Automatic Code Generation of Distributed Parallel Tasks

Onzième rencontre de la communauté française de compilation

Nelson Lossing Corinne Ancourt François Irigoin

firstname.lastname@mines-paristech.fr

MINES ParisTech,
PSL Research University

Aussois, France, 9 Septembre 2016

Motivation

Scientific Program

Signal Processing

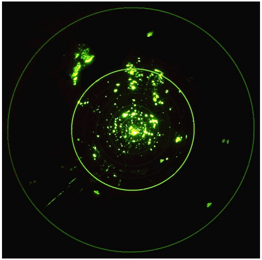
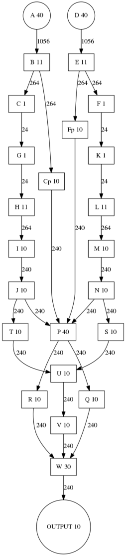


Image Processing



Context

Tools

- Automatic task parallelization (OpenModelica[1])
- Automatic distributed parallelization (Pluto+[2])
- *Black box*

But no automatic distributed parallelization task tool

[1] Mahder Gebremedhin and Peter Fritzson. "Automatic Task Based Analysis and Parallelization in the Context of Equation Based Languages". In: *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT '14. Berlin, Germany: ACM, 2014, pp. 49–52.

[2] Uday Bondhugula. "Compiling Affine Loop Nests for Distributed-memory Parallel Architectures". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013.

Source-to-Source Transformations

- Fortran code
- C code

- *Scientific Program*
- *Image Processing*

- Static analyses
- Instrumentation/ Dynamic analyses
- Transformations
- Source code generation
- Code modelling
- Prettyprint

- Fortran code
- C code

- *Distributed Parallel Code*

Effects and Array Regions

Effects

- Read Effect:
  Set of variables that are read/used by a Statement
- Write Effect:
  Set of variables that are written/defined by a Statement

Convex Array Regions

- Convex Array Region:
  Convex set of elements of an array
- Read Array Region (Read Region):
  Set of convex array regions that are read/used by a Statement
- Write Array Region (Write Region):
  Set of convex array regions that are written/defined by a Statement

Formalization Effects and Array Regions

$$\text{Identifier} \xrightarrow{\rho} \text{Location} \xrightarrow{\sigma} \text{Value}$$

- *Identifier*, *Location*, *Value* `int x = 0;`
- **Environment**, *Env* ρ: *Identifier* $\rightarrow$ *Location*
- **Memory State**, *MemState* σ: *Location* $\rightarrow$ *Value*
- **Statement** S: *Env* $\times$ *MemState* $\rightarrow$ *Env* $\times$ *MemState*
- **Memory Effect** E:
  Statement $\rightarrow$ *Env* $\times$ *MemState* $\rightarrow \mathcal{P}(\textit{Location})$
  - Read Effect E_R
  - Write Effect E_W
- **Array Region** R:
  $\textit{Statement} \rightarrow \textit{Env} \times \textit{MemState} \rightarrow \mathcal{P}((\textit{Location}, \prod_{di \in 0,n} \mathcal{P}(\mathbb{Z}^{di})))$
  - Read Region R_R
  - Write Region R_W

Example Proper Effects and Proper Array Regions

Proper Effects

```
//   Ew: x
x = 0;


//   Ew: i
for(i = 0; i <= 9; i += 1) {
//   Ew: a[i]
//   Er: i
  a[i] = i;
//   Ew: x
//   Er: i x
  x += i;
}


//   Ew: i
for(i = 0; i <= 4; i += 1)
//   Ew: a[i]
//   Er: a[i] i x
  a[i] += x;


//   Ew: i
for(i = 5; i <= 9; i += 1)
//   Ew: a[i]
//   Er: a[i] i x
  a[i] -= x;
```

Proper Array Regions

```
x = 0;


for(i = 0; i <= 9; i += 1) {

//   Rw: (a[PHI1], {PHI1==i, 0<=i, i<=9})
  a[i] = i;


  x += i;
}


for(i = 0; i <= 4; i += 1)
//   Rw: (a[PHI1], {PHI1==i, 0<=i, i<=4})
//   Rr: (a[PHI1], {PHI1==i, 0<=i, i<=4})
  a[i] += x;


for(i = 5; i <= 9; i += 1)
//   Rw: (a[PHI1], {PHI1==i, 5<=i, i<=9})
//   Rr: (a[PHI1], {PHI1==i, 5<=i, i<=9})
  a[i] -= x;
```

Example Cumulated Effects and Array Regions

Cumulated Effects

```
//   Ew: x
x = 0;

//   Ew: a[*] i x
//   Er: i x
for(i = 0; i <= 9; i += 1) {
//   Ew: a[*]
//   Er: i
  a[i] = i;
//   Ew: x
//   Er: i x
  x += i;
}

//   Ew: a[*] i
//   Er: a[*] i x
for(i = 0; i <= 4; i += 1)
//   Ew: a[*]
//   Er: a[*] i x
  a[i] += x;

//   Ew: a[*] i
//   Er: a[*] i x
for(i = 5; i <= 9; i += 1)
//   Ew: a[*]
//   Er: a[*] i x
  a[i] -= x;
```

Array Regions

```
x = 0;


//   Rw: (a[PHI1], {0<=PHI1, PHI1<=9})
for(i = 0; i <= 9; i += 1) {

//   Rw: (a[PHI1], {PHI1==i, 0<=i, i<=9})
  a[i] = i;


  x += i;
}

//   Rw: (a[PHI1], {0<=PHI1, PHI1<=4})
//   Rr: (a[PHI1], {0<=PHI1, PHI1<=4})
for(i = 0; i <= 4; i += 1)
//   Rw: (a[PHI1], {PHI1==i, 0<=i, i<=4})
//   Rr: (a[PHI1], {PHI1==i, 0<=i, i<=4})
  a[i] += x;

//   Rw: (a[PHI1], {5<=PHI1, PHI1<=9})
//   Rr: (a[PHI1], {5<=PHI1, PHI1<=9})
for(i = 5; i <= 9; i += 1)
//   Rw: (a[PHI1], {PHI1==i, 5<=i, i<=9})
//   Rr: (a[PHI1], {PHI1==i, 5<=i, i<=9})
  a[i] -= x;
```

In/Out Effects and Array Regions

Effects

- In Effect:
  Set of variables that are used by a Statement and had to be previously defined
- Out Effect:
  Set of variables that are defined by a Statement and that will be used in its continuation

Array Regions

- In Array Region (In Region):
  Set of convex array regions that are used by a Statement and were previously defined
- Out Array Region (Out Region):
  Set of convex array regions that are defined by a Statement and that will be used in the continuation

Example In/Out Effects and In/Out Array Regions

In/Out Effects

```
//  Eout: x
x = 0;

//  Eout: a[*] x
//  Ein : x
for(i = 0; i <= 9; i += 1) {
//  Eout: a[*]
//  Ein : i
  a[i] = i;
//  Eout: x
//  Ein : i x
  x += i;
}

//  Eout: a[*]
//  Ein : a[*] x
for(i = 0; i <= 4; i += 1)
//  Eout: a[*]
//  Ein : a[*] i x
  a[i] += x;


//  Ein : a[*] x
for(i = 5; i <= 9; i += 1)

//  Ein : a[*] i x
  a[i] -= x;
```

In/Out Array Regions

```

x = 0;


//  Rout: (a[PHI1], {0<=PHI1, PHI1<=9})
for(i = 0; i <= 9; i += 1) {

//  Rout: (a[PHI1], {PHI1==i, 0<=i, i<=9})
  a[i] = i;


  x += i;
}


//  Rin : (a[PHI1], {0<=PHI1, PHI1<=4})
for(i = 0; i <= 4; i += 1)

//  Rin : (a[PHI1], {PHI1==i, 0<=i, i<=4})
  a[i] += x;


//  Rin : (a[PHI1], {5<=PHI1, PHI1<=9})
for(i = 5; i <= 9; i += 1)

//  Rin : (a[PHI1], {PHI1==i, 5<=i, i<=9})
  a[i] -= x;
```

Relation Between Live Variables and In/Out Effects

Live Variables

- Live Variable:
  Set of variables that may be potentially read before their next write
- Live-In: Live Variable immediately before a Statement
- Live-Out: Live Variable immediately after a Statement

Relation Between Live Variables and In/Out Effects

Live Variables

- Live Variable:
  Set of variables that may be potentially read before their next write
- Live-In: Live Variable immediately before a Statement
- Live-Out: Live Variable immediately after a Statement

$$LIVE_{in}(s) = (LIVE_{out}(s) \setminus DEF(s)) \cup USED(s)$$

$$LIVE_{out}(s) = \bigcup_{p \in succ(s)} LIVE_{in}(p)$$

$$LIVE_{out}(f) = \varnothing$$

Relation Between Live Variables and In/Out Effects

Live Variables

- Live Variable:
  Set of variables that may be potentially read before their next write
- Live-In: Live Variable immediately before a Statement
- Live-Out: Live Variable immediately after a Statement

$$LIVE_{in}(s) = (LIVE_{out}(s) \setminus DEF(s)) \cup USED(s)$$

$$LIVE_{out}(s) = \bigcup_{p \in succ(s)} LIVE_{in}(p)$$

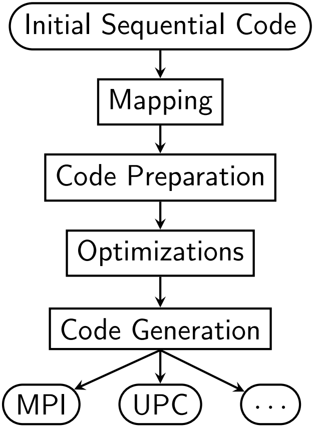
$$LIVE_{out}(f) = \varnothing$$

$$E_{in}(s) = USED(s) \cap LIVE_{in}(s)$$

$$E_{out}(s) = DEF(s) \cap LIVE_{out}(s)$$

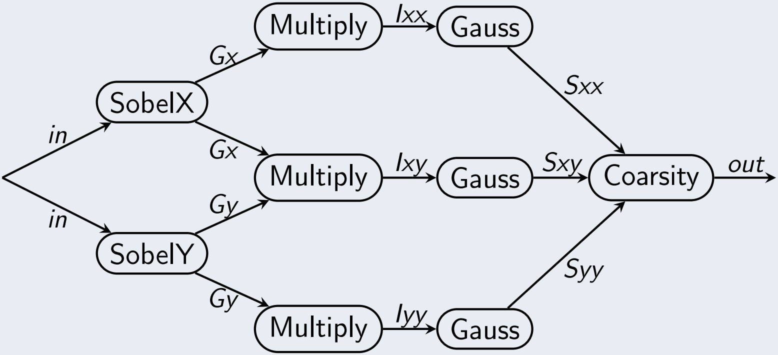
$$LIVE_{out}(s) = [LIVE_{in}(s) \setminus (E_{in}(s) \setminus \ldots)] \cup E_{out}(s)$$

Compilation Process



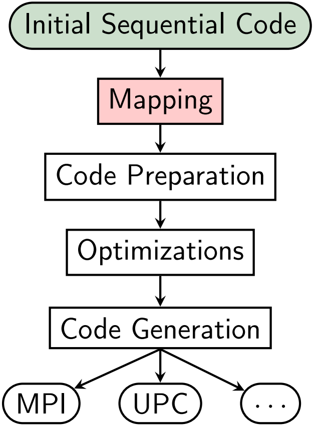
Case Example

Harris&Stephens algorithm[3]



[3] Chris Harris and Mike Stephens. "A combined corner and edge detector". In: *Proc. of Fourth Alvey Vision Conference*. 1988, pp. 147–151

Compilation Process



Mapping

Can be done

- Automatically with a task scheduler[4]
- Manually

Pragma directive

- New pragma `distributed`
- On sequence of instructions, loop, test, etc.
- Not inside loop or condition
- `on_cluster` to define the process to use
- No data dependence information needed [5]

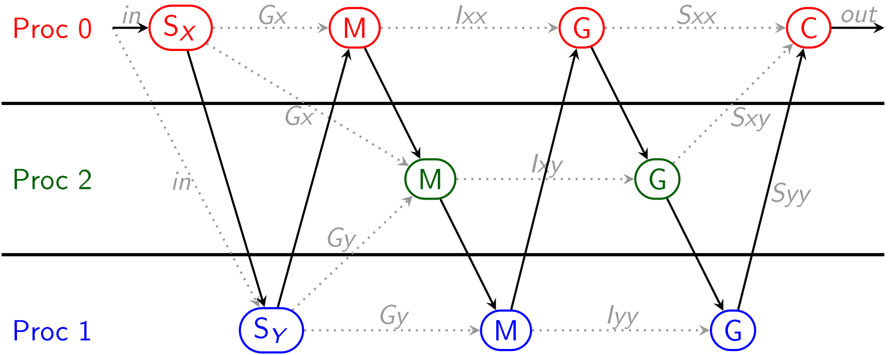
[4] Dounia Khaldi, Pierre Jouvelot, and Corinne Ancourt. "Parallelizing with BDSC, a Resource-constrained Scheduling Algorithm for Shared and Distributed Memory Systems". In: *Parallel Comput.* 41.C (Jan. 2015), pp. 66–89

[5] Martin Tillenius et al. "Resource-Aware Task Scheduling". In: *ACM Trans. Embed. Comput. Syst.* 14.1 (Jan. 2015), 5:1–5:25

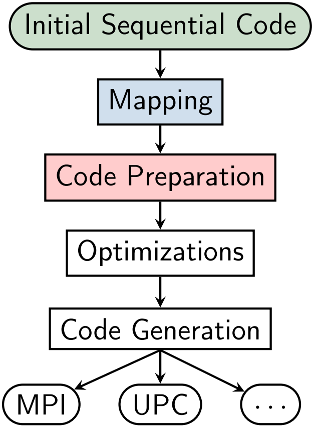
Mapping Example

```
#pragma distributed on_cluster = 0
  init_array(6000, 5900, in);

#pragma distributed on_cluster = 0
  SobelX(6000, 5900, Gx, in);
#pragma distributed on_cluster = 1
  SobelY(6000, 5900, Gy, in);

#pragma distributed on_cluster = 0
  Multiply(6000, 5900, Ixx, Gx, Gx);
#pragma distributed on_cluster = 2
  Multiply(6000, 5900, Ixy, Gx, Gy);
#pragma distributed on_cluster = 1
  Multiply(6000, 5900, Iyy, Gy, Gy);

#pragma distributed on_cluster = 0
  Gauss(6000, 5900, Sxx, Ixx);
#pragma distributed on_cluster = 2
  Gauss(6000, 5900, Sxy, Ixy);
#pragma distributed on_cluster = 1
  Gauss(6000, 5900, Syy, Iyy);

#pragma distributed on_cluster = 0
  CoarsitY(6000, 5900, out, Sxx, Syy, Sxy);

#pragma distributed on_cluster = 0
  print_array(6000, 5900, out);
```

After Mapping



Compilation Process



Code Preparation

Storage Replication

1. **Environment Replication**
   Add declaration for each variable on each process
2. **Store Consistency**
   Add copy/communication for written variables
3. **Localization**
   Substitute "original variables" by "local variables"
4. **Complete Localization**
   Remove "original variables" declarations

Code Preparation – Environment Replication

1. Add declaration for each variable on each process

```
int x;
```

⇒

```
int x;
int x_0;
int x_1;
int x_2;
...
```

Code Preparation – Store Consistency

2. Add copy/communication for written variables

- inside the task
  - More precise
  - Issue for code generation on dynamic cases
  - Redundant copy
- use Write/Out Proper Effects

```
#pragma distributed on_cluster 0
{
  ...
  x=0;
  // copy x on x_0, x_1...
  if (rand()) {
    x=1;
    // copy x on x_0, x_1...
  } else {
    y=1;
    // copy y on y_0, y_1...
  }
  ...
}
```

- between the tasks
  - No dynamic cases
  - No redundant copy
  - Less precise
- use Write/Out Array Regions

```
#pragma distributed on_cluster 0
{
  ...
  x=0;
  if (rand()) {
    x=1;
  } else {
    y=1;
  }
  ...
  // copy x on x_0, x_1...
  // copy y on y_0, y_1...
  ...
}
```

Code Preparation – Localization

3. Substitute “original variables” by “local variables”

```
#pragma distributed on_cluster 0
{
  ...
  x=0;
  ...
  x_0=x;
  x_1=x;
  x_2=x;
  ...
}
```

$\Rightarrow$

```
#pragma distributed on_cluster 0
{
  ...
  x_0=0;
  ...
  x_0=x_0;
  x_1=x_0;
  x_2=x_0;
  ...
}
```

Code Preparation – Localization

3. Substitute "original variables" by "local variables"

```
#pragma distributed on_cluster 0
{
  ...
  x=0;
  ...
  x_0=x;
  x_1=x;
  x_2=x;
  ...
}
```

⇒

```
#pragma distributed on_cluster 0
{
  ...
  x_0=0;
  ...
  x_0=x_0;
  x_1=x_0;
  x_2=x_0;
  ...
}
```

4. Remove "original variables" declarations

```
int x;
int x_0;
int x_1;
int x_2;
...
```

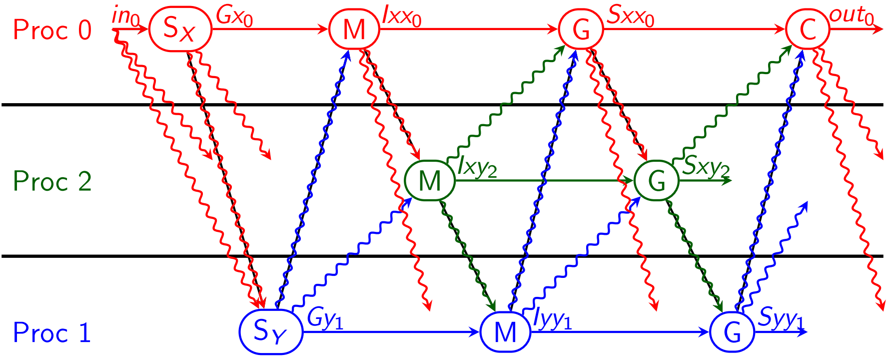
⇒

```
int x_0;
int x_1;
int x_2;
...
```

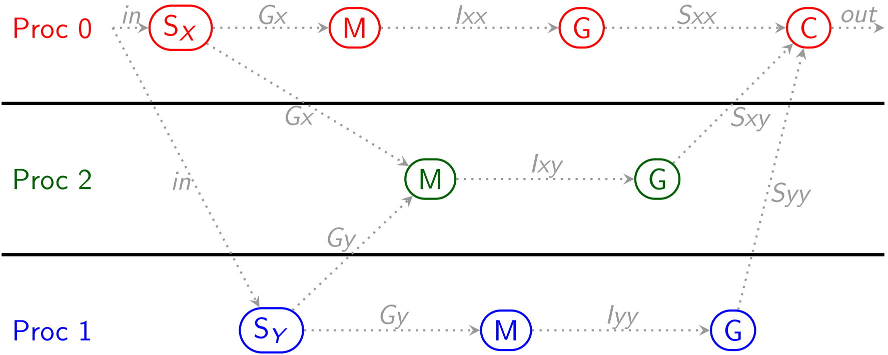
Code Preparation Example

```
/* Variable declaration/allocation. */
double __in_0[6000][5900], __in_1[6000][5900], __in_2[6000][5900];
double __Gx_0[6000][5900], __Gx_1[6000][5900], __Gx_2[6000][5900];
...

#pragma distributed on_cluster = 0
{
  init_array(6000, 5900, __in_0);
  {
    int PHI1, PHI2;
    for (PHI1 = 0; PHI1 <= 5999; PHI1 += 1)
      for (PHI2 = 0; PHI2 <= 5899; PHI2 += 1) {
        __in_1[PHI1][PHI2]=__in_0[PHI1][PHI2];
        __in_2[PHI1][PHI2]=__in_0[PHI1][PHI2];
      }
  }
}

#pragma distributed on_cluster = 0
{
  SobelX(6000, 5900, __Gx_0, __in_0);
  /* Copy __Gx_1=__Gx_0 */
  /* Copy __Gx_2=__Gx_0 */
}
#pragma distributed on_cluster = 1
{
  SobelY(6000, 5900, __Gy_1, __in_1);
  /* Copy __Gy_0=__Gy_1 */
  /* Copy __Gy_2=__Gy_1 */
}

...
```

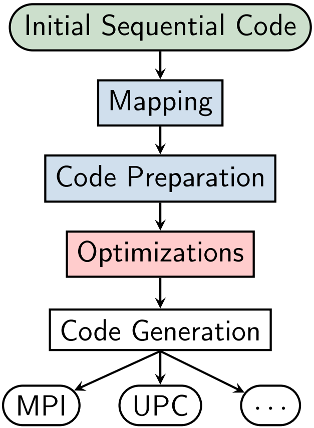
After Code Preparation



Task Graph



Proc 0
in
S_X
Gx
M
Ixx
G
Sxx
C
out
Gx
Sxy
Proc 2
in
M
Ixy
G
Syy
Gy
Proc 1
S_Y
Gy
M
Iyy
G


Initial Sequential Code
Mapping
Code Preparation
Optimizations
Code Generation
MPI
UPC
. . .

Optimizations

Reduce Copy/Communication

- Dead-code Elimination
- Dead-iteration Elimination

Reduce Memory Footprint

- Array Resizing

Reduce Copy/Communication

- Dead-code Elimination
- Dead-iteration Elimination

	case 1	case 2	case 3
task on P	write x_P ~~x_Q = x_P~~ x_R = x_P	write x_P x_Q = x_P ~~x_R = x_P~~	write a_P[0..n] ~~a_Q[0..n]=a_P[0..n]~~ a_Q[0..n/2]=a_P[0..n/2] ~~a_R[0..n]=a_P[0..n]~~ a_R[n/2..n]=a_P[n/2..n]
task on Q Q≠P	...	read x_Q write x_Q x_P = x_Q x_R = x_Q	read a_Q[0..n/2]
task on R R≠{P,Q}	read x_R	read x_R	read a_R[n/2..n]

Reduce Memory Footprint

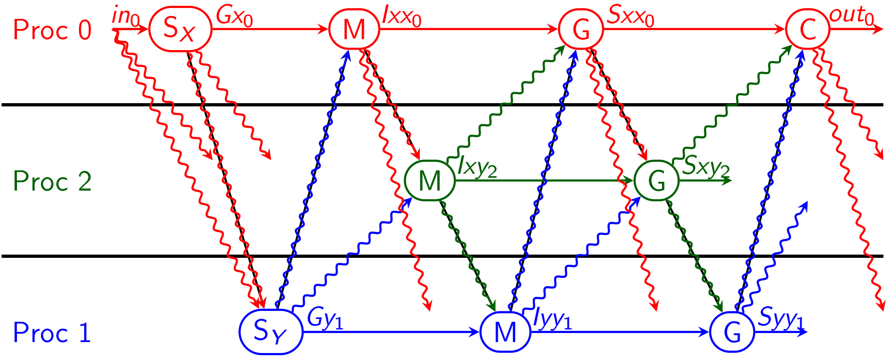
- Array Resizing
  - Use Read/Write Regions
  - Compute new array size
  - Resize array declarations
  - Shift array cell accesses

```c
int a[20];
int i;
for (i=5; i<15; i++)
  a[i] = i*i;
```

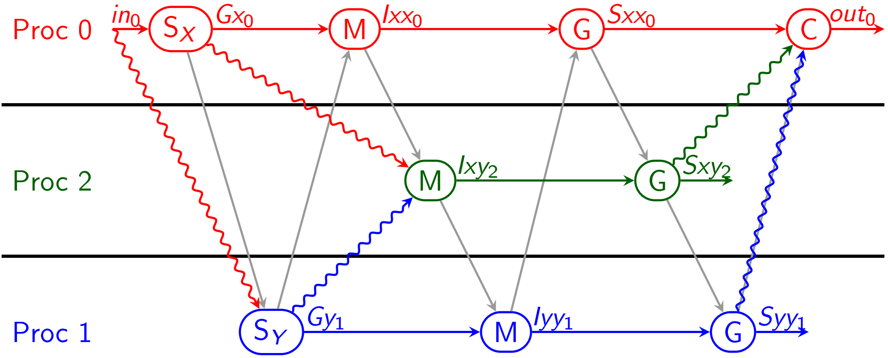
⇒

```c
int a[10];
int i;
for (i=5; i<15; i++)
  a[i-5] = i*i;
```

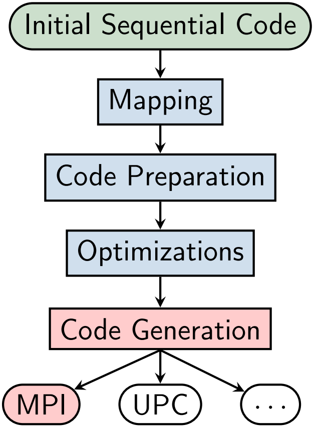
Before Optimizations



After Optimizations



Compilation Process



Parallel Code Generation

MPI Code Generation

- Configure MPI Environment
- Replace pragma block by test on process rank
- Replace copy by
  - Send message for rhs on rhs process to lhs process
  - Receive message for lhs on lhs process from rhs process

MPI Code Generation

- **Configure MPI Environment**
- Replace pragma block by test on process rank
- Replace copy by
  - Send message for rhs on rhs process to lhs process
  - Receive message for lhs on lhs process from rhs process

```
MPI_Status status;
int size, rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (size<REQUIRE_PROC_NUMBER) {
  printf("Not␣enough␣processes␣launched!");
  MPI_Finalize();
  return 0;
}
...
MPI_Finalize();
return 0;
```

Parallel Code Generation

MPI Code Generation

- Configure MPI Environment
- **Replace pragma block by test on process rank**
- Replace copy by
  - Send message for rhs on rhs process to lhs process
  - Receive message for lhs on lhs process from rhs process

```
#pragma distributed on_cluster 0
{
  ...
}
```

$\Rightarrow$

```
if (rank==0)
{
  ...
}
```

MPI Code Generation

- Configure MPI Environment
- Replace pragma block by test on process rank
- **Replace copy by**
  - **Send message for rhs on rhs process to lhs process**
  - **Receive message for lhs on lhs process from rhs process**

```
x_1 = x_0;
```

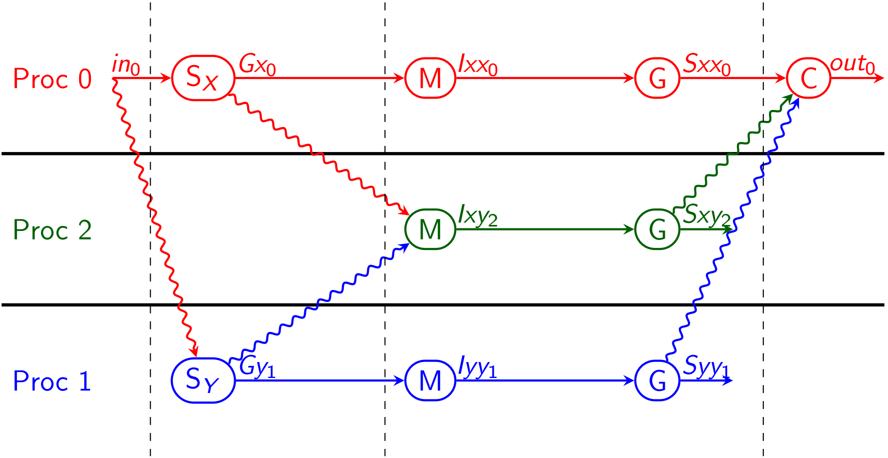
$\Rightarrow$

```
if (rank==0)
  MPI_Send(&x_0, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
if (rank==1)
  MPI_Recv(&x_1, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
```

Parallel Code Generation Example

```
MPI_Status status;
int size, rank;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* ...Check number of process running... */
/* Variable declaration/allocation. */
double __in_0[6000][5900], __in_1[6000][5900];
...

if (rank == 0) {
  init_array(6000, 5900, __in_0);
  MPI_Send(&__in_0[0][0], 6000 * 5900, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
}
if (rank == 1) {
  MPI_Recv(&__in_1[0][0], 6000 * 5900, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
}

if (rank == 0) {
  SobelX(6000, 5900, __Gx_0, __in_0);
  MPI_Send(&__Gx_0[0][0], 6000 * 5900, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);
}
if (rank == 2) {
  MPI_Recv(&__Gx_2[0][0], 6000 * 5900, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
}

...

if (rank == 0)
  CoarsityY(6000, 5900, __out_0, __Sxx_0, __Syy_0, __Sxy_0);
if (rank == 0)
  print_array(6000, 5900, __out_0);
MPI_Finalize();
```

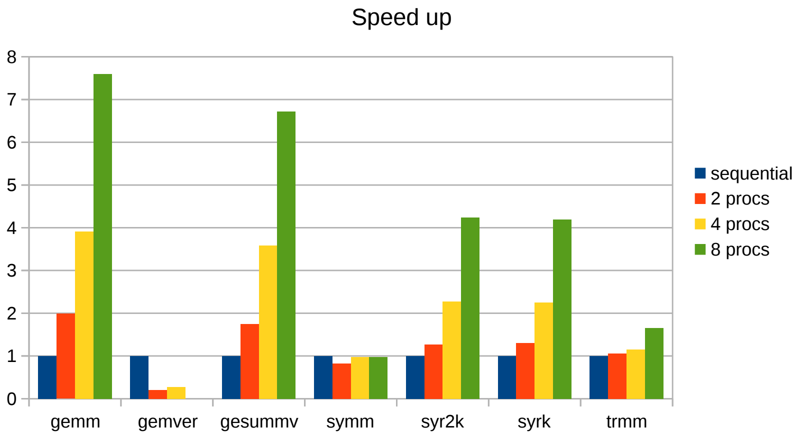
After Parallel Code Generation



Proof Idea

- Initial Sequential Code/Mapping
  Hypothesis: the code is correct
- Code Preparation
  $\forall v, \forall p_1, p_2, \forall T, \forall(\rho', \sigma'), \exists(\rho, \sigma)/(\rho, \sigma) = T(\rho', \sigma'),$
  $\sigma(v_{p_1}) = \sigma(v_{p_2})$
- Optimizations
  $\forall v, \forall p_1, p_2, \forall T_{p_1}, \forall(\rho', \sigma'), \exists(\rho, \sigma)/(\rho, \sigma) = T_{p_1}(\rho', \sigma'),$
  if $\exists T_{p_2} / T_{p_2} \in succ^*(T_{p_1}) \wedge v_{p_2} \in IN(T_{p_2}), \sigma(v_{p_1}) = \sigma(v_{p_2})$
- Parallel Code Generation ($\mathcal{T}$)
  $\forall v, \forall p_1, \sigma_f(v_{p_1}) = \mathcal{T}(\sigma_f)(v_{p_1})$

Experimental Results



Benchmark: BLAS in Polybench
size: ∼3000x4000
type: double

Assumptions & Observations

Assumptions

- Number of processes known at the beginning
- No dynamic parallelism
- Communication overestimation in case of dynamic communications

Observations

- Strongly mapping dependent

Conclusion

Achievement

- Automatic source-to-source transformations
- Succession of simple transformations
- Basic communication functions
- Provable transformations
- Good speed-ups

Future Work

- Improvement of the initial mapping by loop rescheduling
- Comparison with Pluto+
- Asynchronous communications instead of synchronous communications
- Function outlining on each process to improve memory footprint

Automatic Code Generation of Distributed Parallel Tasks

Onzième rencontre de la communauté française de compilation

Nelson Lossing Corinne Ancourt François Irigoin
firstname.lastname@mines-paristech.fr

MINES ParisTech,
PSL Research University

Aussois, France, 9 Septembre 2016