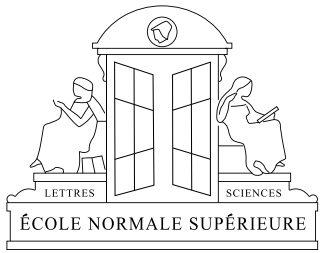
Implicit parallelization of code called from an external and already parallelized environment: from design to implementation.

Pablo Rauzy

pablo.rauzy@ens.fr



December 28, 2011

M1 internship defense

Internship supervised by Clemens Grelck (c.grelck@uva.nl)

The SAC programming language

functional array-based high-performance computing implicit parallelization

Arrays

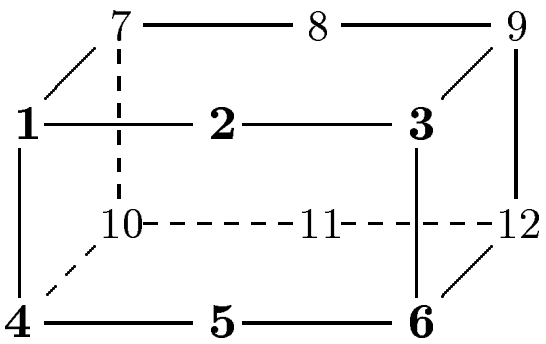
$$A_1 = \langle\ 1 \quad 2 \quad 3\ \rangle$$

shape vector: $[3]$
data vector: $[1, 2, 3]$

$$A_2 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

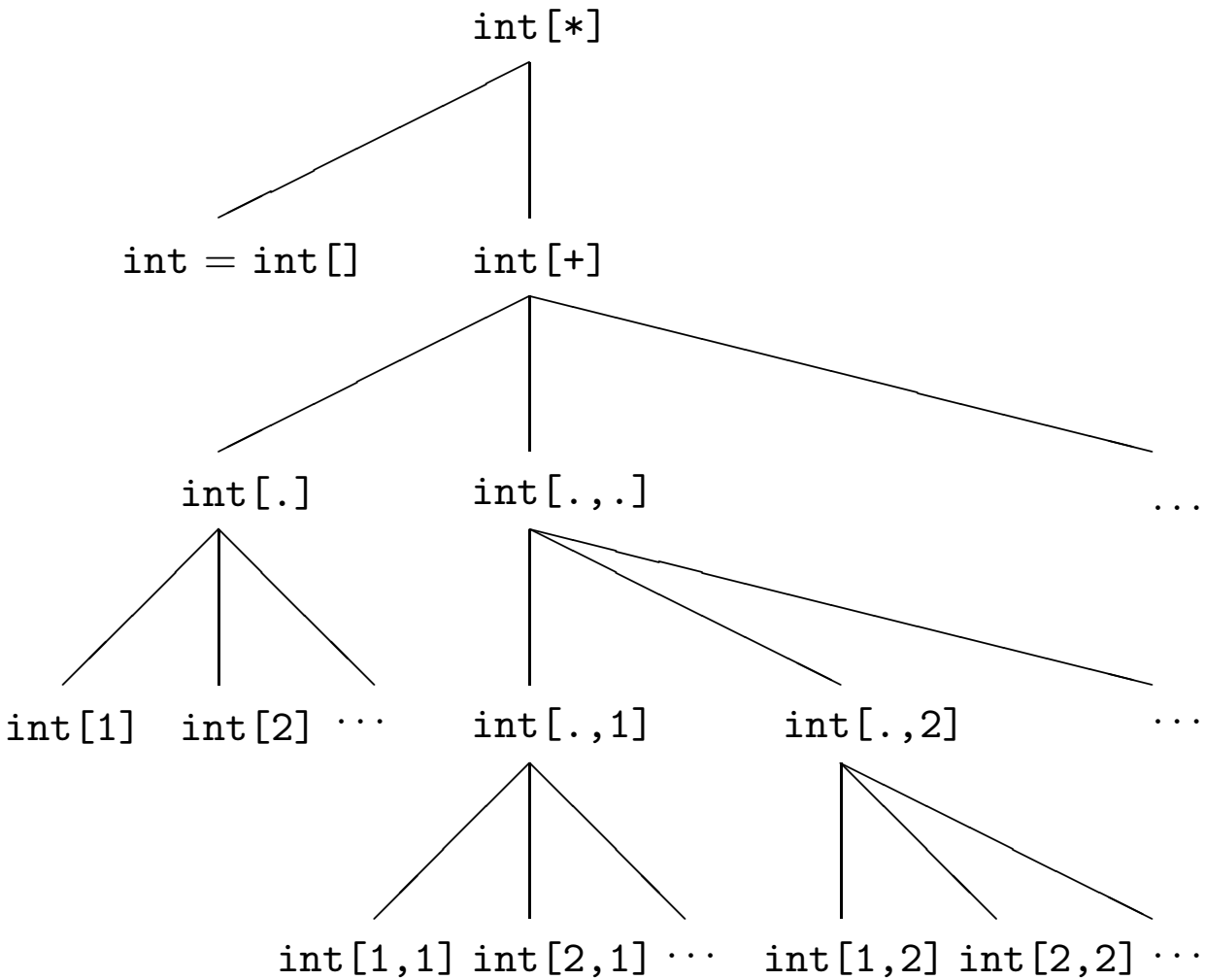
shape vector: $[2, 3]$
data vector: $[1, 2, 3, 4, 5, 6]$

$A_3 =$



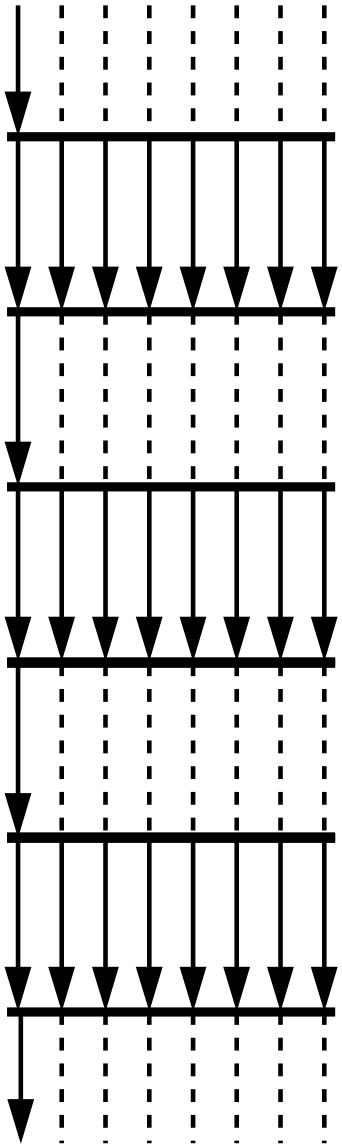
shape vector: $[2, 2, 3]$
data vector: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$

Arrays


int[*]
int = int[]
int[+]
int[.]
int[.,.]
...
int[1]
int[2]
···
int[.,1]
int[.,2]
···
int[1,1]
int[2,1]
···
int[1,2]
int[2,2]
···

Implicit parallelization

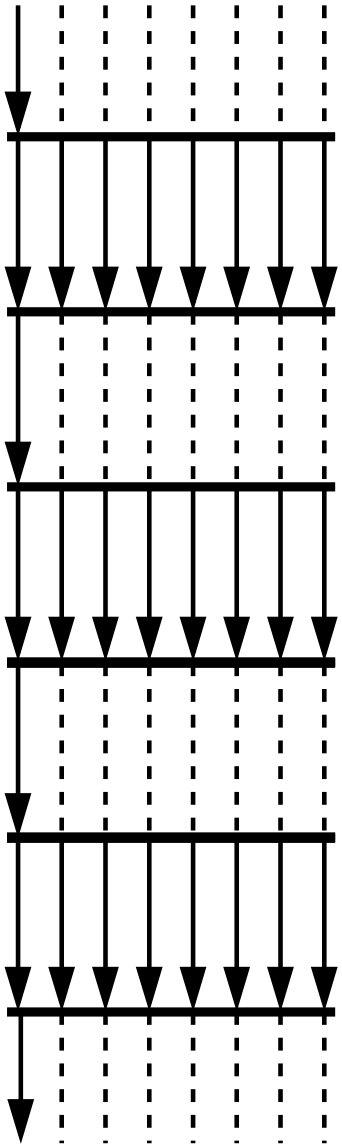
Execution model



- Worker threads are created at the very beginning of the program execution.
- All of them immediately hit a start barrier.
- When the master threads encounter a `with` loop, we lift the start barrier and workers start computing.
- When a worker finishes its work it hits a stop barrier.
- If needed the stop barrier collects the results.
- The stop barrier is then lifted and the thread hits the next start barrier.
- When the master thread finishes its work it waits at the stop barrier for the others to finish before proceeding to the following sequential code.

Implicit parallelization

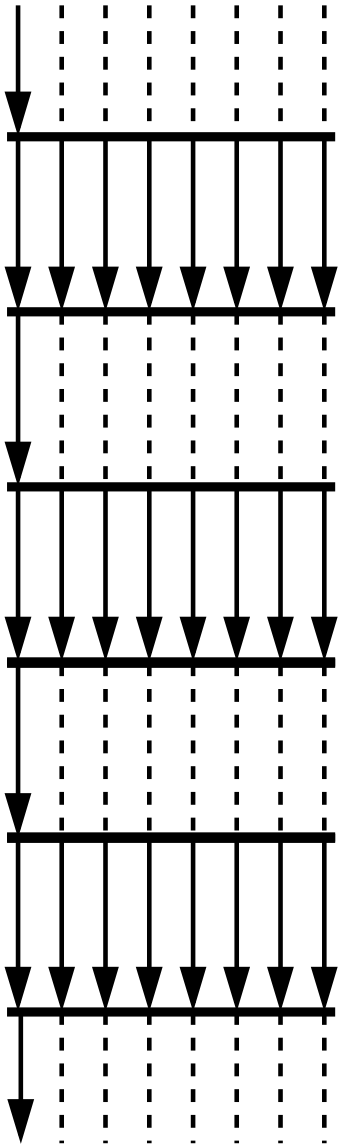
Execution model



- Worker threads are created at the very beginning of the program execution.
- All of them immediately hit a start barrier.
- When the master threads encounter a `with` loop, we lift the start barrier and workers start computing.
- When a worker finishes its work it hits a stop barrier.
- If needed the stop barrier collects the results.
- The stop barrier is then lifted and the thread hits the next start barrier.
- When the master thread finishes its work it waits at the stop barrier for the others to finish before proceeding to the following sequential code.

Implicit parallelization

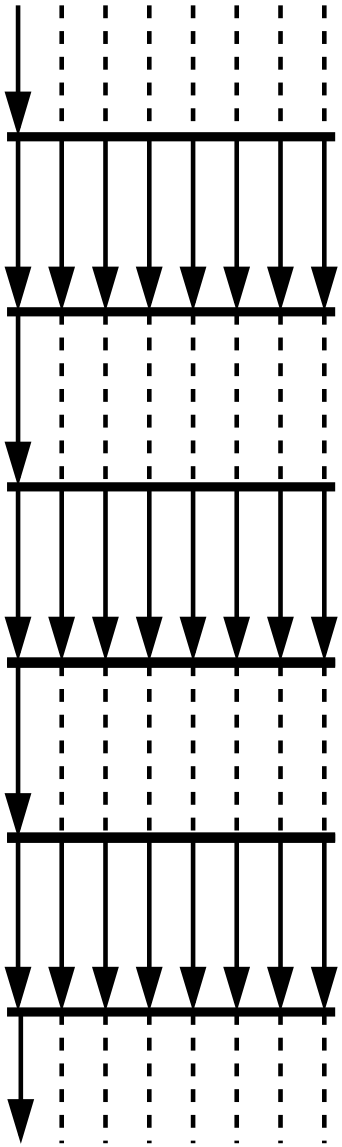
Execution model



- Worker threads are created at the very beginning of the program execution.
- All of them immediately hit a start barrier.
- When the master threads encounter a `with` loop, we lift the start barrier and workers start computing.
- When a worker finishes its work it hits a stop barrier.
- If needed the stop barrier collects the results.
- The stop barrier is then lifted and the thread hits the next start barrier.
- When the master thread finishes its work it waits at the stop barrier for the others to finish before proceeding to the following sequential code.

Implicit parallelization

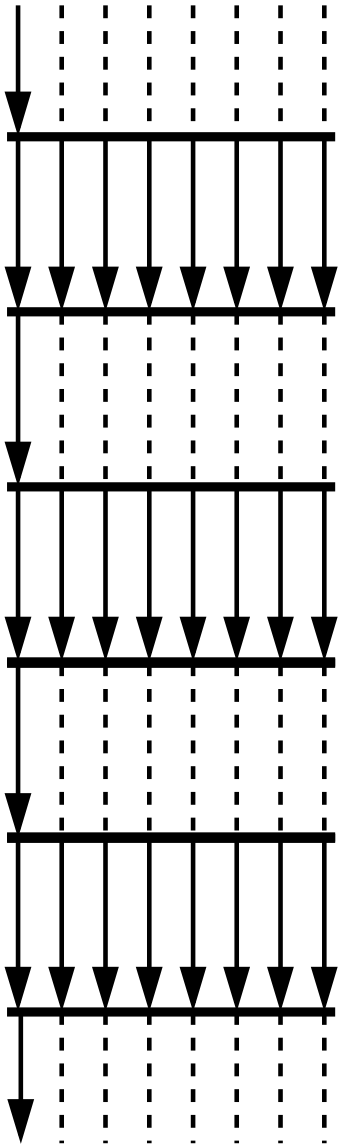
Execution model



- Worker threads are created at the very beginning of the program execution.
- All of them immediately hit a start barrier.
- When the master threads encounter a `with` loop, we lift the start barrier and workers start computing.
- When a worker finishes its work it hits a stop barrier.
- If needed the stop barrier collects the results.
- The stop barrier is then lifted and the thread hits the next start barrier.
- When the master thread finishes its work it waits at the stop barrier for the others to finish before proceeding to the following sequential code.

Implicit parallelization

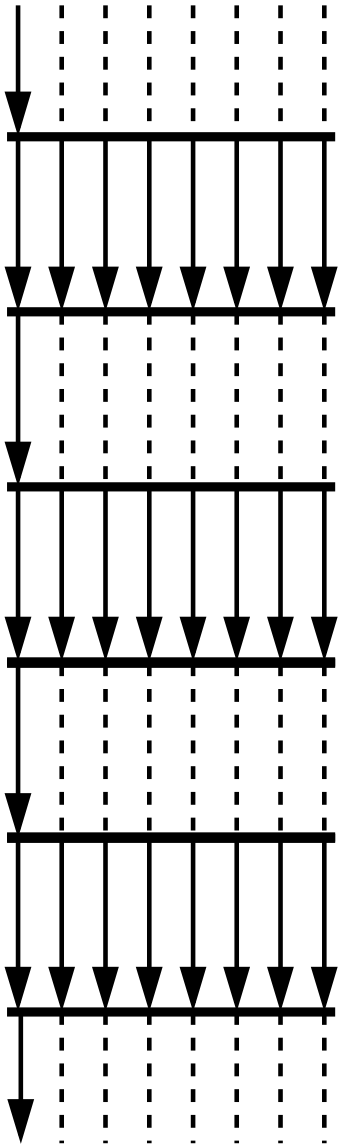
Execution model



- Worker threads are created at the very beginning of the program execution.
- All of them immediately hit a start barrier.
- When the master threads encounter a `with` loop, we lift the start barrier and workers start computing.
- When a worker finishes its work it hits a stop barrier.
- If needed the stop barrier collects the results.
- The stop barrier is then lifted and the thread hits the next start barrier.
- When the master thread finishes its work it waits at the stop barrier for the others to finish before proceeding to the following sequential code.

Implicit parallelization

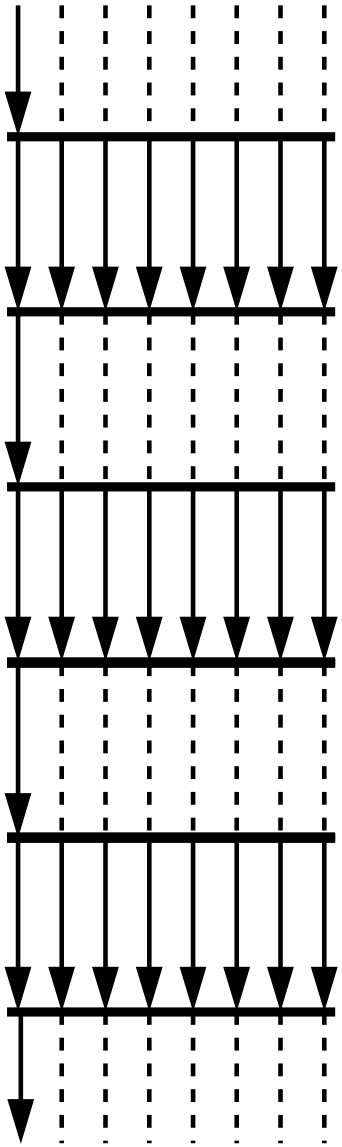
Execution model



- Worker threads are created at the very beginning of the program execution.
- All of them immediately hit a start barrier.
- When the master threads encounter a `with` loop, we lift the start barrier and workers start computing.
- When a worker finishes its work it hits a stop barrier.
- If needed the stop barrier collects the results.
- The stop barrier is then lifted and the thread hits the next start barrier.
- When the master thread finishes its work it waits at the stop barrier for the others to finish before proceeding to the following sequential code.

Implicit parallelization

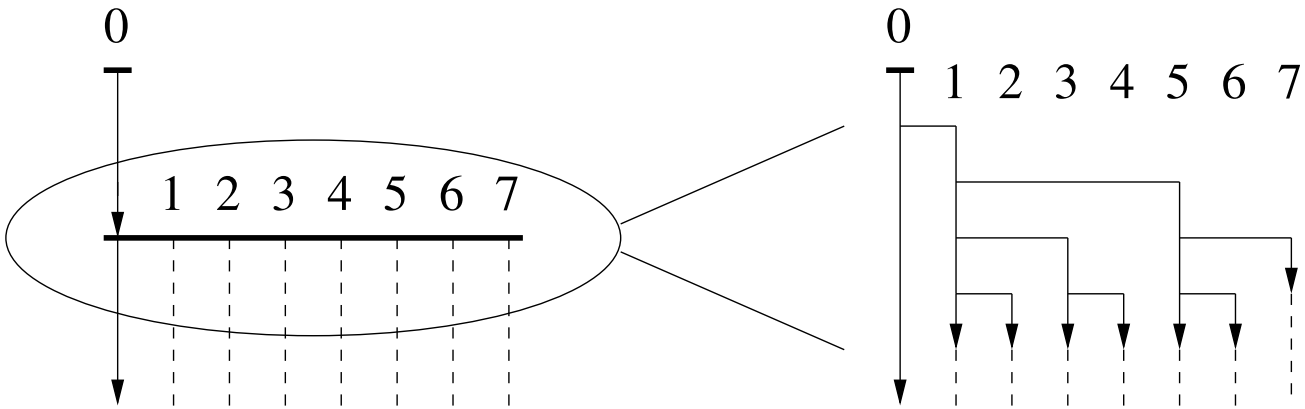
Execution model



- Worker threads are created at the very beginning of the program execution.
- All of them immediately hit a start barrier.
- When the master threads encounter a `with` loop, we lift the start barrier and workers start computing.
- When a worker finishes its work it hits a stop barrier.
- If needed the stop barrier collects the results.
- The stop barrier is then lifted and the thread hits the next start barrier.
- When the master thread finishes its work it waits at the stop barrier for the others to finish before proceeding to the following sequential code.

Implicit parallelization

Worker threads creation



Implicit parallelization

with-loop

```
a = with {
  ( [1, 1] <= iv <= . ) : 42;
} : genarray([13, 37], 0);
```

- Versatile SAC-specific language construct for the definition of aggregate array operations.
- Either define the shape of an array to be created ("genarray" and "modarray"), or a fold operation ("fold" with-loop).
- Allow for the specification of dimension-invariant array operations.

The with-loop is the core operation in the SAC programming language: it is also where the implicit parallelization might happen.

Implicit parallelization

Different versions of functions

SEQ	Sequential functions.
ST	Single threaded context, might go parallel.
MT	Already multi threaded context.

Implicit parallelization

Single Program, Multiple Data

- SPMD functions.
- SPMD frames.
- SPMD barriers.

Implicit parallelization

Single Program, Multiple Data

- SPMD functions.
- SPMD frames.
- SPMD barriers.

Implicit parallelization

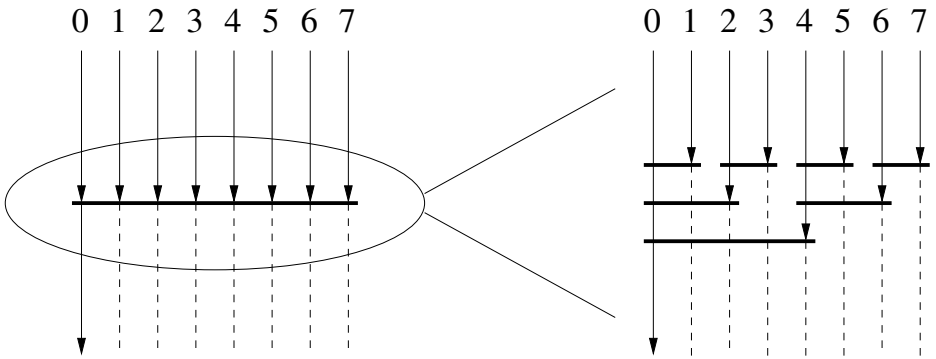
Single Program, Multiple Data

- SPMD functions.
- SPMD frames.
- SPMD barriers.

Implicit parallelization

Single Program, Multiple Data

- SPMD functions.
- SPMD frames.
- SPMD barriers.



Implicit parallelization

Scheduling

- Static or dynamic scheduling (only the former actually work for now).
- Adapt task granularity during execution.

Motivation

SAC is...

- a high-performance computing programming language ;
- designed to take advantage of multi-core computers using powerful implicit parallelization.

SAC is not...

- a general purpose programming language.

- You don't want to write your GUI or your network code in SAC.
- You do want to write your computation-intensive code (e.g., image manipulation) in SAC.

Problem

SAC code called from C is not currently parallelized.

- Implementing parallelism for SAC would be quite straightforward if it only needs to support sequential C programs.
- But we want to launch several different SAC computations at the same time (e.g., in S-Net boxes, or in a program with a GUI).

Solution

What?

- Let the C programmer choose how many cores he wants to use for a SAC computation.

First idea

- Initialize a new runtime for each call.
- Discarded for efficiency concerns.

Second idea

- Use an external SAC daemon which can setup *virtual runtime environment* upon request.
- Discarded for efficiency concerns.

Final solution

Final idea

- Host the SAC daemon inside the C program.

How?

- Create a notion of SAC *runtime resources*.
- The server is replaced by a state of the resources (global data).
- Let the programmer dynamically asks for some resources.
- Use resources to create SAC virtual runtime environments.
- Be able to launch a parallelized computation in a virtual runtime environment.

SAC resources

A resource consist of...

- a number of threads ;
- their “physical” ids ;
- a pointer to an SPMD function ;
- an SPMD frame ;
- an SPMD barrier.

There’s also a global array which for each “physical” thread has:

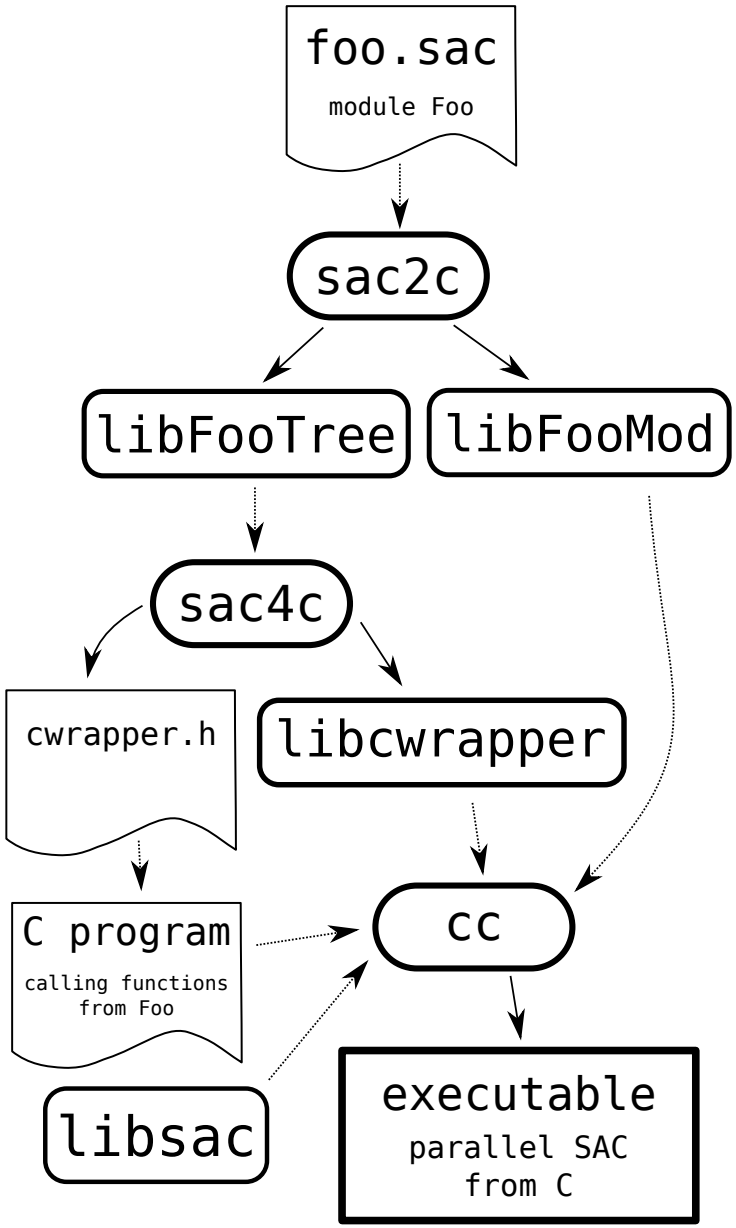
- a flag for the start barrier ;
- a pointer to the resource which currently “owns” it ;
- it’s virtual id inside the said resource.

XT functions

XT Multi threaded context, might go parallel.

SPMD functions used here are not exactly the same as those used by ST functions: we need to setup the virtual runtime environment using the SAC resources.

SAC compiler toolchain



C interface

- Initialisation of the SAC runtime system.
- Requesting and releasing resources.
- Wrapper for SAC XT functions.

Example

Context

- The `SAC_PARALLEL` environment variable is set to 16 (because we have 16 cores).
- We have a SAC module called `Foo` exposing a `bar` and a `quux` functions.
- `Foo` has been compiled with `sac4c -xt`, creating a C wrapper lib for `Foo` and the "cwrapper.h" header file.
- `SAC_InitRuntimeSystem(argc, argv);` has been called in the main function at the beginning of the program.

Example

In some thread

```
int data[W * H], *vec;
SACresources *rcs;
SACarg *arg, *res;

/* populating the data array... */

/* we want to use only 10 cores */
rcs = SAC_RequestResources(10);

arg = SACARGconvertFromIntPointer(data, 2, W, H);

/* res = Foo::bar(arg); */
Foo__bar(rcs, &res, arg);

vec = SACARGconvertToIntArray(res);
```

Example

In the meantime, in a thread far far away (well, not so far actually)

```
int datum[S], *result;
SACresources *sac_resources;
SACarg *sac_datum, *sac_result;

/* populating the datum array... */

/* we can use the 6 other cores */
sac_resources = SAC_RequestResources(6);

sac_datum = SACARGconvertFromIntPointer(datum, 1, S);

/* sac_result = Foo::quux(sac_datum); */
Foo__quux(sac_resources, &sac_result, sac_datum);

result = SACARGconvertToIntArray(sac_result);
```

The end

Questions?