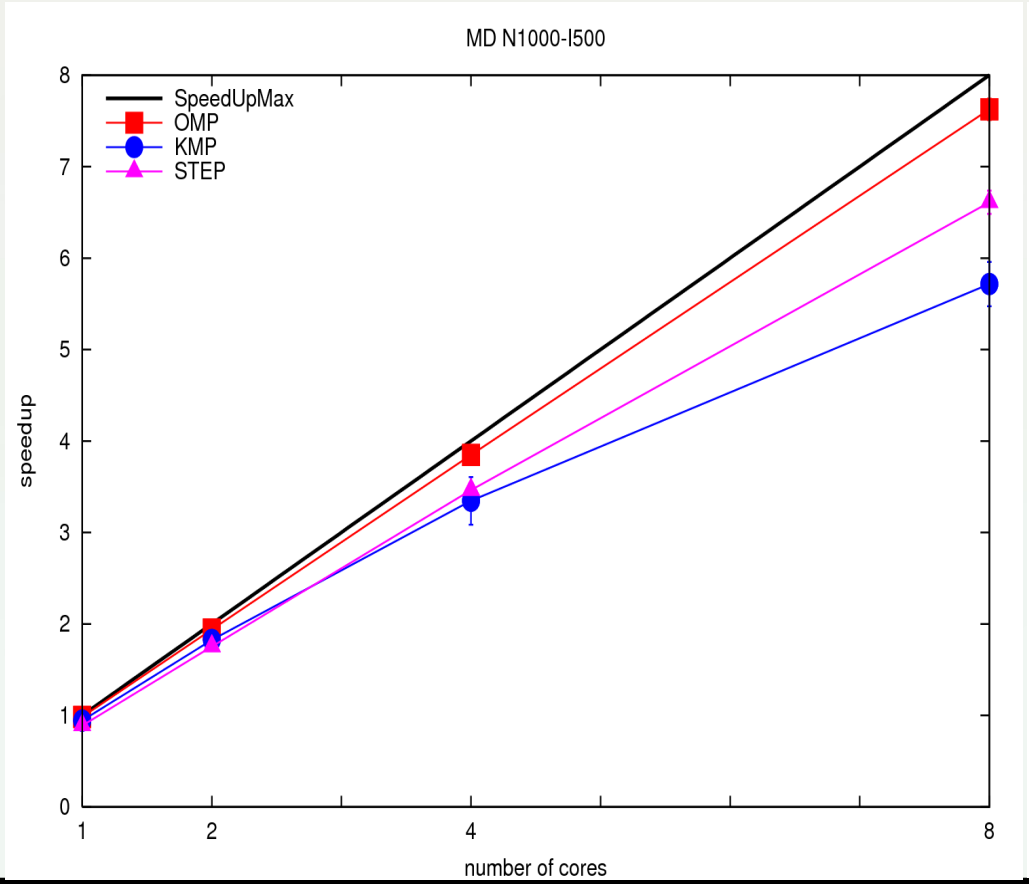
Different available transformations:

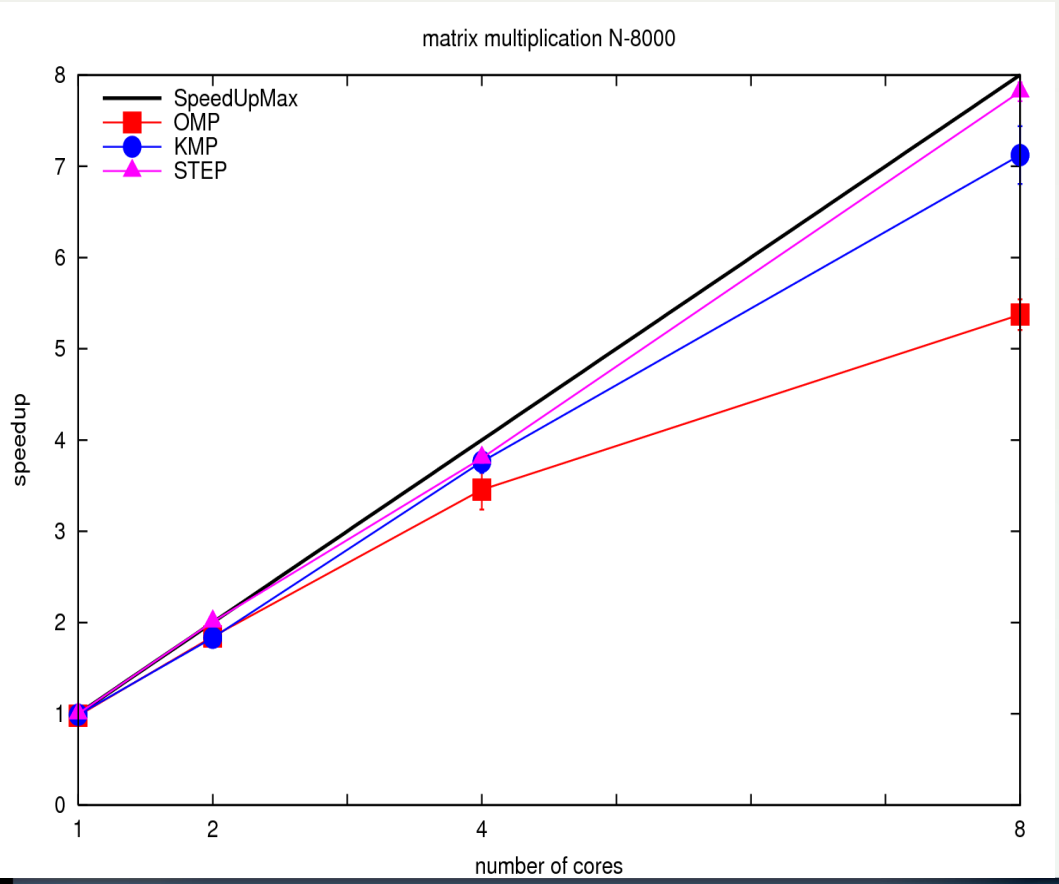
- MPI
- HYBRID
- OMP

Benchmarks: OpenMP / Intel Cluster OpenMP (KMP) / STEP

V.1.14

- **Transformations of some standard benchmarks:**
  - Transformation is correct and run in every case
  - Good performance for coarse-grain parallelism
  - Poor performance with irregular data access patterns





STEP: Conclusion and Perspectives

V.1.15

- **The automatic transformation from OpenMP to MPI is efficient in several cases**
- **... thanks to PIPS interprocedural array regions analyses**
- **Future work**
  - Provide data distribution
  - Generate static communications for partial updates

Par4All for CUDA

V.2.1

Par4All

PIPS Par4All Tutorial

—

CGO 2011

Mehdi Amini[1,2] Béatrice Creusillet[1] Stéphanie Even[3]
Onil Goubier[1] Serge Guelton[3,2] Ronan Keryell[1,3]
Janice Onanian McMahon[1] Grégoire Péan[1] Pierre Villalon[1]

[1]HPC Project

[2]Mines ParisTech/CRI

[3]Institut TÉLÉCOM/TÉLÉCOM Bretagne/HPCAS

2011/04/03

Present motivations

- MOORE's law there are more transistors but they cannot be used at full speed without melting ☹
- Superscalar and cache are less efficient compared to transistor budget
- Chips are too big to be globally synchronous at multi GHz ☹
- Now what cost is to move data and instructions between internal modules, not the computation!
- Huge time and energy cost to move information outside the chip

Parallelism is the only way to go...

Research is just crossing reality!

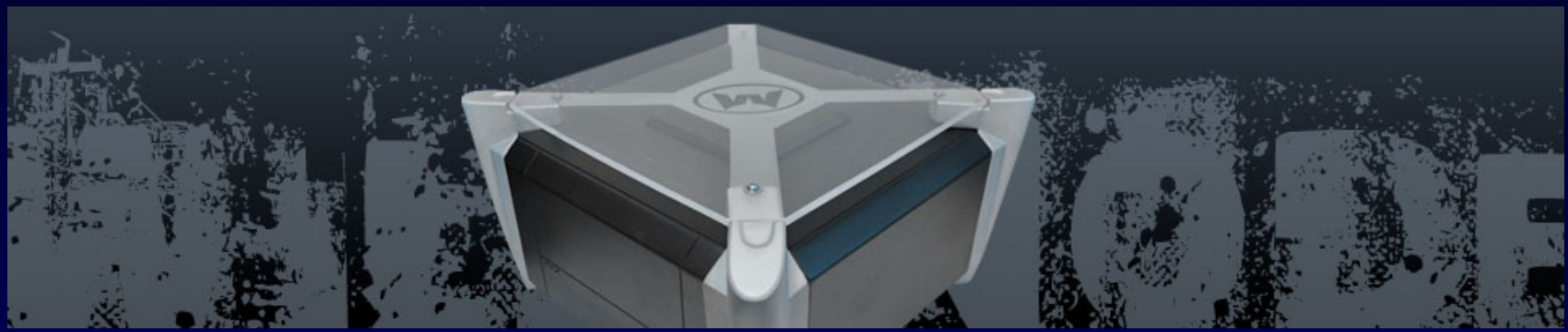
No one size fit all...

Future will be heterogeneous

HPC Project hardware: WildNode from Wild Systems

Through its Wild Systems subsidiary company

- WildNode hardware desktop accelerator
  - Low noise for in-office operation
  - *x*86 manycore
  - nVidia Tesla GPU Computing
  - Linux & Windows



- WildHive
  - Aggregate 2-4 nodes with 2 possible memory views
    - Distributed memory with Ethernet or InfiniBand
    - Virtual shared memory through Linux Kerrighed for single-image system

`http://www.wild-systems.com`

HPC Project software and services

- Parallelize and optimize customer applications, co-branded as a bundle product in a WildNode (e.g. Presagis Stage battle-field simulator, WildCruncher for Scilab//...)
- Acceleration software for the WildNode
  - GPU-accelerated libraries for Scilab/Matlab/Octave/R
  - Transparent execution on the WildNode
- Remote display software for Windows on the WildNode

HPC consulting

- Optimization and parallelization of applications
- *High Performance?...* not only TOP500-class systems: power-efficiency, embedded systems, green computing...
- $\rightsquigarrow$ Embedded system and application design
- Training in parallel programming (OpenMP, MPI, TBB, CUDA, OpenCL...)

The “Software Crisis”

Edsger DIJKSTRA, 1972 Turing Award Lecture, « The Humble Programmer »

“To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.”

`http://en.wikipedia.org/wiki/Software_crisis`

But... it was before parallelism democratization! ☹

Use the Source, Luke...

Hardware is moving quite (too) fast but...

What has survived for 50+ years?

Fortran programs...

What has survived for 40+ years?

IDL, Matlab, Scilab...

What has survived for 30+ years?

C programs, Unix...

- A lot of legacy code could be pushed onto parallel hardware (accelerators) with automatic tools...
- Need automatic tools for source-to-source transformation to leverage existing software tools for a given hardware
- Not as efficient as hand-tuned programs, but quick production phase

We need software tools

- Application development: long-term business $\rightsquigarrow$ long-term commitment in a tool that needs to survive to (too fast) technology change
- HPC Project needs tools for its hardware accelerators (*Wild Nodes* from *Wild Systems*) and to parallelize, port & optimize customer applications

Not reinventing the wheel... No NIH syndrome please!

Want to create your own tool?

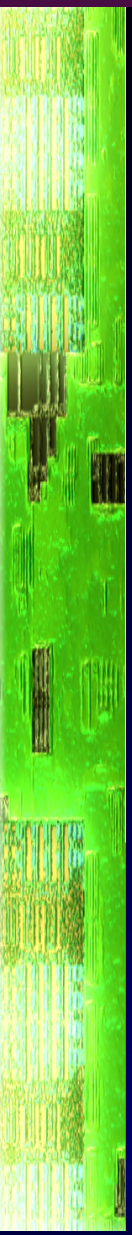
- House-keeping and infrastructure in a compiler is a huge task
- Unreasonable to begin yet another new compiler project...
- Many academic Open Source projects are available...
- ...But customers need products ☺
- ⇝ Integrate your ideas and developments in existing project
- ...or buy one if you can afford (ST with PGI...) ☺
- Some projects to consider
  - Old projects: gcc, PIPS... and many dead ones (SUIF...)
  - But new ones appear too: LLVM, RoseCompiler, Cetus...

Par4All

- ⇝ Funding an initiative to industrialize Open Source tools
- PIPS is the first project to enter the Par4All initiative

`http://www.par4all.org`

PIPS



- PIPS (Interprocedural Parallelizer of Scientific Programs): Open Source project from Mines ParisTech... 23-year old! ☺
- Funded by many people (French DoD, Industry & Research Departments, University, CEA, IFP, Onera, ANR (French NSF), European projects, regional research clusters...)
- One of the projects that introduced polytope model-based compilation
- ≈ 456 KLOC according to David A. Wheeler's `SLOCCount`
- ... but modular and sensible approach to pass through the years
  - ≈300 phases (parsers, analyzers, transformations, optimizers, parallelizers, code generators, pretty-printers...) that can be combined for the right purpose
  - Polytope lattice (sparse linear algebra) used for semantics analysis, transformations, code generation... to deal with big programs, not only loop-nests

PIPS (II)

- NewGen object description language for language-agnostic automatic generation of methods, persistence, object introspection, visitors, accessors, constructors, XML marshaling for interfacing with external tools...
- Interprocedural *à la* `make` engine to chain the phases as needed. Lazy construction of resources
- On-going efforts to extend the semantics analysis for C

- Around 15 programmers currently developing in PIPS (Mines ParisTech, HPC Project, IT SudParis, TÉLÉCOM Bretagne, RPI) with public `svn`, Trac, `git`, mailing lists, IRC, Plone, Skype... and use it for many projects
- But still...
  - Huge need of documentation (even if PIPS uses literate programming...)
  - Need of industrialization
  - Need further communication to increase community size

Current PIPS usage

- Automatic parallelization (Par4All C & Fortran to OpenMP)
- Distributed memory computing with OpenMP-to-MPI translation [STEP project]
- Generic vectorization for SIMD instructions (SSE, VMX, Neon, CUDA, OpenCL...) (SAC project) [SCALOPES]
- Parallelization for embedded systems [SCALOPES]
- Compilation for hardware accelerators (Ter@PIX, SPoC, SIMD, FPGA...) [FREIA, SCALOPES]
- High-level hardware accelerators synthesis generation for FPGA [PHRASE, CoMap]
- Reverse engineering & decompiler (reconstruction from binary to C)
- Genetic algorithm-based optimization [Luxembourg university+TB]
- Code instrumentation for performance measures
- GPU with CUDA & OpenCL [TransMedi@, FREIA, OpenGPU]

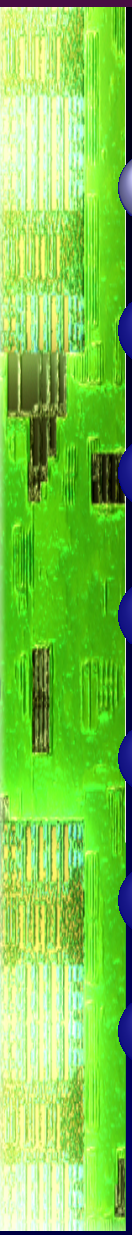
Par4All usage

Generate from sequential C, Fortran & Scilab code

- OpenMP for SMP
- CUDA for nVidia GPU
- SCMP task programs for SCMP machine from CEA
- OpenCL for GPU & ST Platform 2012 (on-going)

Outline

1 Par4All global infrastructure
2 OpenMP code generation
3 GPU code generation
4 Code generation for SCMP
5 Scilab compilation
6 Results
7 Conclusion



Par4All ≡ PyPS scripting in the backstage (I)

- PIPS is a great tool-box to do source-to-source compilation
- ...but not really usable by λ end-user ☹
- $\rightsquigarrow$ Development of Par4All
- Add a user compiler-like infrastructure

$\rightsquigarrow$ `p4a` script as simple as

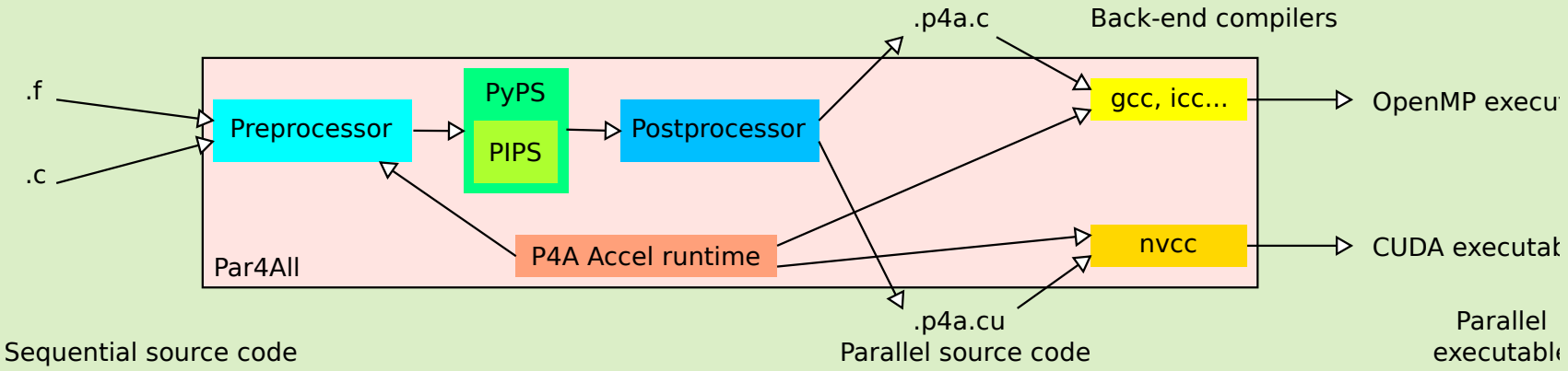
- `p4a --openmp toto.c -o toto`
- `p4a --cuda toto.c -o toto -lm`

- Be multi-target
- Apply some adaptative transformations
- Up to now PIPS was scripted with a special shell-like language: `tpips`
- Not enough powerful (not a programming language)
- Develop a SWIG Python interface to PIPS phases and interface

Par4All ≡ PyPS scripting in the backstage (II)

- All the power of a widely spread real language
- Automate with introspection through the compilation flow
- Easy to add any glue, pre-/post-processing to generate target code

Overview



- Invoke PIPS transformations
  - With different recipes according to generated stuff
  - Special treatments on kernels...
- Compilation and linking infrastructure: can use `gcc`, `icc`, `nvcc`, `nvcc+gcc`, `nvcc+icc`

Par4All ≡ PyPS scripting in the backstage (III)

- House keeping code
- Fundamental: colorizing and filtering some PIPS output, running cursor... ☺

Outline

1 Par4All global infrastructure
2 OpenMP code generation
3 GPU code generation
4 Code generation for SCMP
5 Scilab compilation
6 Results
7 Conclusion

Parallelization to OpenMP (I)

- The easy way... Already in PIPS
- Used to bootstrap the start-up with stage-0 investors ☺
- Indeed, we used only `bash`-generated `tpips` at this time (2008, no PyPS yet), needed a lot of bug squashing on C support in PIPS...
- Now in `src/simple_tools/p4a_process.py`, function `process()`

```
# First apply some generic parallelization:
processor.parallelize(fine = input.fine,
                      apply_phases_before = input.apply_phases['abp'],
                      apply_phases_after = input.apply_phases['aap'])
[...]
if input.openmp and not input.accel:
    # Parallelize the code in an OpenMP way:
    processor.ompify(apply_phases_before = input.apply_phases['abo'],
                     apply_phases_after = input.apply_phases['aao'])

# Write the output files.
output.files = processor.save(input.output_dir,
                              input.output_prefix,
                              input.output_suffix)
```

Parallelization to OpenMP (II)

- src/simple_tools/p4a_process.py, function p4a_processor::parallelize()

```
1       def parallelize(self, fine = False, filter_select = None,
2                       filter_exclude = None, apply_phases_before = [], ap
            """Apply transformations to parallelize the code in the workspa
4   ce"""
            all_modules = self.filter_modules(filter_select, filter_exclude
6
            for ph in apply_phases_before:
8               # Apply requested phases before parallelization:
                    getattr(all_modules, ph)()
10
            # Try to privatize all the scalar variables in loops:
12          all_modules.privatize_module()

14          if fine:
                # Use a fine-grain parallelization à la Allen & Kennedy:
16              all_modules.internalize_parallel_code(concurrent=True)
            else:
18              # Use a coarse-grain parallelization with regions:
                all_modules.coarse_grain_parallelization(concurrent=True)
20
            for ph in apply_phases_after:
```

Parallelization to OpenMP (III)

```
22                  # Apply requested phases after parallelization:
                    getattr(all_modules, ph)()
```

- Subliminal message to PIPS/Par4All developers: write clear code with good comments since it can end up verbatim into presentations like this ☺

OpenMP output sample (I)

```
!$omp parallel do private(I, K, X)
C multiply the two square matrices of ones
      DO J = 1, N
0016
!$omp parallel do private(K, X)
         DO I = 1, N
0017
            X = 0
0018
!$omp parallel do reduction(+:X)
            DO K = 1, N
0019
               X = X+A(I,K)*B(K,J)
0020
            ENDDO
!$omp end parallel do
            C(I,J) = X
0022
         ENDDO
!$omp end parallel do
      ENDDO
!$omp end parallel do
```

Outline

1. Par4All global infrastructure
2. OpenMP code generation
3. GPU code generation
4. Code generation for SCMP
5. Scilab compilation
6. Results
7. Conclusion

Basic GPU execution model

A sequential program on a host launches computational-intensive kernels on a GPU

- Allocate storage on the GPU
- Copy-in data from the host to the GPU
- Launch the kernel on the GPU
- The host waits...
- Copy-out the results from the GPU to the host
- Deallocate the storage on the GPU

Generic scheme for other heterogeneous accelerators too

Challenges in automatic GPU code generation

- Find parallel kernels
- Improve data reuse inside kernels to have better compute intensity (even if the memory bandwidth is quite higher than on a CPU...)
- Access the memory in a GPU-friendly way (to coalesce memory accesses)
- Take advantage of complex memory hierarchy that make the GPU fast (shared memory, cached texture memory, registers...)
- Reduce the copy-in and copy-out transfers that pile up on the PCIe
- Reduce memory usage in the GPU (no swap there, yet...)
- Limit inter-block synchronizations
- Overlap computations and GPU-CPU transfers (via streams)

Automatic parallelization (I)

Most fundamental for a parallel execution

Finding parallelism!

Several parallelization algorithms are available in PIPS

- For example classical Allen & Kennedy use loop distribution more vector-oriented than kernel-oriented  (or need later loop-fusion)
- Coarse grain parallelization based on the independence of array regions used by different loop iterations
  - Currently used because generates GPU-friendly coarse-grain parallelism
  - Accept complex control code without *if-conversion*

```
1 # First apply some generic parallelization:
2 processor.parallelize(fine = input.fine,
                        apply_phases_before = input.apply_phases['abp'],
4                       apply_phases_after = input.apply_phases['aap'])
```

Automatic parallelization (II)

Then GPUification can begin

```
1 if input.accel:
2     # Generate code for a GPU-like accelerator. Note that we can
3     # have an OpenMP implementation of it if OpenMP option is set
4     # too:
5     processor.gpuify(apply_phases_kernel_after = input.apply_phases['akag']
6                      apply_phases_kernel_launcher = input.apply_phases['
7                      apply_phases_wrapper = input.apply_phases['awg'],
8                      apply_phases_after = input.apply_phases['aag'])
```

Outlining (I)

Parallel code $\rightsquigarrow$ Kernel code on GPU

- Need to extract parallel source code into kernel source code: outlining of parallel loop-nests
- Before:

```
1  #pragma omp parallel for private(j)
2    for(i = 1;i <= 499; i++)
       for(j = 1;j <= 499; j++) {
4        save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]
                          + space[i][j - 1] + space[i][j + 1]);
6      }
```

Outlining (II)

- After:

```
1  p4a_kernel_launcher_0(space, save);
2  [...]
   void p4a_kernel_launcher_0(float_t space[SIZE][SIZE],
4                             float_t save[SIZE][SIZE]) {
     for(i = 1; i <= 499; i += 1)
6        for(j = 1; j <= 499; j += 1)
           p4a_kernel_0(i, j, save, space);
8  }
   [...]
10 void p4a_kernel_0(float_t space[SIZE][SIZE],
                     float_t save[SIZE][SIZE],
12                   int i,
                     int j) {
14   save[i][j] = 0.25*(space[i-1][j]+space[i+1][j]
                        +space[i][j-1]+space[i][j+1]);
16 }
```

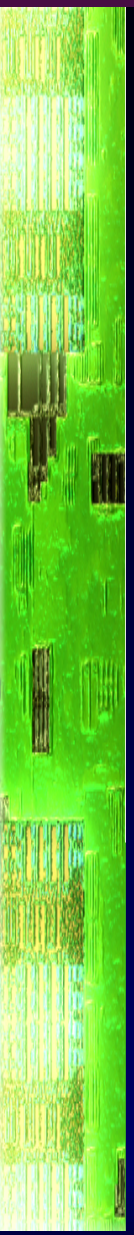
Done with:

Outlining (III)

```
# First, only generate the launchers to work on them later. They are
# generated by outlining all the parallel loops. If in the fortran case
# we want the launcher to be wrapped in an independant fortran function
# to ease future post processing.
all_modules.gpu_ify(GPU_USE_WRAPPER = False,
                    GPU_USE_KERNEL = False,
                    GPU_USE_FORTRAN_WRAPPER = self.fortran,
                    GPU_USE_LAUNCHER = True,
                    #OUTLINE_INDEPENDENT_COMPILATION_UNIT = self.c99,
                    concurrent=True)
```

From array regions to GPU memory allocation (I)

- Memory accesses are summed up for each statement as *regions* for array accesses: integer polytope lattice
- There are regions for write access and regions for read access
- The regions can be exact if PIPS can prove that only these points are accessed, or they can be inexact, if PIPS can only find an over-approximation of what is really accessed



From array regions to GPU memory allocation (///)

Example

```
1 for(i = 0; i <= n-1; i += 1)
2   for(j = i; j <= n-1; j += 1)
      h_A[i][j] = 1;
```

can be decorated by PIPS with write array regions as:

```
1 //  <h_A[PHI1][PHI2]-W-EXACT-{0<=PHI1, PHI2+1<=n, PHI1<=PHI2}>
    for(i = 0; i <= n-1; i += 1)
3 //  <h_A[PHI1][PHI2]-W-EXACT-{PHI1==i, i<=PHI2, PHI2+1<=n, 0<=i}>
      for(j = i; j <= n-1; j += 1)
5 //  <h_A[PHI1][PHI2]-W-EXACT-{PHI1==i, PHI2==j, 0<=i, i<=j, 1+j<=n}>
        h_A[i][j] = 1;
```

- These read/write regions for a kernel are used to allocate with a `cudaMalloc()` in the host code the memory used inside a kernel and to deallocate it later with a `cudaFree()`

Communication generation (I)

Conservative approach to generate communications

- Associate any GPU memory allocation with a copy-in to keep its value in sync with the host code
- Associate any GPU memory deallocation with a copy-out to keep the host code in sync with the updated values on the GPU

-  But a kernel could use an array as a local (private) array
- ...PIPS does have many privatization phases ☺
-  But a kernel could initialize an array, or use the initial values without writing into it or use/touch only a part of it or...

Communication generation (II)

More subtle approach

PIPS gives 2 very interesting region types for this purpose

- In-region abstracts what really needed by a statement
- Out-region abstracts what really produced by a statement to be used later elsewhere

- In-Out regions can directly be translated with CUDA into
  - copy-in

```
1  cudaMemcpy(accel_address, host_address,
2             size, cudaMemcpyHostToDevice)
```

  - copy-out

```
1  cudaMemcpy(host_address, accel_address,
2             size, cudaMemcpyDeviceToHost)
```

Communication generation (III)

```
1 # Add communication around all the call site of the kernels. Since
2 # the code has been outlined, any non local effect is no longer an
  # issue:
4 kernel_launchers.kernel_load_store(concurrent=True,
                                     ISOLATE_STATEMENT_EVEN_NON_LOCAL = True
6                                    )
```

Loop normalization (I)

- Hardware accelerators use fixed iteration space (thread index starting from 0...)
- Parallel loops: more general iteration space
- Loop normalization

Before

```
1 for(i = 1;i < SIZE - 1; i++)
2   for(j = 1;j < SIZE - 1; j++) {
      save[i][j] = 0.25*(space[i - 1][j] + space[i + 1][j]
4                        + space[i][j - 1] + space[i][j + 1]);
    }
```

Loop normalization (II)

After

```
1  for(i = 0;i < SIZE - 2; i++)
     for(j = 0;j < SIZE - 2; j++) {
3      save[i+1][j+1] = 0.25*(space[i][j + 1] + space[i + 2][j + 1]
                            + space[i + 1][j] + space[i + 1][j + 2]);
5    }
```

```
1  # Select kernel launchers by using the fact that all the generated
   # functions have their names beginning with the launcher prefix:
3  launcher_prefix = self.get_launcher_prefix ()
   kernel_launcher_filter_re = re.compile(launcher_prefix + "_.*[^!]$")
5  kernel_launchers = self.workspace.filter(lambda m:
                          kernel_launcher_filter_re.match(m.name))
7
   # Normalize all loops in kernels to suit hardware iteration spaces:
9  kernel_launchers.loop_normalize(
       # Loop normalize to be GPU friendly, even if the step is already 1:
11     LOOP_NORMALIZE_ONE_INCREMENT = True,
       # Arrays start at 0 in C, 1 in Fortran so the iteration loops:
13     LOOP_NORMALIZE_LOWER_BOUND = self.fortran == True ? 1 : 0,
       # It is legal in the following by construction (...Hmmm to verify)
```

Loop normalization (III)

```
15      LOOP_NORMALIZE_SKIP_INDEX_SIDE_EFFECT = True,
        concurrent=True)
```

From preconditions to iteration clamping (I)

- Parallel loop nests are compiled into a CUDA kernel wrapper launch
- The kernel wrapper itself gets its virtual processor index with some `blockIdx.x*blockDim.x + threadIdx.x`
- Since only full blocks of threads are executed, if the number of iterations in a given dimension is not a multiple of the `blockDim`, there are incomplete blocks ☹
- An incomplete block means that some index overrun occurs if all the threads of the block are executed 

From preconditions to iteration clamping (II)

- So we need to generate code such as

```
1  void p4a_kernel_wrapper_0(int k, int l,...)
2  {
     k = blockIdx.x*blockDim.x + threadIdx.x;
4    l = blockIdx.y*blockDim.y + threadIdx.y;
     if (k >= 0 && k <= M - 1 && l >= 0 && l <= M - 1)
6      kernel(k, l, ...);
   }
```

  But how to insert these guards?

- The good news is that PIPS owns *preconditions* that are predicates on integer variables. Preconditions at entry of the kernel are:

```
1  // P(i,j,k,l) {0<=k, k<=63, 0<=l, l<=63}
```

- Guard $\equiv$ directly translation in C of preconditions on loop indices that are GPU thread indices

```
1  # Add iteration space decorations and insert iteration clamping
   # into the launchers onto the outer parallel loop nests:
3  kernel_launchers.gpu_loop_nest_annotate(concurrent=True)
```

Complexity analysis (I)

- Launching a GPU kernel is expensive
  - so we need to launch only kernels with a significant speed-up (launching overhead, memory CPU-GPU copy overhead...)
- Some systems use `#pragma` to give a go/no-go information to parallel execution

```
1 #pragma omp parallel if(size>100)
```

- ∃ phase in PIPS to symbolically estimate complexity of statements
- Based on preconditions
- Use a SuperSparc2 model from the '90s... ☺
- Can be changed, but precise enough to have a coarse go/no-go information
- To be refined: use memory usage complexity to have information about memory reuse (even a big kernel could be more efficient on a CPU if there is a good cache use)

Optimized reduction generation

- Reduction are common patterns that need special care to be correctly parallelized

$$s = \sum_{i=0}^{N} x_i$$

- Reduction detection already implemented in PIPS
- Efficient computation on GPU needs to create local reduction trees in the thread-blocks
  - Use existing libraries but may need several kernels?
  - Inline reduction code?
- Not yet implemented in Par4All

Communication optimization

- Naive approach : load/compute/store
- Useless communications if a data on GPU is not used on host between 2 kernels... ☹
- ⇝ Use static interprocedural data-flow communications
  - Fuse various GPU arrays : remove GPU (de)allocation
  - Remove redundant communications

⇝ New `p4a --com-optimization` option

Fortran to C-based GPU languages

- Fortran 77 parser available in PIPS
- CUDA & OpenCL are C++/C99 with some restrictions on the GPU-executed parts
- Need a Fortran to C translator (`f2c`...)?
- Only one internal representation is used in PIPS
  - Use the Fortran parser
  - Use the... C pretty-printer!
- But the IO Fortran library is complex to use... and to translate
  - If you have IO instructions in a Fortran loop-nest, it is not parallelized anyway because of sequential side effects ☺
  - So keep the Fortran output everywhere but in the parallel CUDA kernels
  - Apply a memory access transposition phase `a(i,j)` ⇝ `a[j-1][i-1]` inside the kernels to be pretty-printed as C
- Compile and link C GPU kernel parts + Fortran main parts
- Quite harder than expected... Use Fortran 2003 for C interfaces...

Par4All Accel runtime (I)

- CUDA or OpenCL can not be directly represented in the internal representation (IR, abstract syntax tree) such as `__device__` or `<<< >>>`
- PIPS motto: keep the IR as simple as possible by design
- Use some calls to intrinsics functions that can be represented directly
- Intrinsics functions are implemented with (macro-)functions
  - `p4a_accel.h` has indeed currently 2 implementations
    - `p4a_accel-CUDA.h` than can be compiled with CUDA for nVidia GPU execution or emulation on CPU
    - `p4a_accel-OpenMP.h` that can be compiled with an OpenMP compiler for simulation on a (multicore) CPU
- Add CUDA support for complex numbers

Par4All Accel runtime (II)

- On-going support of OpenCL written in C/CPP/C++
- Can be used to simplify manual programming too (OpenCL...)
  - Manual radar electromagnetic simulation code @TB
  - One code target CUDA/OpenCL/OpenMP
- OpenMP emulation for almost free
  - Use Valgrind to debug GPU-like and communication code !
  - May even improve performance compared to native OpenMP generation because of memory layout change

Working around CUDA limitations

- CUDA is not based on C99 but rather on C89 + few C++ extensions
- Some PIPS generated code from C99 user code may not compile ☹
- ⇝ Use some array linearization at some places

```
if (self.fortran == False):
    kernels.linearize_array(LINEARIZE_ARRAY_USE_POINTERS=True,LINEARIZE_ARR
    wrappers.linearize_array(LINEARIZE_ARRAY_USE_POINTERS=True,LINEARIZE_AR
else:
    kernels.linearize_array_fortran(LINEARIZE_ARRAY_USE_POINTERS=False,LINE
    wrappers.linearize_array_fortran(LINEARIZE_ARRAY_USE_POINTERS=False,LIN
```

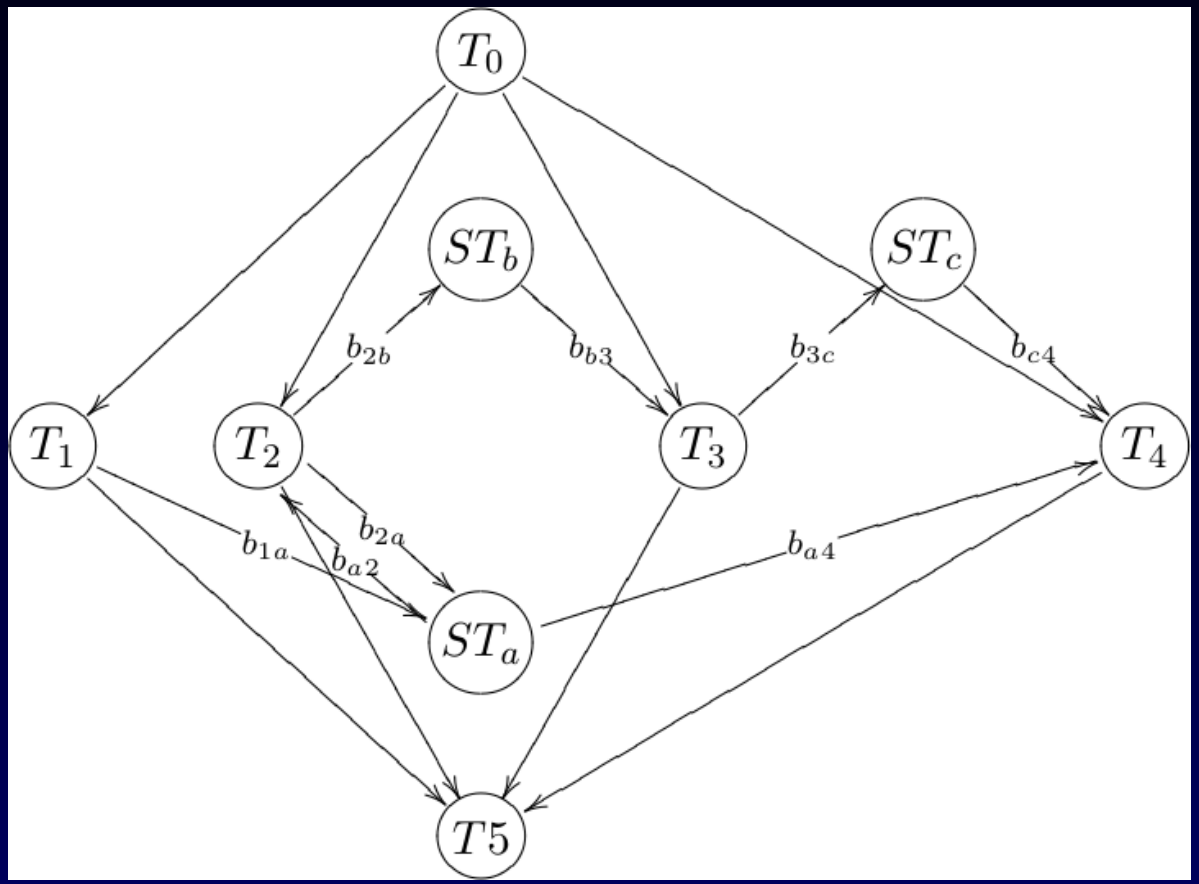
Outline

1 Par4All global infrastructure
2 OpenMP code generation
3 GPU code generation
4 Code generation for SCMP
5 Scilab compilation
6 Results
7 Conclusion

SCMP computer (I)

- Embedded accelerator developed at French CEA
  - Task graph oriented parallel multiprocessor
  - Hardware task graph scheduler
  - Synchronizations
  - Communication through memory page sharing
- Generating code from THALES (TCF) GSM sensing application in SCALOPES European project
- Reuse output of PIPS GPU phases + specific phases
  - SCMP code with tasks
  - SCMP task descriptor files
- Adapted Par4All Accel run-time

SCMP tasks



In general case, different tasks can produce data in unpredictable way: use helper data server tasks to deal with coherency when several producers

SCMP task code (before/after)

```
int main() {
  int i, t, a[20], b[20];
  for (t=0; t < 100; t++)
  {
kernel_tasks_1:
    for(i=0; i <10; i++)
      a[i] = i+t;

kernel_tasks_2:
    for(i=10; i <20; i++)
      a[i] = 2*i+t;

kernel_tasks_3:
    for(i=10; i <20; i++)
      printf("a[%d] = %d\n",
             i, a[i]);
  }
  return(0);
}
```

```
int main() {
 P4A_scmp_reset();
 int i, t, a[20], b[20];
 for(t = 0; t <= 99; t += 1) {
  [...]
  {
   //PIPS generated variable
   int (*P4A__a__1)[10] = (int (*)[10]) 0;
   P4A_scmp_malloc((void **) &P4A__a__1,
     sizeof(int)*10, P4A__a__1_id,
     P4A__a__1_prod_p || P4A__a__1_cons_p, P4A__
   if (scmp_task_2_p)
     for(i = 10; i <= 19; i += 1)
       (*P4A__a__1)[i-10] = 2*i+t;
     P4A_copy_from_accel_1d(sizeof(int), 20, 10,
       P4A_sesam_server_a_p ? &a[0] : NULL, *P4A
       P4A__a__1_id, P4A__a__1_prod_p || P4A__a_
     P4A_scmp_dealloc(P4A__a__1, P4A__a__1_id,
       P4A__a__1_prod_p || P4A__a__1_cons_p, P4A
  }
 [...]
 }
 return(ev_T004);
}
```

Performance of GSM sensing on SCMP

- Speed-up on 4 PE SCMP:
  - $\times$2.35 with manual parallelization by SCMP team
  - $\times$1.86 with automatic Par4All parallelization

Outline

1. Par4All global infrastructure
2. OpenMP code generation
3. GPU code generation
4. Code generation for SCMP
5. Scilab compilation
6. Results
7. Conclusion

Scilab language

- Interpreted scientific language widely used like Matlab
- Free software
- Roots in free version of Matlab from the 80's
- Dynamic typing (scalars, vectors, (hyper)matrices, strings...)
- Many scientific functions, graphics...
- Double precision everywhere, even for loop indices (now)
- Slow because everything decided at runtime, garbage collecting
  - Implicit loops around each vector expression
    - Huge memory bandwidth used
    - Cache thrashing
    - Redundant control flow
- Strong commitment to develop Scilab through Scilab Enterprise, backed by a big user community, INRIA...
- HPC Project WildNode appliance with Scilab parallelization
- Reuse Par4All infrastructure to parallelize the code

Scilab & Matlab (I)

- Scilab/Matlab input : *sequential* or array syntax
- Compilation to C code
  - Our COLD compiler is *not* Open Source
  - There is such Open Source compiler from hArtes European project written in... Scilab ☺
- Parallelization of the generated C code
- Type inference to guess (crazy ☹) semantics
  - Heuristic: first encountered type is forever
- May get speedup $>$ 1000 ☺
- WildCruncher product from HPC Project: *x*86+GPU appliance with nice interface
  - Scilab — mathematical model & simulation
  - Par4All — automatic parallelization
  - //Geometry — polynomial-based 3D rendering & modelling

Outline

1 Par4All global infrastructure
2 OpenMP code generation
3 GPU code generation
4 Code generation for SCMP
5 Scilab compilation
6 Results
7 Conclusion

Hyantes (I)

- Geographical application: library to compute neighbourhood population potential with scale control
- WildNode with 2 Intel Xeon X5670 @ 2.93GHz (12 cores) and a nVidia Tesla C2050 (Fermi), Linux/Ubuntu 10.04, gcc 4.4.3, CUDA 3.1
  - Sequential execution time on CPU: 30.355s
  - OpenMP parallel execution time on CPUs: 3.859s, speed-up: 7.87
  - CUDA parallel execution time on GPU: 0.441s, speed-up: 68.8
- With single precision on a HP EliteBook 8730w laptop (with an Intel Core2 Extreme Q9300 @ 2.53GHz (4 cores) and a nVidia GPU Quadro FX 3700M (16 multiprocessors, 128 cores, architecture 1.1)) with Linux/Debian/sid, gcc 4.4.5, CUDA 3.1:
  - Sequential execution time on CPU: 34.7s
  - OpenMP parallel execution time on CPUs: 13.7s, speed-up: 2.53
  - OpenMP emulation of GPU on CPUs: 9.7s, speed-up: 3.6
  - CUDA parallel execution time on GPU: 1.57s, speed-up: 24.2

Hyantes (II)

Original main C kernel:

```
1  void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range,
2    town pt[rangex][rangey], town t[nb])
3  {
4      size_t i,j,k;
5
6      fprintf(stderr,"begin computation ...\n");
7
8      for(i=0;i<rangex;i++)
9          for(j=0;j<rangey;j++) {
10             pt[i][j].latitude =(xmin+step*i)*180/M_PI;
11             pt[i][j].longitude =(ymin+step*j)*180/M_PI;
12             pt[i][j].stock =0.;
13             for(k=0;k<nb;k++) {
14                 data_t tmp = 6368.* acos(cos(xmin+step*i)*cos( t[k].latitude )
15                      * cos((ymin+step*j)-t[k].longitude)
16                      + sin(xmin+step*i)*sin(t[k].latitude));
17                 if( tmp < range )
18                     pt[i][j].stock += t[k].stock  / (1 + tmp) ;
19             }
20         }
21     fprintf(stderr,"end computation ...\n");
22 }
```

Example given in `par4all.org` distribution

Hyantes (III)

OpenMP code:

```
1  void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, d
2  {
3     size_t i, j, k;
4
5     fprintf(stderr, "begin computation ...\n");
6
7  #pragma omp parallel for private(k, j)
8     for(i = 0; i <= 289; i += 1)
9        for(j = 0; j <= 298; j += 1) {
10          pt[i][j].latitude = (xmin+step*i)*180/3.14159265358979323846;
11          pt[i][j].longitude = (ymin+step*j)*180/3.14159265358979323846;
12          pt[i][j].stock = 0.;
13          for(k = 0; k <= 2877; k += 1) {
14             data_t tmp = 6368.*acos(cos(xmin+step*i)*cos(t[k].latitude)*cos
15             if (tmp<range)
16                pt[i][j].stock += t[k].stock/(1+tmp);
17          }
18       }
19    fprintf(stderr, "end computation ...\n");
20 }
21 void display(town pt[290][299])
22 {
```

Hyantes (IV)

```
    size_t i, j;
    for(i = 0; i <= 289; i += 1) {
        for(j = 0; j <= 298; j += 1)
            printf("%lf %lf %lf\n", pt[i][j].latitude, pt[i][j].longitude, pt[
        printf("\n");
    }
}
```

Hyantes (V)

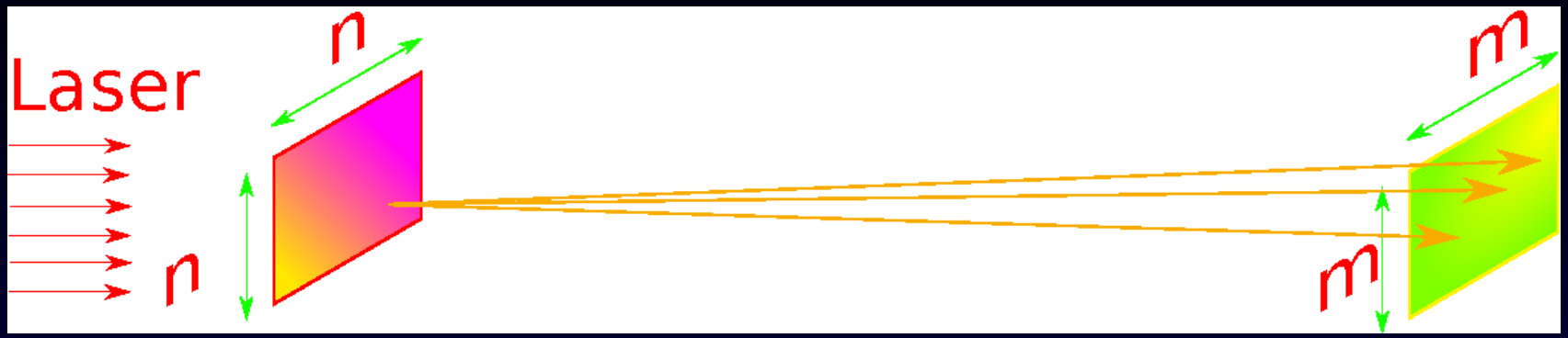
Generated GPU code:

```
1  void run(data_t xmin, data_t ymin, data_t xmax, data_t ymax, data_t step, data_t range,
     town pt[290][299], town t[2878])
3  {
     size_t i, j, k;
5    //PIPS generated variable
     town (*P_0)[2878] = (town (*)[2878]) 0, (*P_1)[290][299] = (town (*)[290][299]) 0;
7
     fprintf(stderr, "begin computation ...\n");
9    P4A_accel_malloc(&P_1, sizeof(town[290][299])-1+1);
     P4A_accel_malloc(&P_0, sizeof(town[2878])-1+1);
11   P4A_copy_to_accel(pt, *P_1, sizeof(town[290][299])-1+1);
     P4A_copy_to_accel(t, *P_0, sizeof(town[2878])-1+1);
13
     p4a_kernel_launcher_0(*P_1, range, step, *P_0, xmin, ymin);
15   P4A_copy_from_accel(pt, *P_1, sizeof(town[290][299])-1+1);
     P4A_accel_free(*P_1);
17   P4A_accel_free(*P_0);
     fprintf(stderr, "end computation ...\n");
19 }

21 void p4a_kernel_launcher_0(town pt[290][299], data_t range, data_t step, town t[2878],
     data_t xmin, data_t ymin)
23 {
     //PIPS generated variable
25   size_t i, j, k;
     P4A_call_accel_kernel_2d(p4a_kernel_wrapper_0, 290,299, i, j, pt, range,
27                            step, t, xmin, ymin);
29 }

P4A_accel_kernel_wrapper void p4a_kernel_wrapper_0(size_t i, size_t j, town pt[290][299],
```

Hyantes (VI)

```
31    data_t range, data_t step, town t[2878], data_t xmin, data_t ymin)
{
33    // Index has been replaced by P4A_vp_0:
      i = P4A_vp_0;
35    // Index has been replaced by P4A_vp_1:
      j = P4A_vp_1;
37    // Loop nest P4A end
      p4a_kernel_0(i, j, &pt[0][0], range, step, &t[0], xmin, ymin);
39  }

41  P4A_accel_kernel void p4a_kernel_0(size_t i, size_t j, town *pt, data_t range,
      data_t step, town *t, data_t xmin, data_t ymin)
43  {
      //PIPS generated variable
45    size_t k;
      // Loop nest P4A end
47    if (i<=289&&j<=298) {
        pt[299*i+j].latitude = (xmin+step*i)*180/3.14159265358979323846;
49      pt[299*i+j].longitude = (ymin+step*j)*180/3.14159265358979323846;
        pt[299*i+j].stock = 0.;
51      for(k = 0; k <= 2877; k += 1) {
          data_t tmp = 6368.*acos(cos(xmin+step*i)*cos((*(t+k)).latitude)*cos(ymin+step*j
53              -(*(t+k)).longitude)+sin(xmin+step*i)*sin((*(t+k)).latitude));
          if (tmp<range)
55          pt[299*i+j].stock += t[k].stock/(1+tmp);
        }
57    }
  }
```

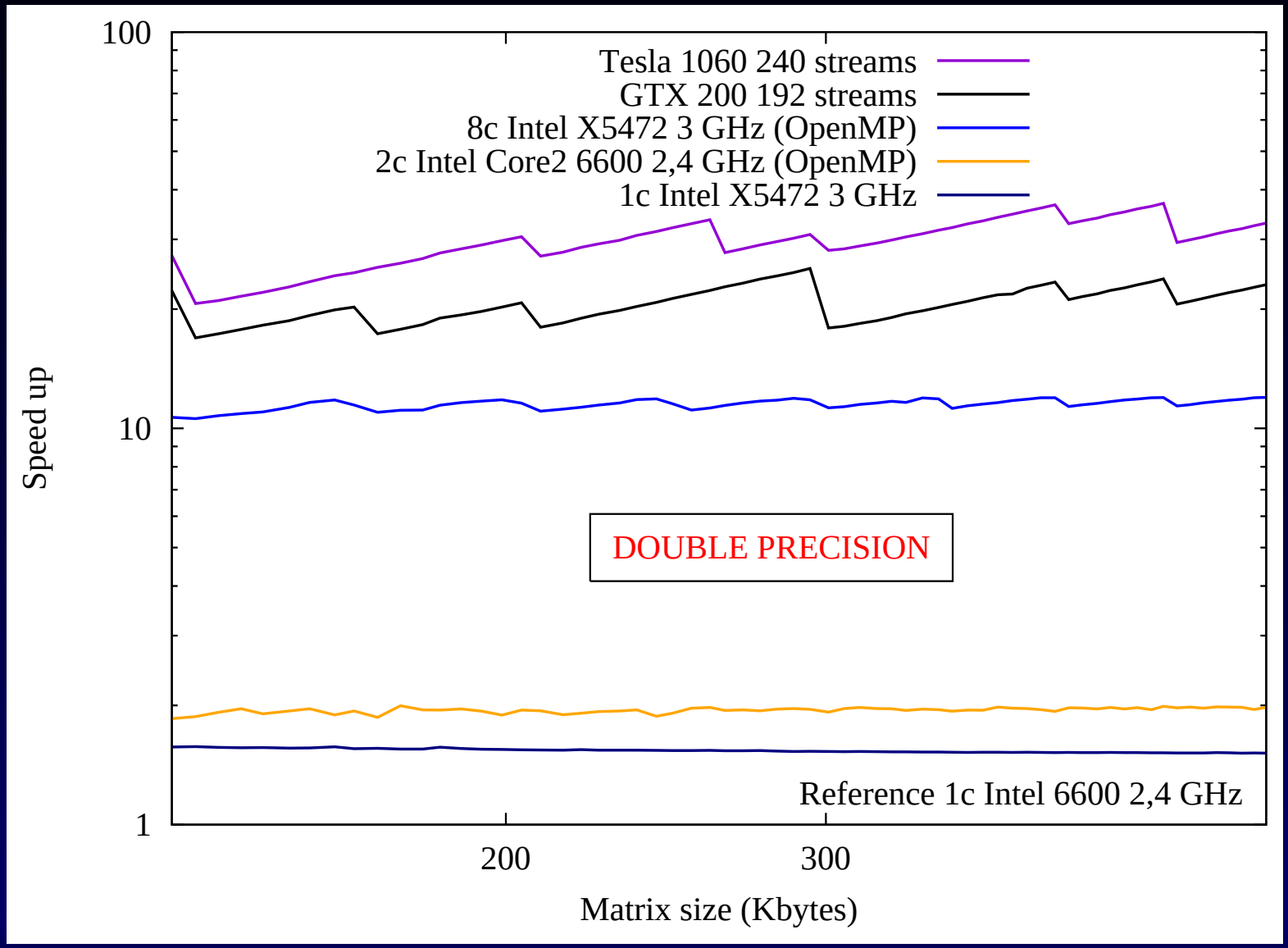
Results on a customer application



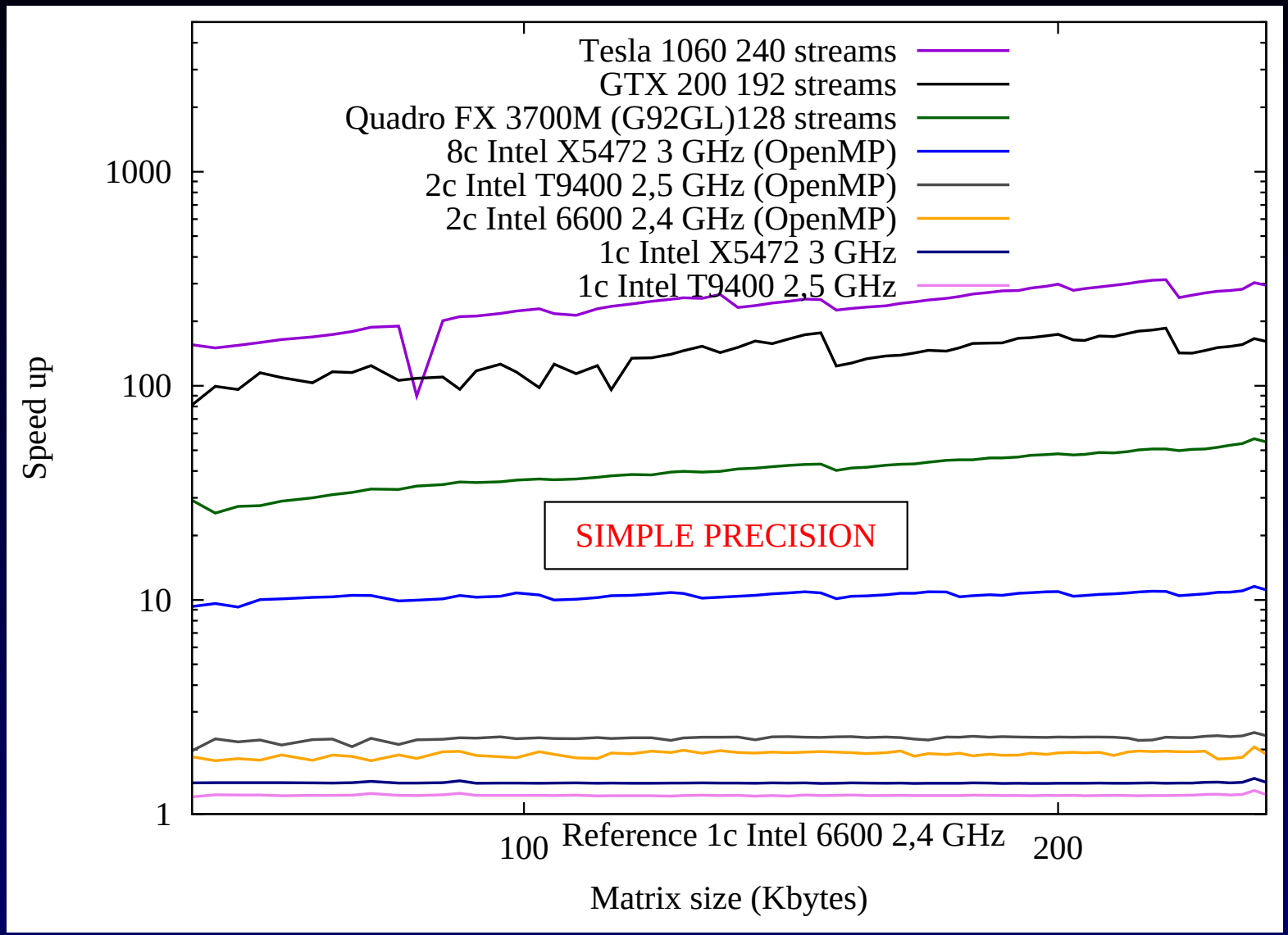
- Holotetrix's primary activities are the design, fabrication and commercialization of prototype diffractive optical elements (DOE) and micro-optics for diverse industrial applications such as LED illumination, laser beam shaping, wavefront analyzers, etc.
- Hologram verification with direct Fresnel simulation
- Program in C
- Parallelized with
  - Par4All CUDA and CUDA 2.3, Linux Ubuntu x86-64
  - Par4All OpenMP, gcc 4.3, Linux Ubuntu x86-64
- Reference: Intel Core2 6600 @ 2.40GHz

`http://www.holotetrix.com`

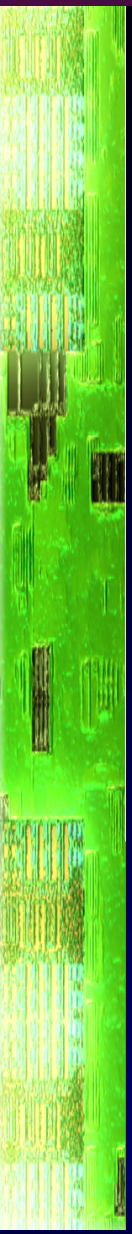
Comparative performance



Keep it stupid simple... precision



Stars-PM



- *Particle-Mesh* N-body cosmological simulation
- C code from Observatoire Astronomique de Strasbourg
- Use FFT 3D
- Example given in `par4all.org` distribution

Stars-PM time step

```
1  void iteration(coord pos[NP][NP][NP],
2                 coord vel[NP][NP][NP],
                  float dens[NP][NP][NP],
4                 int data[NP][NP][NP],
                  int histo[NP][NP][NP]) {
6    /* Split space into regular 3D grid: */
   discretisation(pos, data);
8    /* Compute density on the grid: */
   histogram(data, histo);
10   /* Compute attraction potential
        in Fourier's space: */
12 potential(histo, dens);
     /* Compute in each dimension the resulting forces and
14      integrate the acceleration to update the speeds: */
   forcex(dens, force);
16 updatevel(vel, force, data, 0, dt);
   forcey(dens, force);
18 updatevel(vel, force, data, 1, dt);
   forcez(dens, force);
20 updatevel(vel, force, data, 2, dt);
     /* Move the particles: */
22 updatepos(pos, vel);
}
```

Stars-PM & Jacobi results with p4a 1.1 (I)

- 2 Xeon Nehalem X5670 (12 cores @ 2,93 GHz)
- 1 GPU nVidia Tesla C2050
- Automatic call to CuFFT instead of FFTW
- 150 iterations of Stars-PM

Execution time	p4a	Simulation Cosmo. 32^3	Simulation Cosmo. 64^3	Simulation Cosmo. 128^3	Jacobi
Sequential	(gcc -O3)	0,68	6,30	98,4	24,5
OpenMP 6 *threads*	--openmp	0,16	1,28	16,6	13,8
CUDA base	--cuda	0,88	5,21	31,4	67,7
Optim. comm.	--cuda --com-opt.	0,20	1,17	8,9	6,5
Manual optim.	(gcc -O3)	0,05	0,26	1,7	

Current limitation for Stars-PM with p4a: histogram is not parallelized... PIPS detects the reductions but we do not generate CuDPP calls yet

Outline

1 Par4All global infrastructure
2 OpenMP code generation
3 GPU code generation
4 Code generation for SCMP
5 Scilab compilation
6 Results
7 Conclusion

Coding rules

- Automatic parallelization is not magic
- Use abstract interpretation to « understand » programs
- undecidable in the generic case ($\approx$ halting problem)
- Quite easier for well written programs
- Develop a coding rule manual to help parallelization and... sequential quality!
  - Avoid useless pointers
  - Take advantage of C99 (arrays of non static size...)
  - Use higher-level C, do not linearize arrays...
  - ...
- Prototype of coding rules report on-line on `par4all.org`

Future challenges (I)

- Make a compiler with features that compose: able to generate heterogeneous code for heterogeneous machine with all together:
  - MPI code generation between nodes
  - Generate OpenMP parallel code for SMP Processors inside node
  - Multi-GPU with each SMP thread controlling a GPU
  - Work distribution (*à la *PU*?) between GPU and OpenMP
  - Generate CUDA/OpenCL GPU or other accelerator code
  - Generate SIMD vector code in OpenMP
  - Generate SIMD vector code in GPU code
- These concepts arrive in PyPS through multiple inheritance, mix-ins (use Python dynamic structure a lot!)
- Parallel evolution of Par4All & PyPS $\rightsquigarrow$ refactoring of Par4All back to PyPS future features
- Rely a lot on Par4All Accel run-time
  - Define good minimal abstractions

Future challenges (II)

- Simplify compiler infrastructure
- Improve target portability
- Finding a good ratio between specific architecture features and global efficiency
- Future is to static compilation + run-time optimizations...

Conclusion (I)

- Manycores & GPU: impressive peak performances and memory bandwidth, power efficient
- Domain is maturing: any languages, libraries, applications, tools... Just choose the good one ☺
- Open standards to avoid sticking to some architectures
- Automatic tools can be used for quick start
- Need software tools and environments that will last through business plans or companies
- Open implementations are a warranty for long time support for a technology (cf. current tendency in military and national security projects)
- Par4All motto: keep things simple
- Open Source for community network effect
- Easy way to begin with parallel programming

Conclusion (II)

- Source-to-source
  - Give some programming examples
  - Good start that can be reworked upon
  - Avoid sticking too much on specific target details
- Relying on compilation framework speeds up developments a lot
-  Real codes are often not well written to be parallelized... even by human being ☹
- At least writing clean C99/Fortran/Scilab... code should be a prerequisite
- Take a positive attitude... Parallelization is a good opportunity for deep cleaning (refactoring, modernization...) ⇝ improve also the original code
-  Entry cost
-  Exit cost! ☹
  - Do not loose control on *your* code and *your* data !

Par4All is currently supported by...

- HPC Project
- Institut TÉLÉCOM/TÉLÉCOM Bretagne
- MINES ParisTech
- European ARTEMIS SCALOPES project
- European ARTEMIS SMECY project
- French NSF (ANR) FREIA project
- French NSF (ANR) MediaGPU project
- French System@TIC research cluster OpenGPU project
- French System@TIC research cluster SIMILAN project
- French Sea research cluster MODENA project
- French Images and Networks research cluster TransMedi@ project (finished)

Present motivations 2
HPC Project hardware: WildNode from Wild Systems 3
HPC Project software and services 4
The “Software Crisis” 5
Use the Source, Luke... 6
We need software tools 7
Not reinventing the wheel... No NIH syndrome please! 8
PIPS 9
Current PIPS usage 11
Par4All usage 12

1 Par4All global infrastructure
Outline 13
Par4All ≡ PyPS scripting in the backstage 14

2 OpenMP code generation
Outline 17
Parallelization to OpenMP 18
OpenMP output sample 21

3 GPU code generation
Outline 22
Basic GPU execution model 23
Challenges in automatic GPU code generation 24
Automatic parallelization 25
Outlining 27
From array regions to GPU memory allocation 30
Communication generation 32
Loop normalization 35
From preconditions to iteration clamping 38
Complexity analysis 40
Optimized reduction generation 41
Communication optimization 42
Fortran to C-based GPU languages 43
Par4All Accel runtime 44
Working around CUDA limitations 46

4 Code generation for SCMP
Outline 47
SCMP computer 48
SCMP tasks 49
SCMP task code (before/after) 50
Performance of GSM sensing on SCMP 51

5 Scilab compilation
Outline 52
Scilab language 53
Scilab & Matlab 54

6 Results
Outline 55
Hyantes 56
Results on a customer application 62
Comparative performance 63
Keep it stupid simple... precision 64
Stars-PM 65
Stars-PM time step 66
Stars-PM & Jacobi results with `p4a 1.1` 67

7 Conclusion
Outline 68
Coding rules 69
Future challenges 70
Conclusion 72
Par4All is currently supported by... 74
You are here! 75

VI. Conclusion

VI. Conclusion

VI. Conclusion

VI.0.2

- **Many analyses and transformations**
- **Ready to be combined for new projects**
- **Interprocedural source-to-source tool**
- **Automatic consistency and persistence management**
- **Easy to extend: a matter of hours, not days!**
- **PIPS is used, developped and supported by different institutions:**
  - MINES ParisTech, TELECOM Bretagne, TELECOM SudParis, HPC Project, ...
- **Used in several on-going projects:**
  - FREIA, OpenGPU, SCALOPES, Par4All...
- **May seem difficult to dominate**
  - A little bit of effort at the beginning saves a lot of time

PIPS Future Work

VI.0.3

- **Full support of C:**
  - Semantics analyses extended to structures and pointers
  - Points-to analysis
  - Convex array regions extended to struct and pointers
- **Support of Fortran 95 (using gfortran parser)**
- **Code generators for specific hardware:**
  - CUDA
  - OpenCL
  - SSE
  - Support for FPGA-based hardware accelerator
  - Backend for a SIMD parallel processor
- **Optimization of the OpenMP to MPI translation**

PIPS Online Resources

- **Website:** http://pips4u.org
  - Documentation:
    - **Getting Started** (examples from the Tutorial)
    - **Guides and Manuals** (PDF, HTML):
      - Developers Guide
      - Tpips User Manual
      - Internal Representation for Fortran and C
      - PIPS High-Level Software Interface • Pipsmake Configuration
- **SVN repository:** http://svn.pips4u.org/svn
- **Debian packages**: http://ridee.enstb.org/debian/
- **Trac site:** http://svn.pips4u.org/trac
- **IRC:** irc://irc.freenode.net/pips
- **Mailing lists:** pipsdev at cri.mines-paristech.fr (developer discussions)
  pips-support at cri.mines-paristech.fr (user support)

Credits

- **Laurent Daverio**
  - Coordination and integration
  - Python scripts for OpenOffice slide generation
- **Corinne Ancourt**
- **Fabien Coelho**
- **Stéphanie Even**
- **Serge Guelton**
- **François Irigoin**
- **Pierre Jouvelot**
- **Ronan Keryell**
- **Frédérique Silber-Chaussumier**
- **And all the PIPS contributors...**

Python scripts for Impress (Open Office)

VI.0.6

- **Include files**
- **Colorize files**
- **Compute document outline**
- **Visualize the document structure**