Generating Message-Passing Parallel Programs from Abstract Specifications by Partial Evaluation

Christoph Herrmann
University of Passau

Talk at Imperial College, London
September 28, 2004

Overview

- Motivation
- Meta-programming in MetaOCaml
- Formal treatment of parallelism
- Implementation of parallel skeletons
  - example: divide-and-conquer
  - special instance: long number multiplication
- Conclusions

Objectives

(1) Efficient parallel target programs

(2) Short development time

(3) Ease of use without experience in parallelism

(4) Software quality

Objective 1: efficient parallel target programs

- Parallel programming patterns
  - predefined
  - parameterized

Objective 1: efficient parallel target programs

- Parallel programming patterns
  - predefined
  - parameterized
- Meta-programming / specialization, to avoid overhead for
  - copying and reorganization of data
  - case analysis

Objective 2: short development time

- Automatic program generation
- Use like a library

Objective 3: ease of use without experience

- Focus on functionality
- Hiding of operational view

Objective 4: software quality

- Type safety
- Semantic definition guides implementation
- Functional cost model

Modern Approaches to Structured Parallelism

- Programming languages with annotations (HPF)
- Libraries for structured parallel processing (MPI)

Modern Approaches to Structured Parallelism

- Programming languages with annotations (HPF)
- Libraries for structured parallel processing (MPI)
- Programming languages with skeletons
  - dedicated (HDC, P3L, PMLS)
  - embedded (based on: C++, MetaOCaml)

Host Languages for Embedding Parallel Skeletons

	C++	(Meta)OCaml
industrial use	widespread	very rare
compilation speed	slow	fast
code performance	good	good
run-time code-generation	explicit	default

Host Languages for Embedding Parallel Skeletons

	C++	(Meta)OCaml
industrial use	widespread	very rare
compilation speed	slow	fast
code performance	good	good
run-time code-generation	explicit	default

Choice: MetaOCaml

Overview

- Motivation
- Meta-programming in MetaOCaml
- Formal treatment of parallelism
- Implementation of parallel skeletons
  - example: divide-and-conquer
  - special instance: long number multiplication
- Conclusions

Meta-Programming

is the

- analysis
- transformation
- generation

of object-programs by meta-programs

Meta-Programming Extensions to OCaml

brackets (`.< >.`): enclose object program part

```
# let a = .< 2*4 >. ;;
val a : ('a, int) code = .<(2 * 4)>.
```

Meta-Programming Extensions to OCaml

brackets (`.< >.`): enclose object program part

```
# let a = .< 2*4 >. ;;
val a : ('a, int) code = .<(2 * 4)>.
```

escape (`.~`): inserts object program part

```
# let b = .< 9 + .~a >. ;;
val b : ('a, int) code = .<(9 + (2 * 4))>.
```

Meta-Programming Extensions to OCaml

brackets (`.< >.`): enclose object program part

```
# let a = .< 2*4 >. ;;
val a : ('a, int) code = .<(2 * 4)>.
```

escape (`.~`): inserts object program part

```
# let b = .< 9 + .~a >. ;;
val b : ('a, int) code = .<(9 + (2 * 4))>.
```

run (`.!`): evaluates object program part

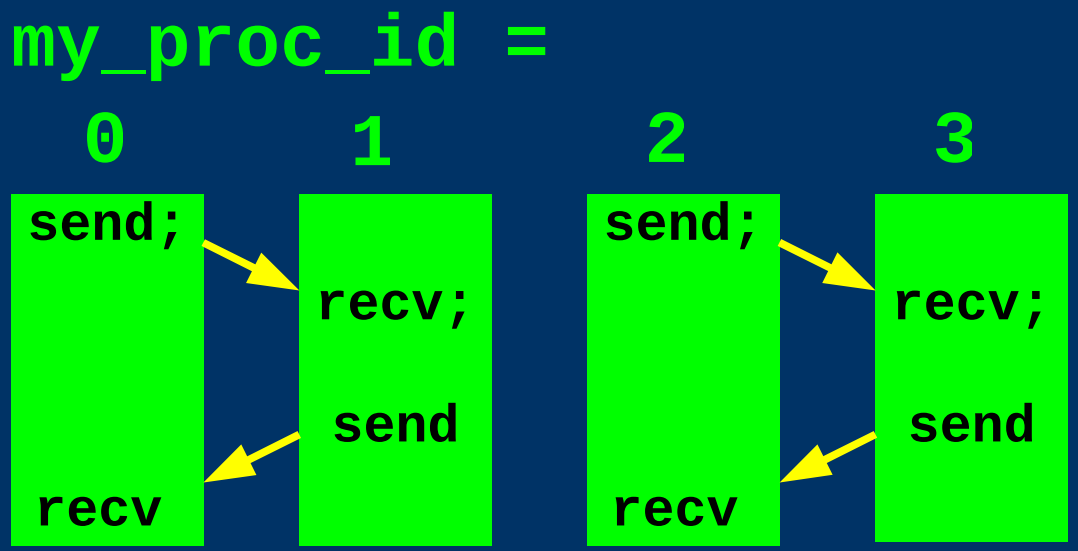
```
# let c = .!b ;;
val c : int = 17
```

Meta-Programming in Parallel

one meta-program → many object-programs: one for each process

simple example

```
if even(my_proc_id)
   then .< send;
           recv >.
   else .< recv;
           send >.
```



Overview

- Motivation
- Meta-programming in MetaOCaml
- Formal treatment of parallelism
- Implementation of parallel skeletons
  - example: divide-and-conquer
  - special instance: long number multiplication
- Conclusions

A Specification Language for Parallelism

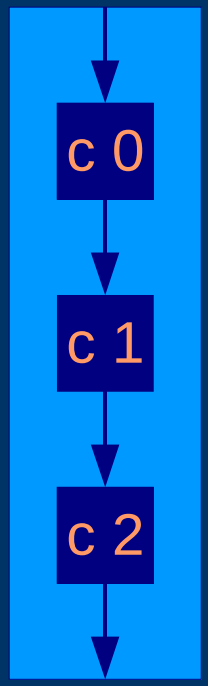
```
exp
::= Atom of sequential part
  | Comm of communications
  | Seq  of sequential composition of exp
  | Par  of parallel composition of exp
```

Sequential / Parallel Composition

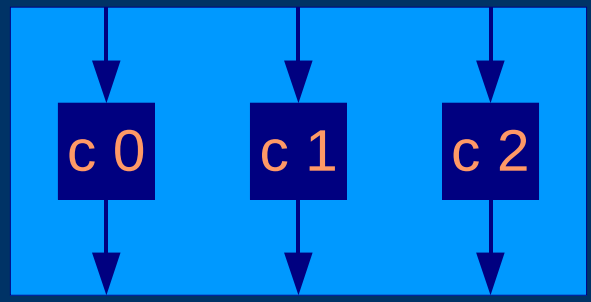
`n: int` *number of parts*

`c: int→exp` *specification of each part*

`Seq(3,c)`

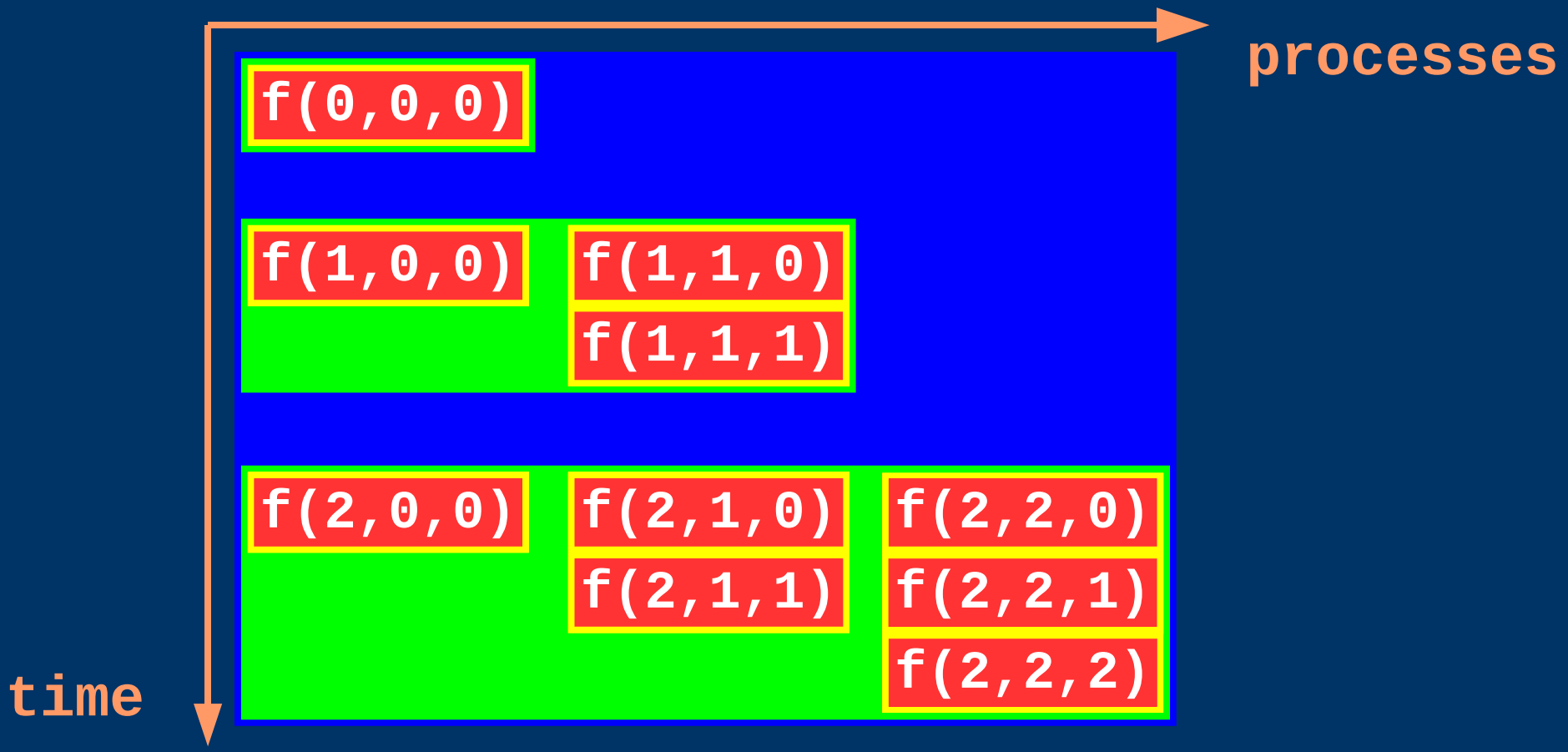


`Par(3,c)`



Example: Nested Seq and Par

```
Seq (3,fun s -> Par (s+1, fun p ->
        Seq(p+1, fun t -> Atom (f (s,p,t))))))
```



Communications

`Comm (n,c,.)`

`n: int` *number of point-to-point transmissions*

`c: int → commrec` *specification of each transmission*

`commrec`

- *par-index and buffer index of sending process*
- *par-index and buffer index of receiving process*
- *tag for message distinction*

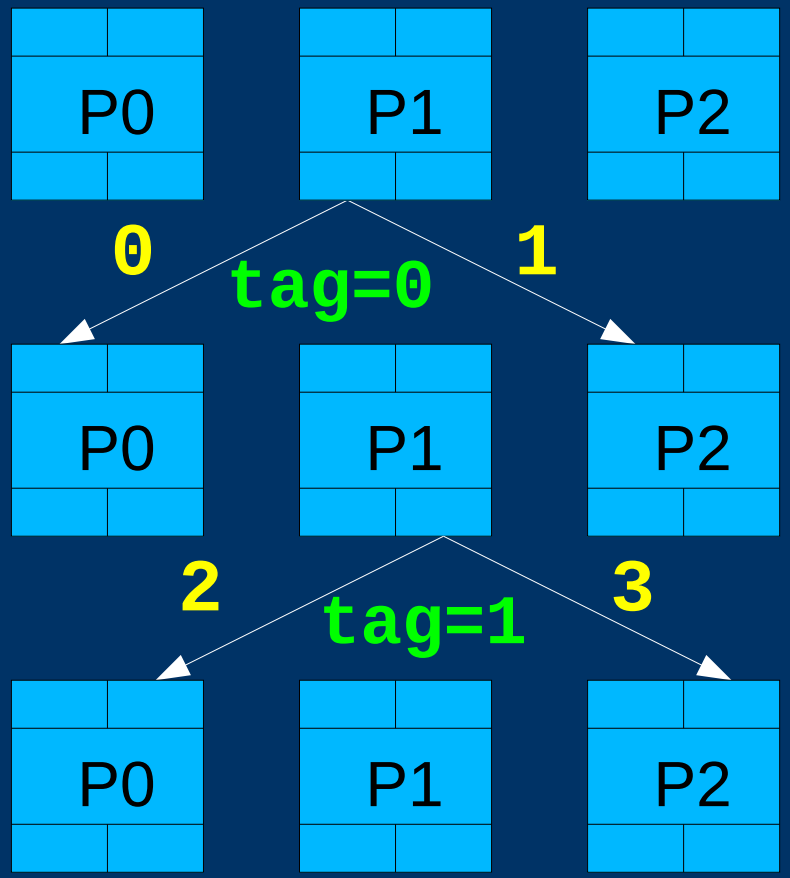
par-index: index in the smallest enclosing **`Par`**

Communications (Example)

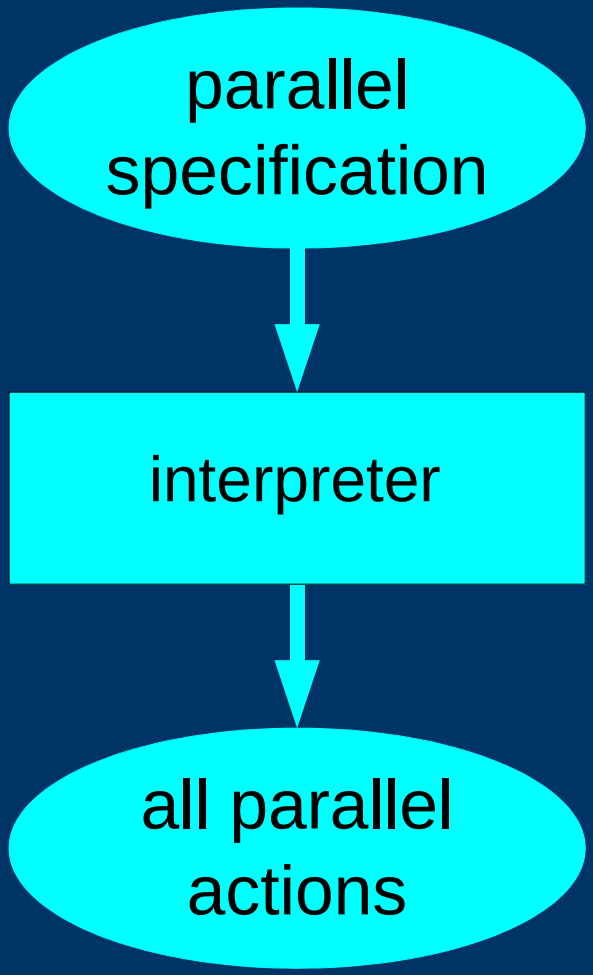
```
Par (3, fun i → Comm (4,c i,.))
```

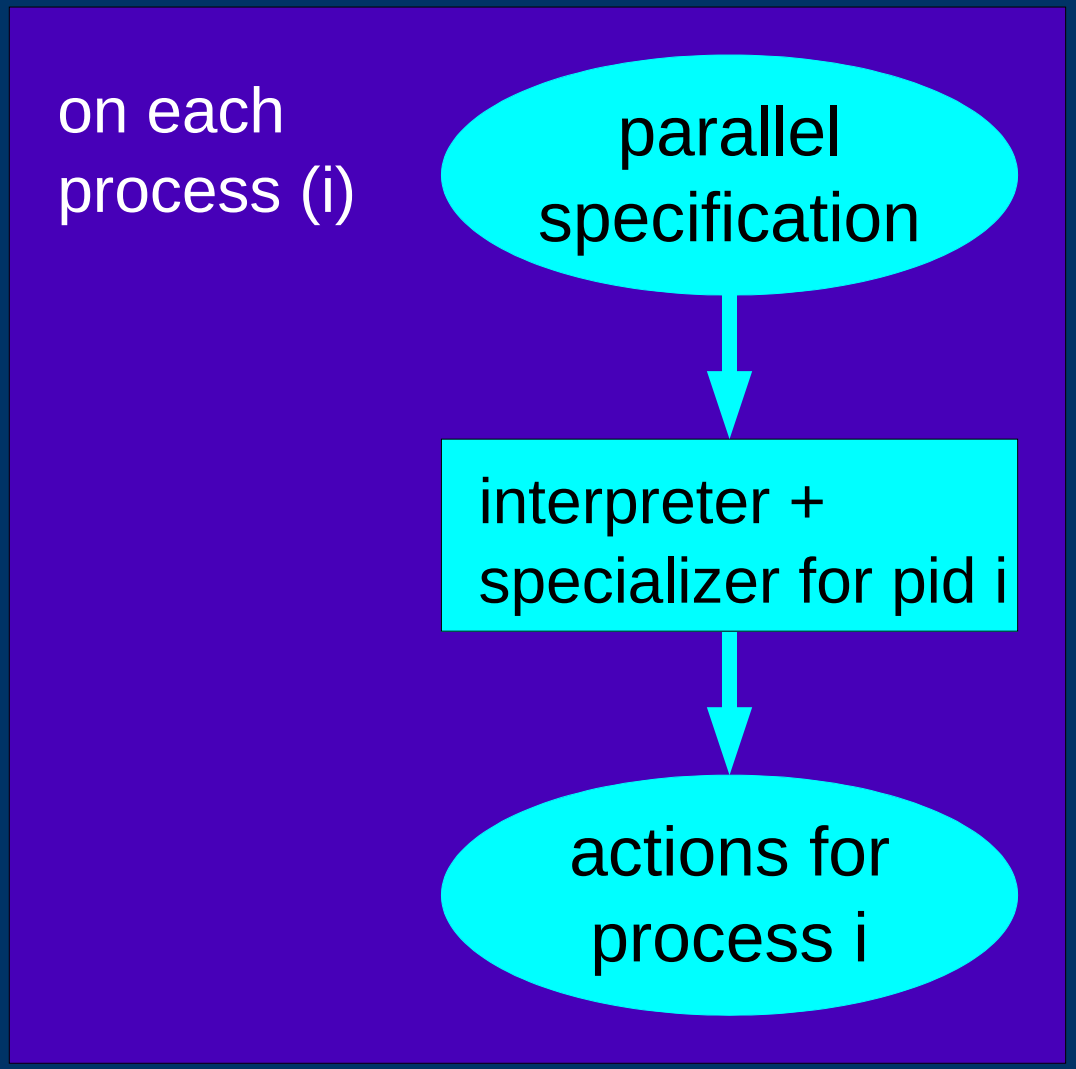
i	sender		receiver		tag
	pid	buf	pid	buf	
0	1	0	0	0	0
1	1	0	2	0	0
2	1	1	0	1	1
3	1	1	2	1	1

pid: *par-index*
buf: *buffer index*



Specification → Parallel Program

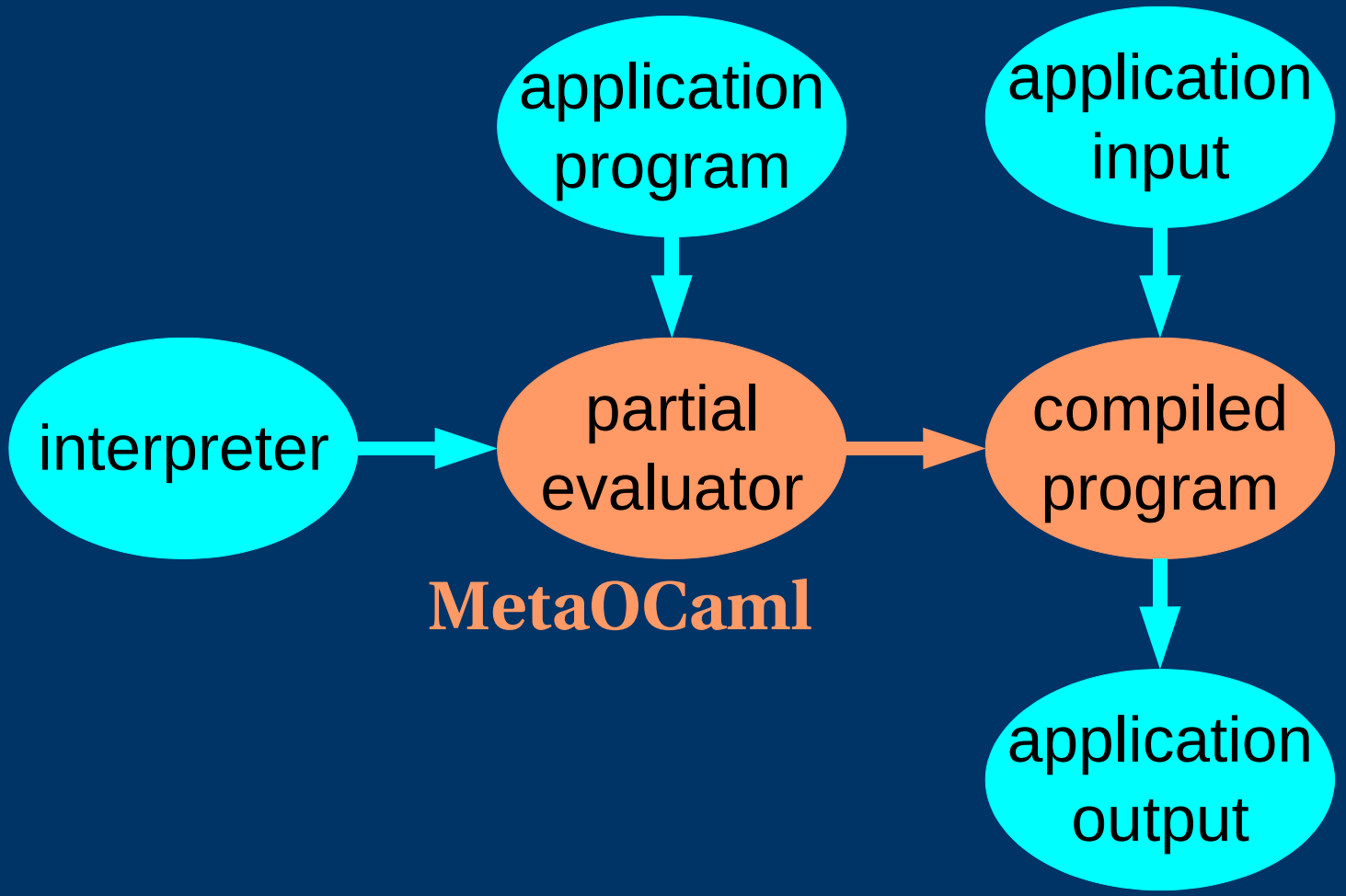




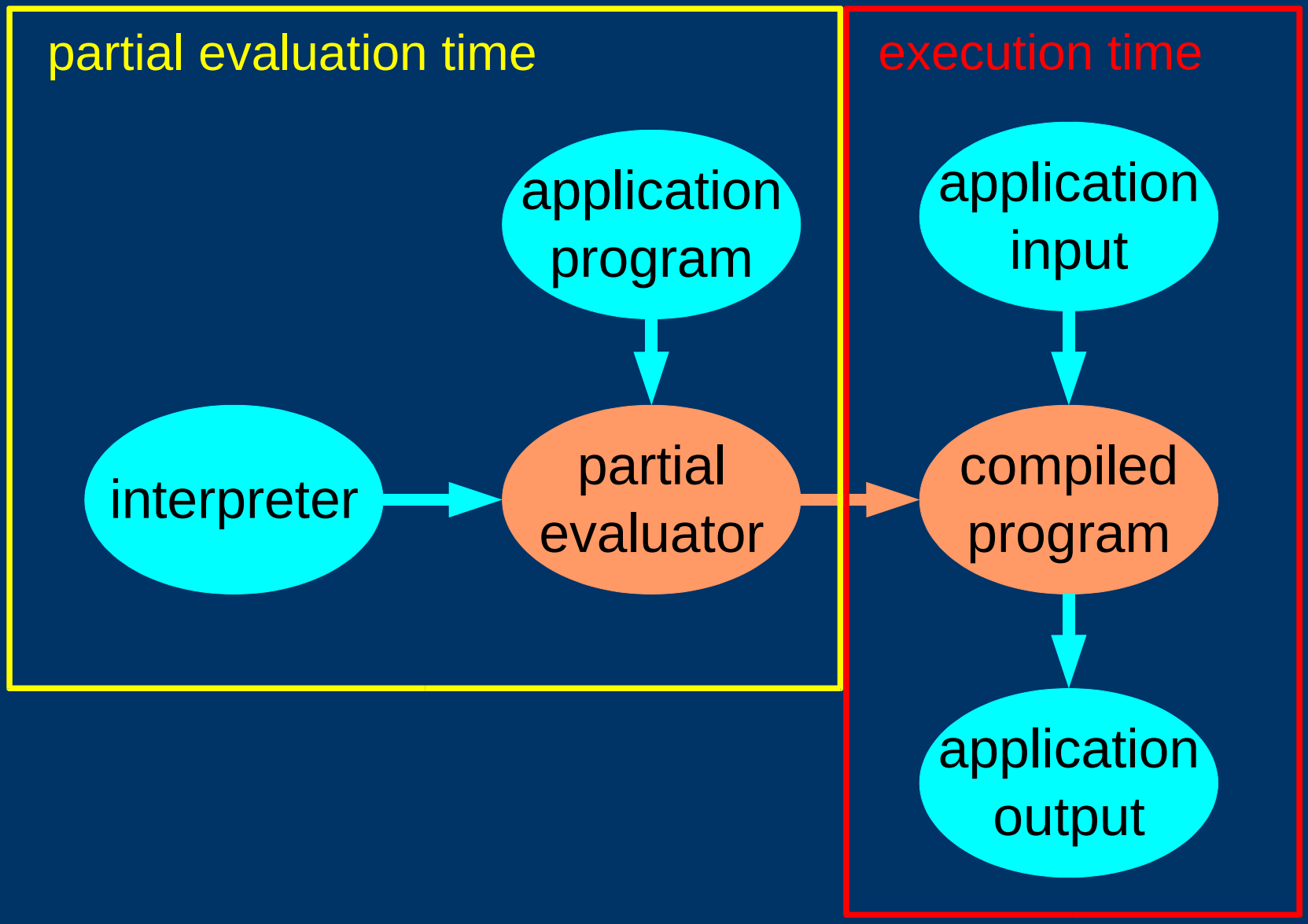
Size of Interpreter & Specializer

```
let rec interpret spec relpart submasters = match spec with
  Atom addCode
   -> if myrank=submasters.(relpart) then addCode else id
| Seq (0,f)
   -> id
| Seq (n,f) when n>0
   -> let interp s = interpret s relpart submasters
      in fun x -> interp (f (n-1)) (interp (Seq (n-1,f)) x)
| Par (0,f)
   -> id
| Par (n,f) when n>0
   -> let partadrs = Array.make n 0
      and base = ref submasters.(relpart)
      and mypart = ref 0 in
      for i=0 to n-1 do
         let (_,_,u) = wdu (f i) in
         partadrs.(i) <- !base;
         if !base<=myrank && !base+u>myrank then mypart := i;
         base := !base + u
      done;
      interpret (f !mypart) !mypart partadrs
| Comm (n,ci,buffers)
   -> fun preCode
         ->
         if myrank!=submasters.(relpart)
         then preCode
         else
          let step acc i =
             let c = ci i in
             if c.source = relpart
             then send buffers (c.sindex)
                       (submasters.(c.dest)) (c.ctag) acc
             else if c.dest = relpart
                  then receive buffers (c.dindex)
                               (submasters.(c.source)) (c.ctag) acc
                  else acc
          in
          let commCode = fold_left step .<()>. (fromto 0 (n-1))
          in .< begin let y = .~preCode in .~commCode ; y end >.
```

Interpretation + Partial Evaluation = Compilation



Two-Stage Parallel Programming



Cost Model Used in the Specialization

	w: work	d: depth	u: usedPs
`Atom _` `Comm _`	1	1	1
`Seq`(n,f)	$\mathrm{sum}_{0 \le i < n}\ \mathrm{w}(f\,i)$	$\mathrm{sum}_{0 \le i < n}\ \mathrm{d}(f\,i)$	$\mathrm{max}_{0 \le i < n}\ \mathrm{u}(f\,i)$
`Par`(n,f)	$\mathrm{sum}_{0 \le i < n}\ \mathrm{w}(f\,i)$	$\mathrm{max}_{0 \le i < n}\ \mathrm{d}(f\,i)$	$\mathrm{sum}_{0 \le i < n}\ \mathrm{u}(f\,i)$

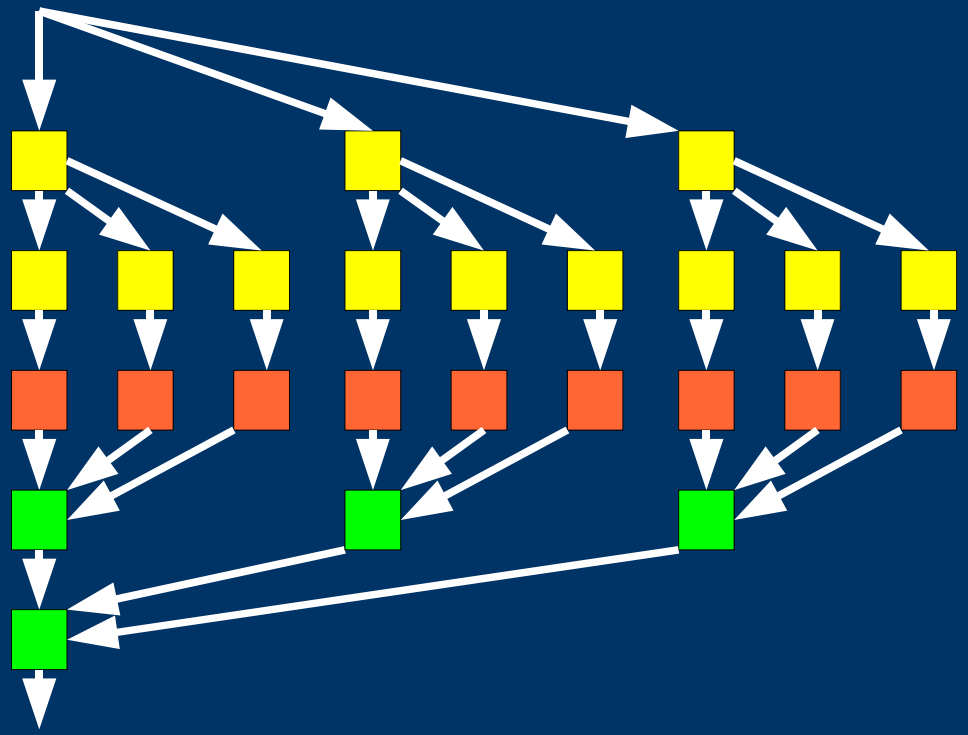
Overview

- Motivation
- Meta-programming in MetaOCaml
- Formal treatment of parallelism
- Implementation of parallel skeletons
  - example: divide-and-conquer
  - special instance: long number multiplication
- Conclusions

Specification of Divide-and-Conquer (1)

example
- recursion depth = 2
- division degree = 3

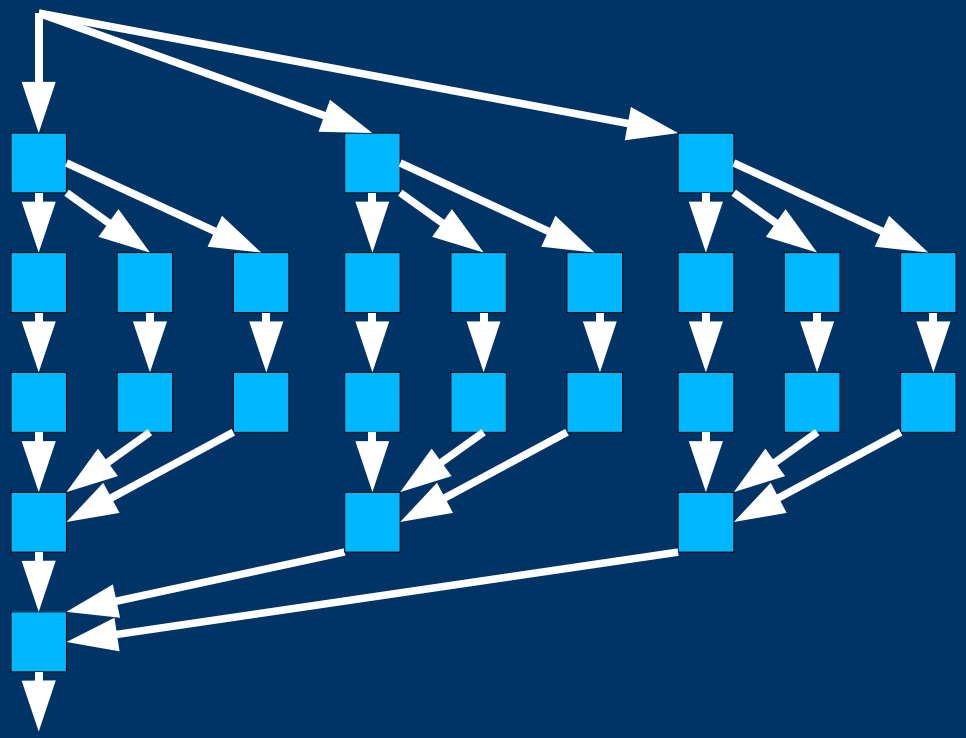
```
Par (3, ...
      Seq (_,...
            Par (3, ...
                  Seq (_, ...)
                  )))
```



Specification of Divide-and-Conquer (2)

example
- recursion depth = 2
- division degree = 3

```
Par (3, ...
      Seq (_,...
            Par (3, ...
                  Seq (_, ...)
                  )))
```



meta-program: recursive construction rule

```
dc  0           = Atom ...
dc (depth) = Par (degree, ...Seq(_,...dc (depth-1..)...)
```

Size of D&C Skeleton Implementation

```
 1 let rec dc degree basic divide combine depth  =
 2   if depth=0
 3   then Atom (fun x -> (.< let (orig,y)=.~x in (orig,basic y) >.))
 4   else Par (degree,
 5             fun mypart ->
 6  cseq
 7  [ Atom (fun x -> .< begin  let (orig,y) = .~x in
 8                             buffers.(depth) <- y;
 9                             (orig,y)                end >.);
10    Comm (degree-1,
11          (fun i -> {source=0; sindex=depth;
12                     dest=i+1; dindex=0; ctag=depth }),
13          .<buffers>.);
14    Atom (fun x -> .< let q = .~x in
15                     let (orig,y) = if mypart=0
16                                    then q
17                                    else ([],buffers.(0)) in
18                     (y::orig, divide mypart y) >.);
19    dc degree basic divide combine (depth-1);
20    Atom (fun x -> .< let q = .~x in buffers.(0) <- snd q; q >.);
21    Comm (degree-1,
22          (fun i -> {source=i+1; sindex=0;
23                     dest=0; dindex=i+1; ctag=depth }),
24          .<buffers>.);
25    Atom (fun x -> .< let q = .~x in
26                     if mypart>0
27                        then q
28                        else let (inp::orig,y) = q in
29                             let res = combine (inp,buffers) in
30                             (orig,res)
31                  >.)
32  ])
```

Overview

- Motivation
- Meta-programming in MetaOCaml
- Formal treatment of parallelism
- Implementation of parallel skeletons
  - example: divide-and-conquer
  - special instance: long number multiplication
- Conclusions

Long Number Multiplication Problem

Numbers represented in radix k (e.g., $k=10^9$)

value: $\sum_{i=0}^{n-1} c_i k^i$

Problem

- input: arrays a of size n and b of size m
- output: array c of size $m*n$ such that:

$$\left(\sum_{i=0}^{n-1} a_i k^i\right) * \left(\sum_{i=0}^{m-1} b_i k^i\right) = \sum_{i=0}^{n*m-1} c_i k^i$$

Solving Long Number Multiplication

1. Reduce problem to polynomial multiplication (abstract k to X)
2. Add carry correction to some operations (use value X=k)

Computationally expensive: 1.:

Solve polynomial multiplication by divide-and-conquer, splitting the coefficient arrays:

$$(aX^n+b)*(cX^n+d)=(a*c)X^{2n}+(a*d+b*c)X^n+b*d$$

Apply Karatsuba's division into only three subproblems:

1. $a*c$
2. $b*d$
3. $(a+b)*(c+d)$

$$a*d+b*c=(a+b)*(c+d)-a*c-b*d$$

Size of Sequential Multiplication Code

```
 1 let rec karat_seq xs ys =
 2   let n = Array.length xs and m = Array.length ys in
 3   if min m n <= 32
 4   then                              (* solution for small problem sizes *)
 5     let zs = Array.make (n+m) 0 in
 6     let cy = ref zero1 in
 7     for i=0 to n-1 do  (* polynomial multiplication kernel *)
 8       cy := zero1;                                  32→64 bit
 9       for j=0 to m-1 do
10         let sum = addl (addl !cy (of_int zs.(i+j)))
11                        (mull (of_int xs.(i)) (of_int ys.(j))) in
12         cy := div1 sum radix1;
13         zs.(i+j) <- to_int (subl sum (mull !cy radix1))
14       done;
15       zs.(i+m) <- to_int !cy      64→32 bit
16     done;
17     zs
18   else                                (* solution for large problem sizes *)
19     let low    = karat_seq (lowpartCY    xs) (lowpartCY    ys)
20     and high   = karat_seq (highpartCY   xs) (highpartCY   ys)
21     and mixed  = karat_seq (mixedpartsCY xs) (mixedpartsCY ys)
22     in seq_combine (.,low,high,mixed)
```

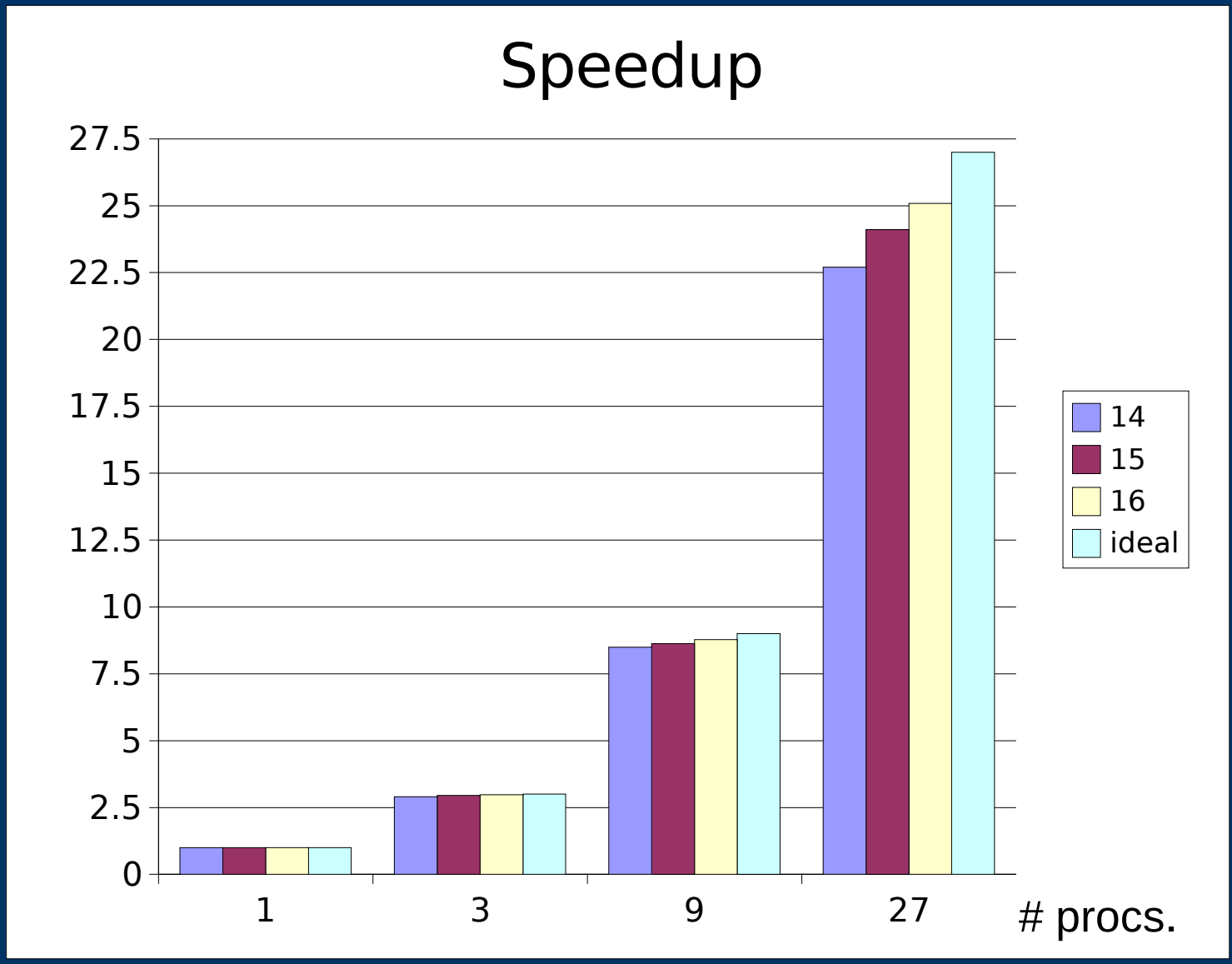
several single loops for efficient carry correction

Results (Number Multiplication):

execution times in seconds on the hpcLine cluster at Passau

log. problem size		14	15	16
sequential	native code	22.6	68.8	216.0
	bytecode	49.9	151.9	464.2
parallel bytecode for given number of processes	1	50.4	152.1	462.0
	3	17.1	51.5	155.9
	9	5.9	17.6	52.9
	27	2.2	6.3	18.5

Results (Number Multiplication):



Overview

- Motivation
- Meta-programming in MetaOCaml
- Formal treatment of parallelism
- Implementation of parallel skeletons
  - example: divide-and-conquer
  - special instance: long number multiplication
- Conclusions

Conclusions

- Partial evaluation after program loading is useful
  - it permits more abstraction without overhead, a powerful technique for implementing skeletons
  - the time for partial evaluation was negligible

Conclusions

- Partial evaluation after program loading is useful
  - it permits more abstraction without overhead, a powerful technique for implementing skeletons
  - the time for partial evaluation was negligible
- MetaOCaml is recommended
  - fast partial evaluation in type-safe form
  - functional style for abstracting (e.g. indices)
  - imperative style for efficient program kernels
  - JIT and native-code compilers are on the way

Further Research

- Languages used in industry
  - investigate potential of other meta-programming systems, e.g., C++ templates
  - mimic the desired meta-programming features by program generators

Further Research

- Languages used in industry
  - investigate potential of other meta-programming systems, e.g., C++ templates
  - mimic the desired meta-programming features by program generators
- Compile-time partial evaluation
  - program construction via specializing components

Further Research

- Languages used in industry
  - investigate potential of other meta-programming systems, e.g., C++ templates
  - mimic the desired meta-programming features by program generators
- Compile-time partial evaluation
  - program construction via specializing components
- Problem adaptation at run time
  - use multiple staging in Ocaml:
    further code can be generated at run time to deal with the current requirements

Thank you for your attention!

Project page containing program sources

http://www.infosun.fmi.uni-passau.de/cl/metaprog/