Using MetaOCaml to Implement Skeletons and Domain-Specific Languages

Christoph Herrmann
University of Passau

Talk at Texas A&M University
College Station/TX
March 10, 2005

Overview

- Metaprogramming with MetaOCaml
- Implementation of Parallel Skeletons
- Domain-Specific Language Implementation
- Run-time and Compile-time Metaprogramming

Meta-Programming

is the

- analysis
- transformation
- generation

of object-programs by meta-programs

Motivation for Metaprogramming

- A high degree of abstraction often means:
  - data objects: dynamic memory allocation
  - operations: subroutine call or even interpreted

  ⇒ detrimental for performance
- High performance often requires:
  - contiguous data objects (fitting into the cache)
  - operations supported by the hardware
- Metaprogramming: abstraction and performance
  - metaprogram: abstract, maybe slow
  - object program: efficient, maybe low-level

Motivation for MetaOCaml

- Type Safety
  only type-correct programs can be generated
- Multiple Staging, Homogeneous Metaprogramming
  generated programs can generate programs
- Run-time Metaprogramming
  program generation can depend on run-time data
- Abstraction (for program generators)
  higher-order polymorphic functions
- Performance (for final programs)
  loop programs working on arrays

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
```

MetaOCaml Example: Power Function

unstaged:

```
let rec power (n, x) =
  match n with
     0 -> 1
   | n -> x * (power (n-1, x));;
```

staged:

```
let rec power' (n, x) =
  match n with
     0 -> .< 1 >.
   | n -> .< .~x * .~(power' (n-1, x))>.;;
```

apply staged:

```
power' (3, .< x >.);;
```

result:

```
.< x*x*x*1 >.
```

Overview

- Metaprogramming with MetaOCaml
- Implementation of Parallel Skeletons
- Domain-Specific Language Implementation
- Run-time and Compile-time Metaprogramming

Notion of Skeleton

- Predefined program schema
- Use: like a library function, preferably abstract
- Implementation:
  - uses expert knowledge
  - structure not affected by a compiler
- Covering an entire algorithmic paradigm, examples:
  - Reduce (summation, product, maximum)
  - Scan (carry lookahead)
  - Dynamic Programming (CYK, optimal parenthesization)
  - Divide & Conquer (sorting, matrix multiplication)
  - Branch & Bound (travelling salesperson)

Parallel Metaprogramming: Objectives

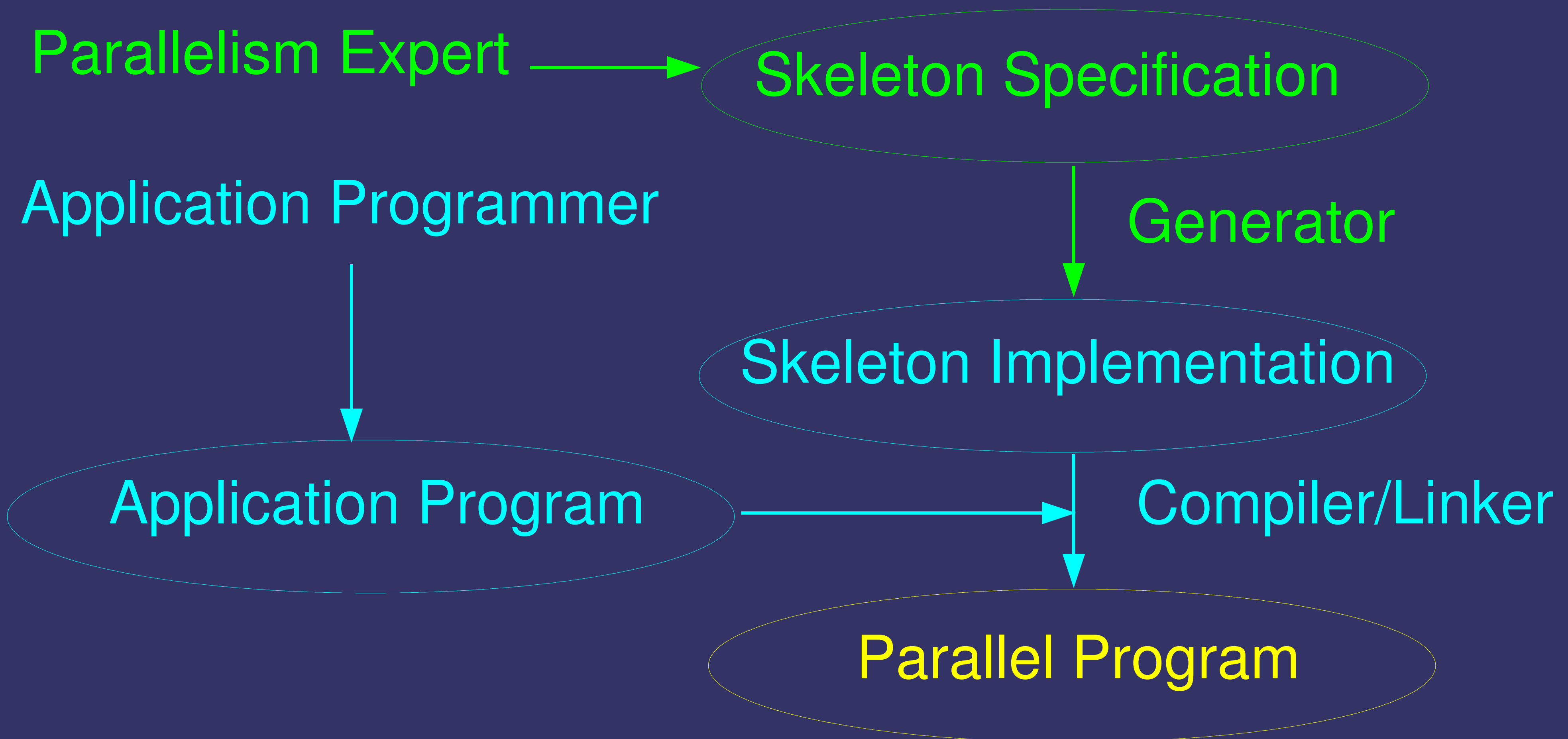
(1) Efficient parallel target programs
- → Predefined parameterized patterns
- → Program specialization

(2) Short development time
- → Automatic program generation

(3) Ease of use without experience in parallelism
- → Application programmer: functional view

(4) Software quality
- → Type safety
- → Cost model

Metaprogramming Parallel Skeletons



Meta-Programming Parallel Skeletons

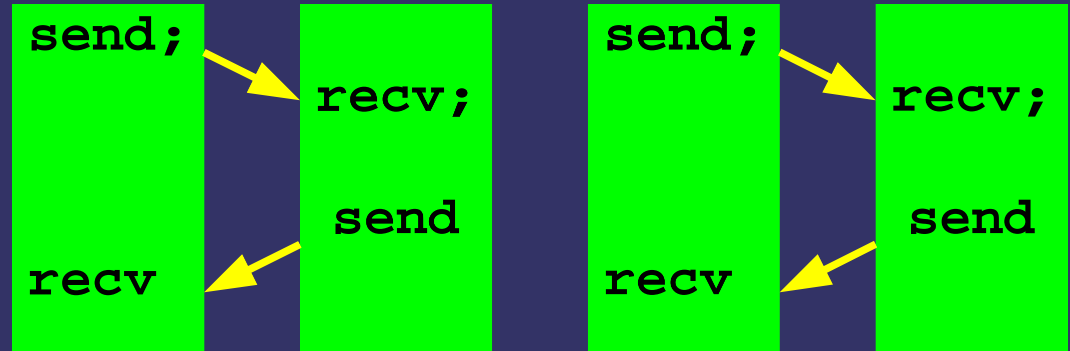
one meta-program

many object programs

simple example:

my_proc_id =

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



A Specification Language for Parallelism

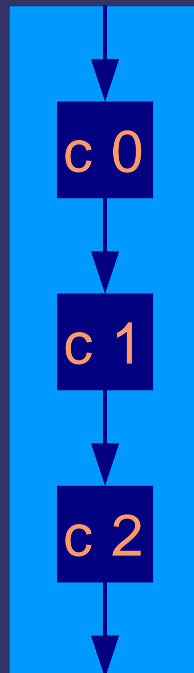
```
exp
::= Atom of sequential part
  | Comm of communications
  | Seq  of sequential composition of exp
  | Par  of parallel composition of exp
```

Sequential / Parallel Composition

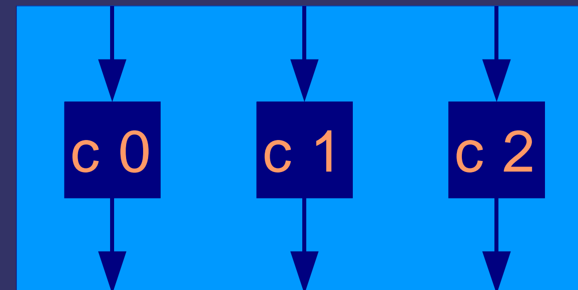
`n: int` *number of parts*

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

`Seq(3,c)`



`Par(3,c)`

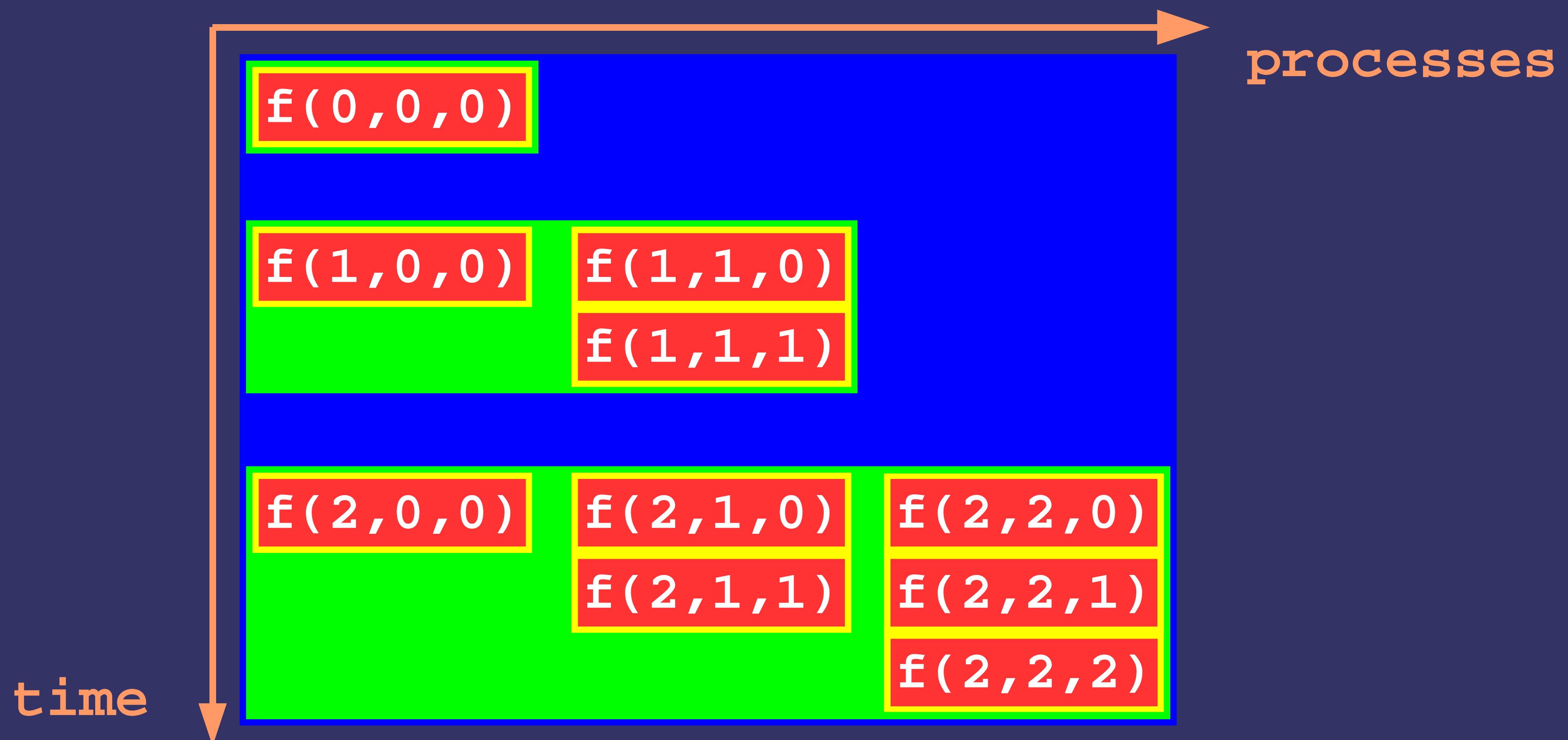


Cost Model Used in the Specialization

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

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`

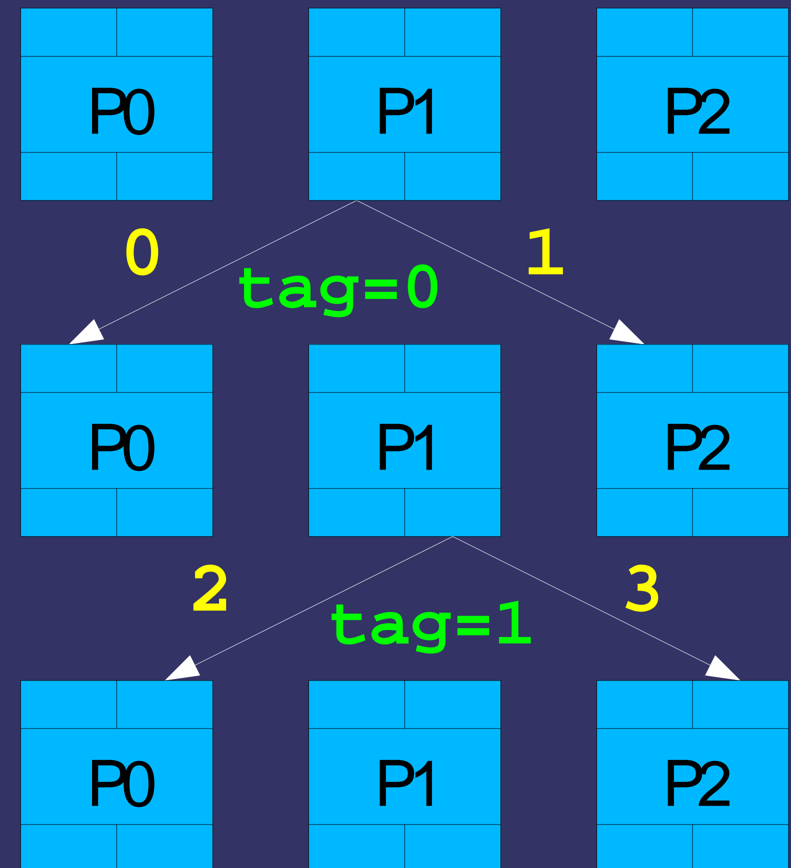
- *par-index and buffer index of sending process*
- *par-index and buffer index of receiving process*
- *tag for message distinction*

<u>par-index:</u> index in the smallest enclosing `Par`

Communications (Example)

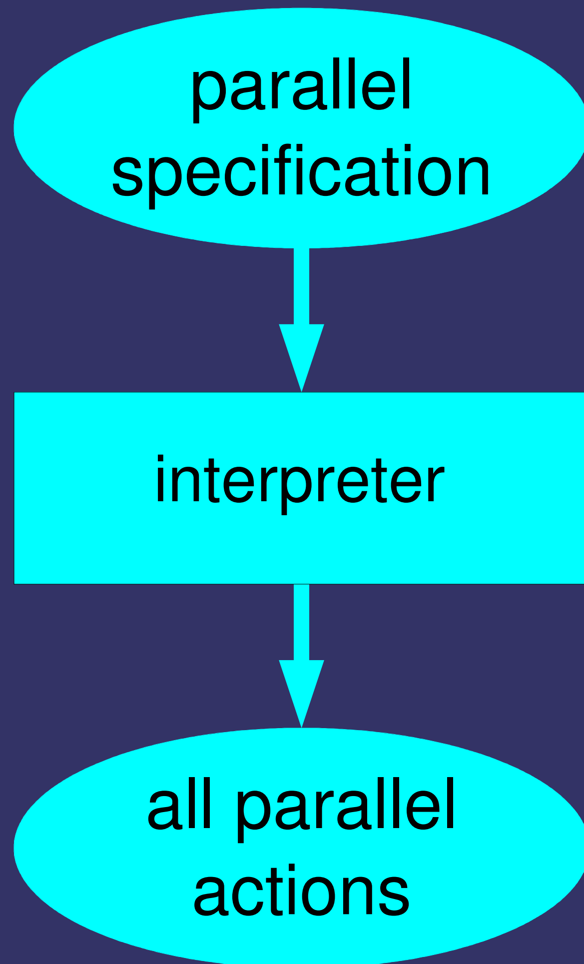
```
Par (3, fun i → Comm (4,c i,.))
```

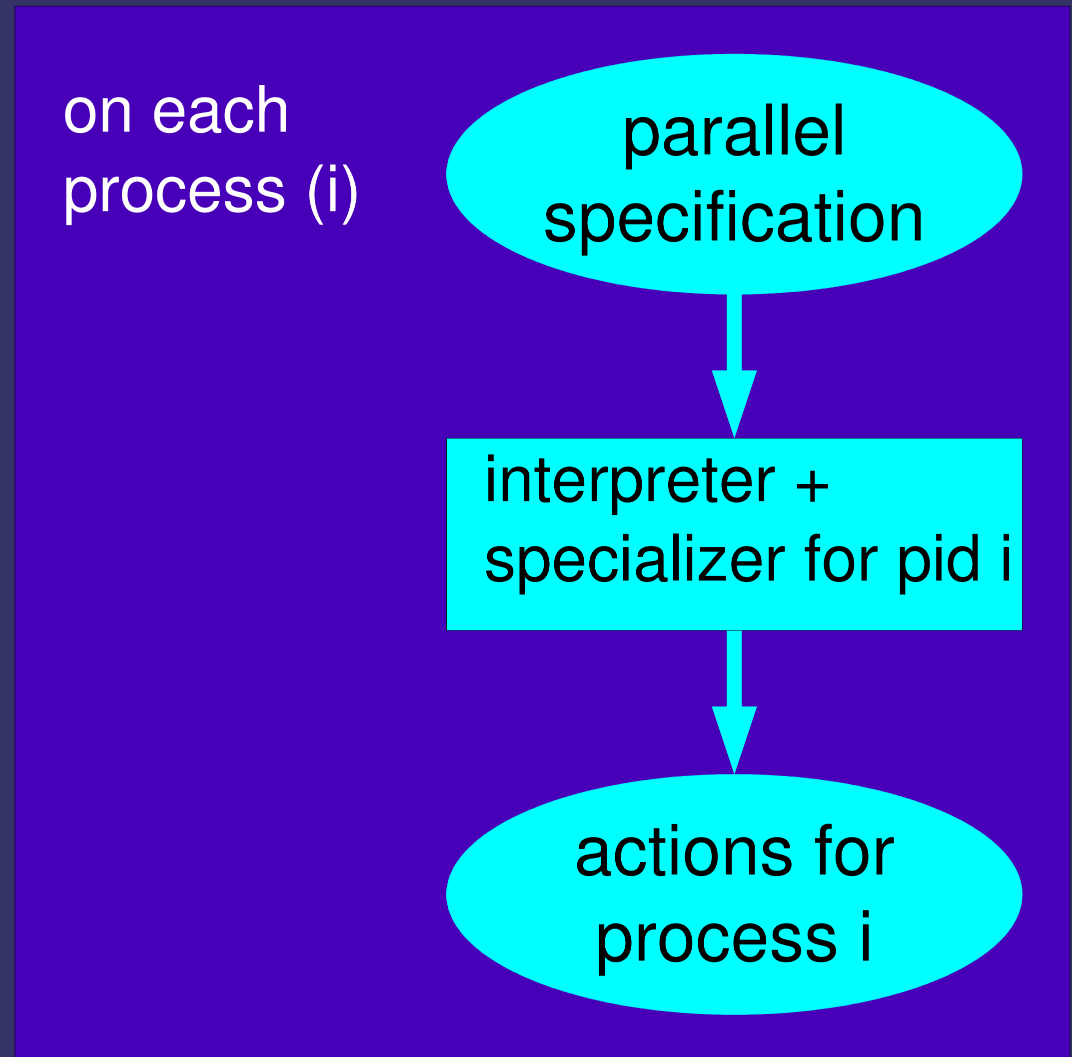
i	sender pid	sender buf	receiver pid	receiver buf	tag
0	1	0	0	0	0
1	1	0	2	0	0
2	1	1	0	1	1
3	1	1	2	1	1



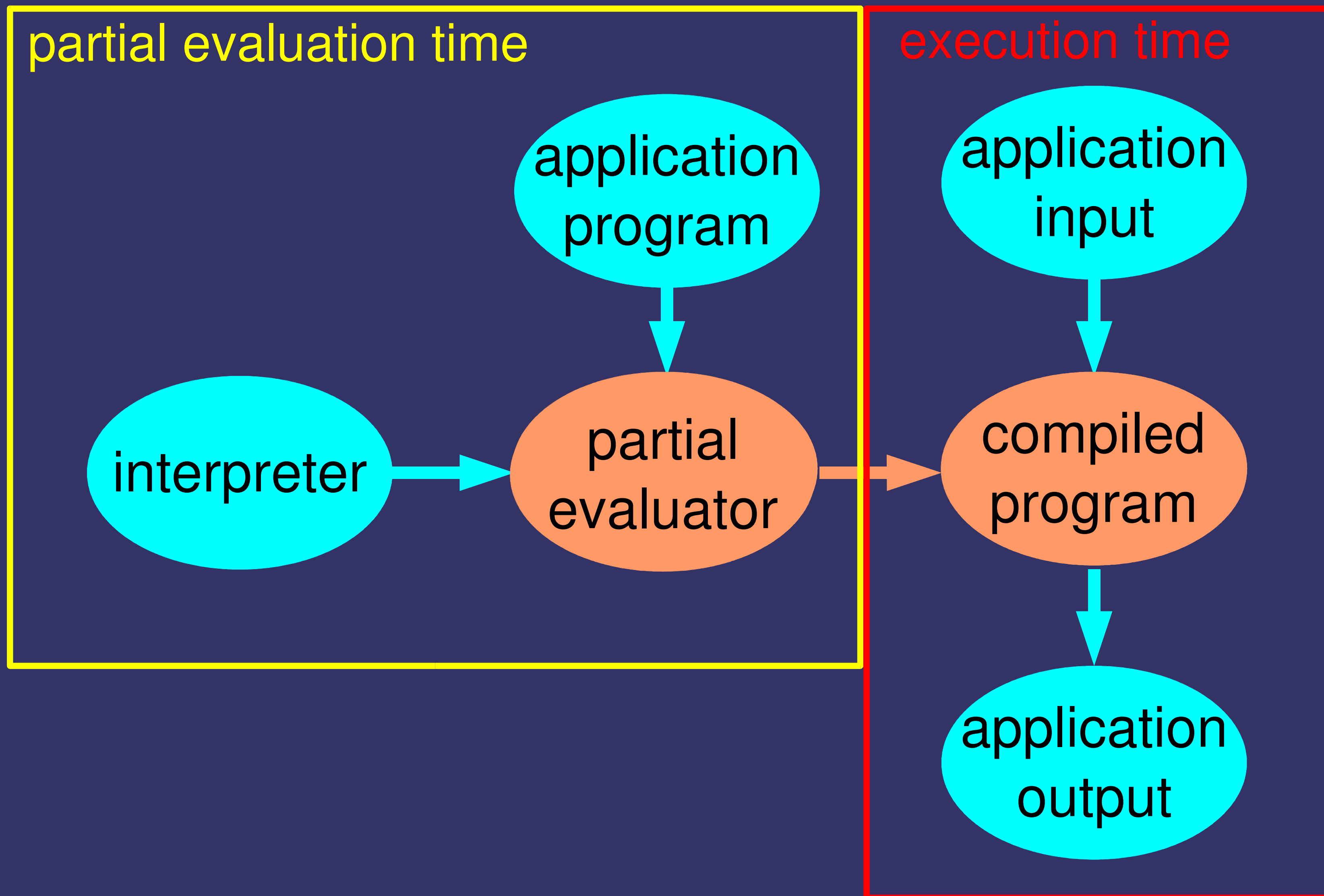
pid: *par-index*
buf: *buffer index*

Specification → Parallel Program

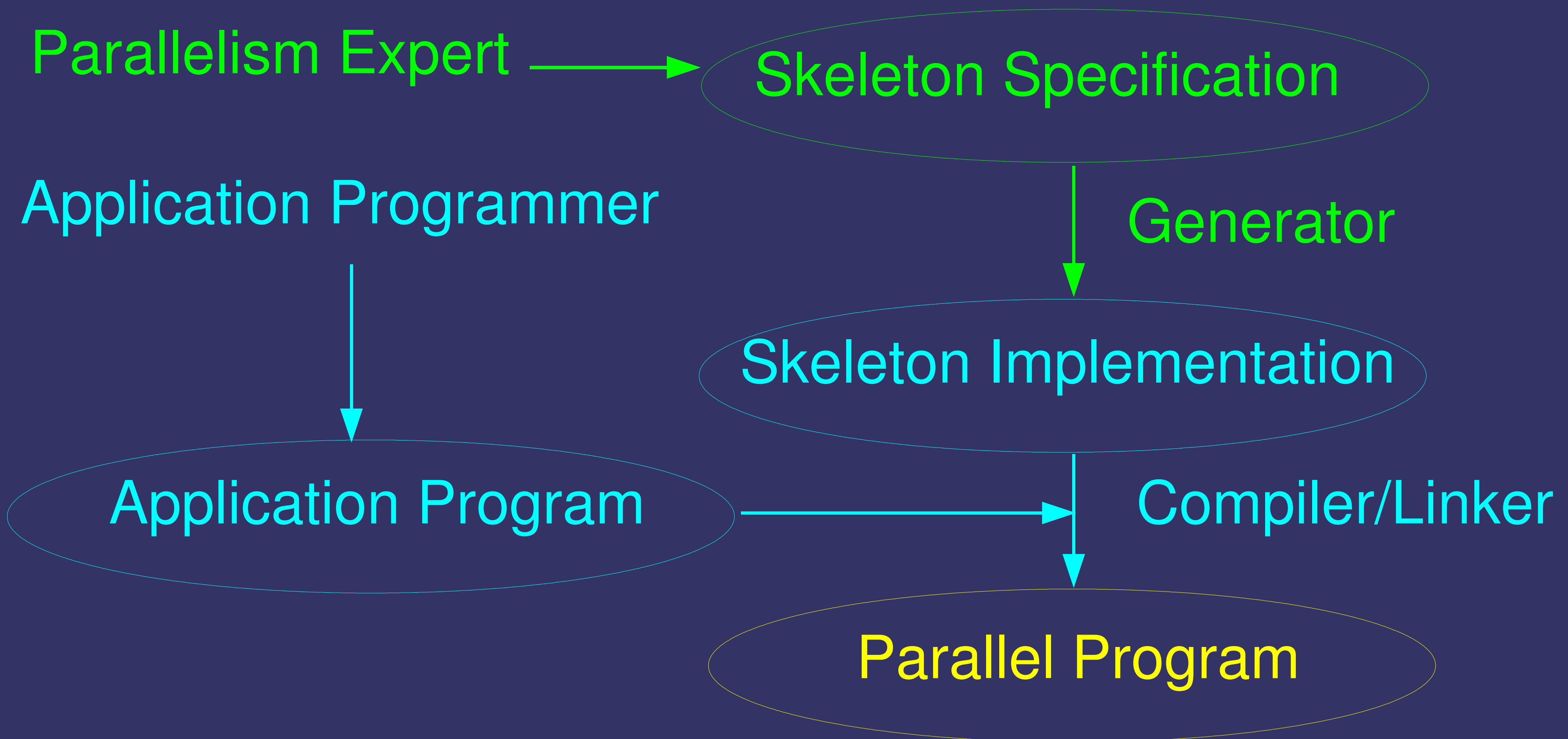




Interpretation + Partial Evaluation = Compilation



Metaprogramming Parallel Skeletons



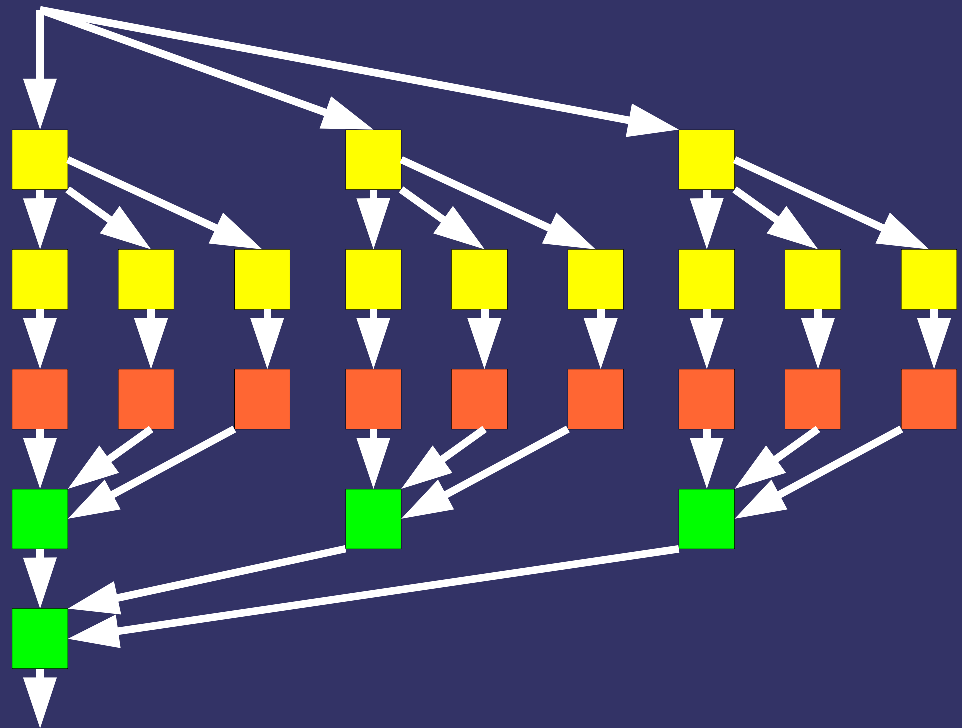
Size of Interpreter & Specializer

```
 1 let rec interpret spec relpart submasters = match spec with
 2   Atom addCode
 3    -> if myrank=submasters.(relpart) then addCode else id
 4 | Seq (0,f)
 5    -> id
 6 | Seq (n,f) when n>0
 7    -> let interp s = interpret s relpart submasters
 8       in fun x -> interp (f (n-1)) (interp (Seq (n-1,f)) x)
 9 | Par (0,f)
10    -> id
11 | Par (n,f) when n>0
12    -> let partadrs = Array.make n 0
13       and base = ref submasters.(relpart)
14       and mypart = ref 0 in
15       for i=0 to n-1 do
16          let (_,_,u) = wdu (f i) in
17          partadrs.(i) <- !base;
18          if !base<=myrank && !base+u>myrank then mypart := i;
19          base := !base + u
20       done;
21       interpret (f !mypart) !mypart partadrs
22 | Comm (n,ci,buffers)
23    -> fun preCode
24           ->
25           if myrank!=submasters.(relpart)
26           then preCode
27           else
28            let step acc i =
29               let c = ci i in
30               if c.source = relpart
31               then send buffers (c.sindex)
32                         (submasters.(c.dest)) (c.ctag) acc
33               else if c.dest = relpart
34                    then receive buffers (c.dindex)
35                                 (submasters.(c.source)) (c.ctag) acc
36                    else acc
37            in
38            let commCode = fold_left step .<()>. (fromto 0 (n-1))
39            in .< begin let y = .~preCode in .~commCode ; y end >.
```

Specification of Divide-and-Conquer (1)

example
- recursion depth = 2
- division degree = 3

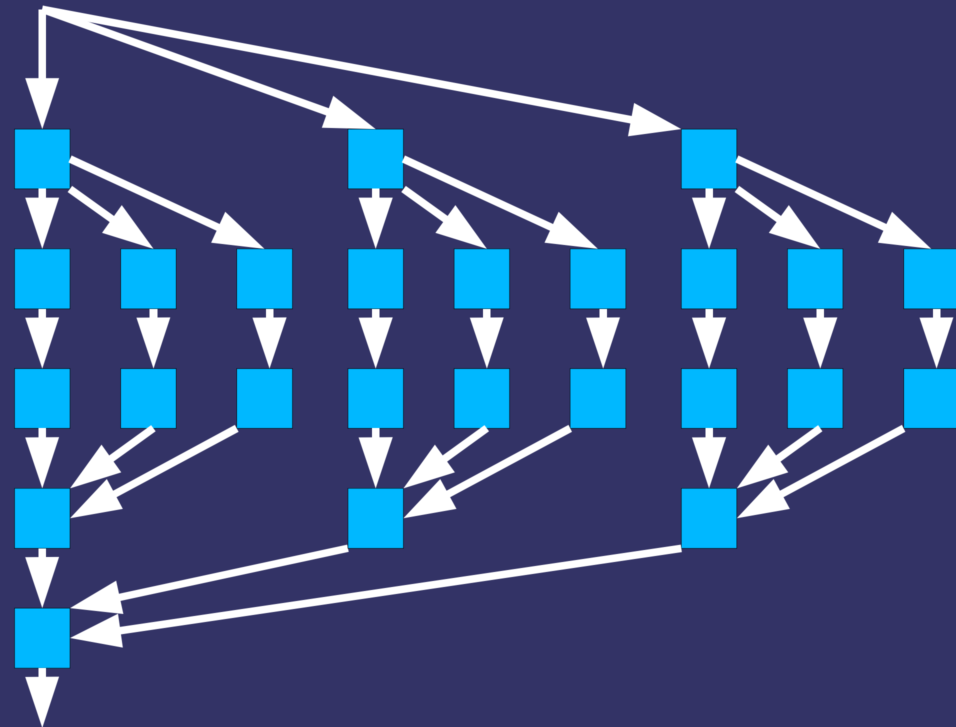
Par (3, ...
Seq (_,...
Par (3, ...
Seq (_, ...)
)))



Specification of Divide-and-Conquer (2)

example
- recursion depth = 2
- division degree = 3

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



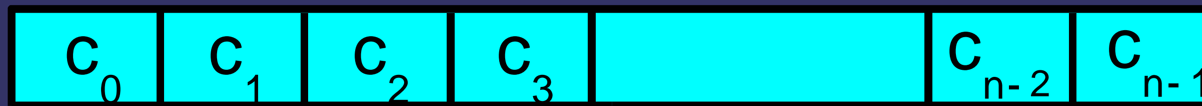
meta-program: recursive construction rule
dc 0 = Atom ...
dc (depth) = Par (degree, ...Seq(_,...dc (depth-1..)...)

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  ])
```

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 zerol in
 7     for i=0 to n-1 do  (* polynomial multiplication kernel *)
 8       cy := zerol;                                      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 := divl sum radixl;
13         zs.(i+j) <- to_int (subl sum (mull !cy radixl))
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)
```

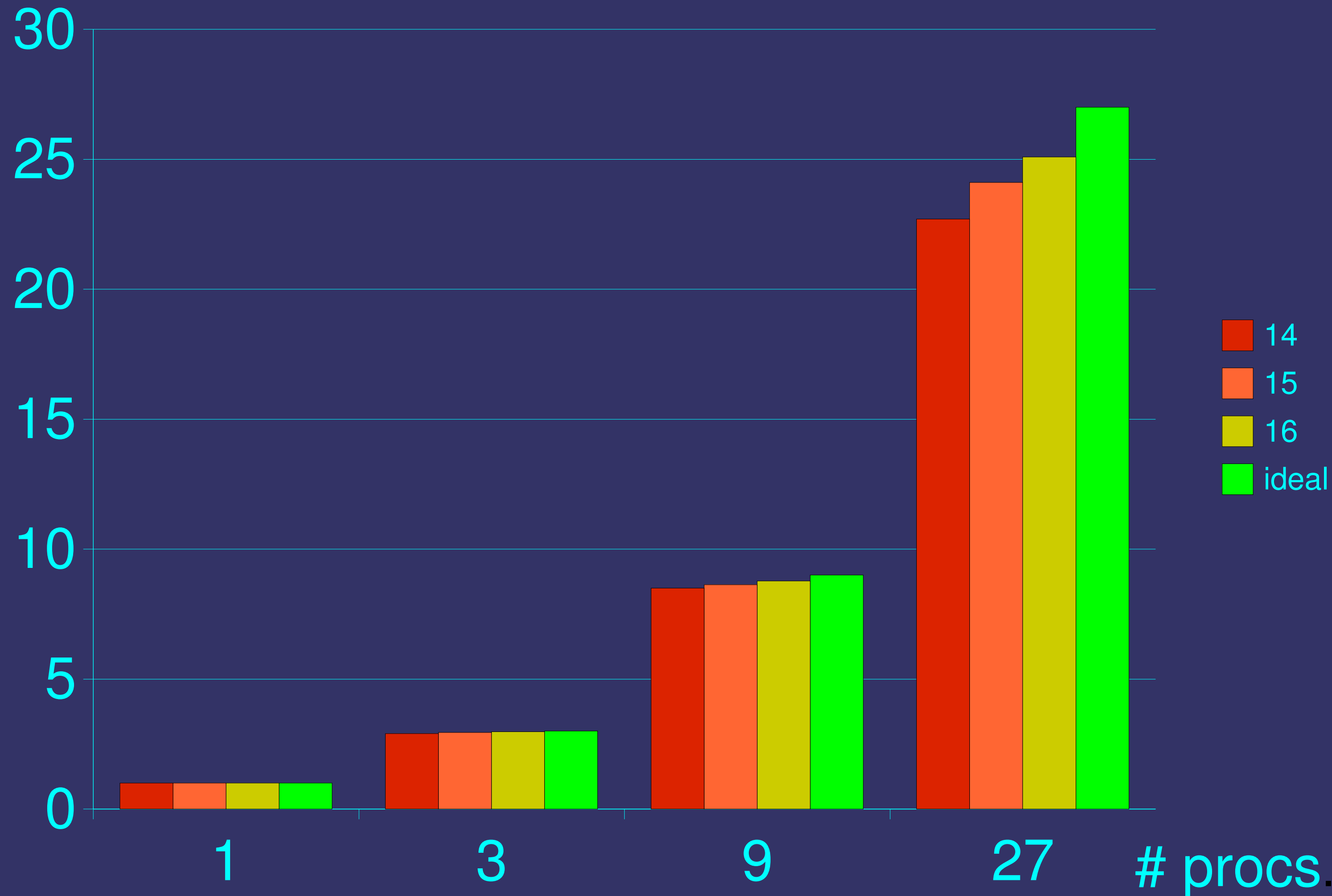
several single loops for efficient carry correction

Results (Number Multiplication):

execution times in seconds on the hpcLine cluster in 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

- Metaprogramming with MetaOCaml
- Implementation of Parallel Skeletons
- Domain-Specific Language Implementation
- Run-time and Compile-time Metaprogramming

Domain-Specific Language Implementation

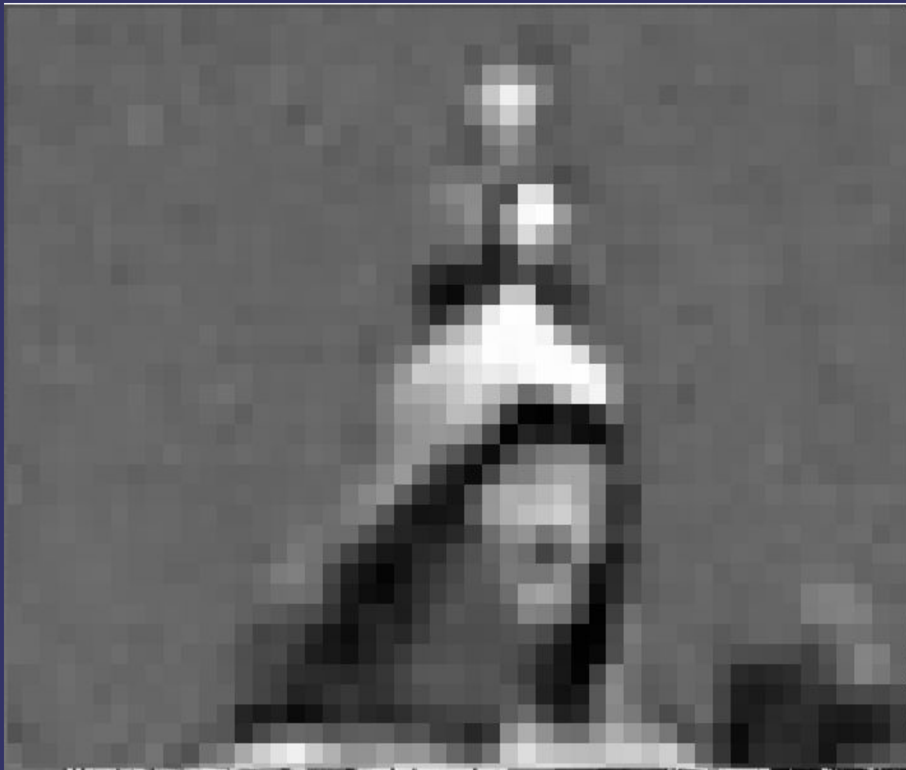
- Domains, e.g.:
  - data bases
  - high-performance scientific computations
  - signal / image processing
- Exploit Domain-Specific Laws
  - relational algebra
  - linear algebra
- Implementation
  - dedicated compiler, or
  - interpreter, partially evaluated

Image Processing Tasks

- Image Filtering
- Zoom
- Interpolation
- Dynamic Effects

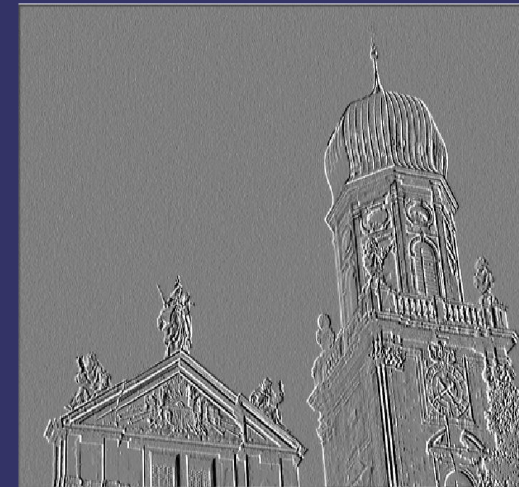
Example Filtering (Gradient)

Example Zoom / Interpolation



Example: Dynamic Effects (Mirroring in Waves)

Gradient Filtering



Filter Specification:

```
let m = [-1.0 0.0 1.0
        |-1.0 0.0 1.0
        |-1.0 0.0 1.0]
in
  [ 3 channels:
    0.5 + sum i from 0 to 2 of
             sum j from 0 to 2 of
                m[i,j] * image(row-i-1, col-j-1, current)
  ]
```

Gradient Filtering, Program Generation

```
let m = [-1.0 0.0 1.0
        |-1.0 0.0 1.0
        |-1.0 0.0 1.0]
in
  [ 3 channels:
    0.5 + sum i from 0 to 2 of
             sum j from 0 to 2 of
                m[i,j] * image(row-i-1, col-j-1, current)
  ]
```

```
.<fun (row_1, col_2)->
    let (c_3) =
      int_of_float ((0.5 +.
  (((((-1. *. ((float_of_int rast.(row_1-0-1).(col_2-0-1).chan) /. 255)) +.
       (0. *. ((float_of_int rast.(row_1-0-1).(col_2-1-1).chan) /. 255))) +.
       (1. *. ((float_of_int rast.(row_1-0-1).(col_2-2-1).chan) /. 255))) +.
    (((-1. *. ((float_of_int rast.(row_1-1-1).(col_2-0-1).chan) /. 255)) +.
       (0. *. ((float_of_int rast.(row_1-1-1).(col_2-1-1).chan) /. 255))) +.
       (1. *. ((float_of_int rast.(row_1-1-1).(col_2-2-1).chan) /. 255)))) +.
    (((-1. *. ((float_of_int rast.(row_1-2-1).(col_2-0-1).chan) /. 255)) +.
       (0. *. ((float_of_int rast.(row_1-2-1).(col_2-1-1).chan) /. 255))) +.
       (1. *. ((float_of_int rast.(row_1-2-1).(col_2-2-1).chan) /. 255)))))
 *. 255.) in
if ((c_3)<0) then 0 else if ((c_3)>255) then 255 else (c_3)>.
```

Results (Gradient Filtering)

			times in milliseconds					
compilation	filter execution	simpl.	simplify	codegen	.!	execution	total	speedup
bytecode	compiled	yes	3.39	3.22	19.26	9369.82	9395.69	20.6
		no		1.24	11.74	20372.45	20385.43	9.5
	interpreted	yes	3.4			238340.00	238343.40	0.8
		no				591319.00	591319.00	0.3
native code	compiled	yes	1.06	1.46	92.63	2496.84	2591.99	74.7
		no		0.52	66.35	3936.93	4003.80	48.4
	interpreted	yes	1.07			65147.89	65148.96	3.0
		no				193671.90	193671.90	1.0

Review on Speedups

maximum speedups obtained by

(a) eliminating interpretation overhead (×48)

(b) parallelization (×25 on 27 processors)

(c) Ocaml native-code (×5 vs. bytecode)

(d) domain-specific optimizations (×3)

speedups not composable, e.g. (a+d): ×75

Overview

- Metaprogramming with MetaOCaml
- Implementation of Parallel Skeletons
- Domain-Specific Language Implementation
- Run-time and Compile-time Metaprogramming

Run-Time Metaprogramming with the MetaOCaml Staging Mechanism

- MetaOCaml programs consist of stages
- Each stage is
  - one step of configuration, the last being the execution
  - independent of further stages (but may use their results, if it likes to)
- Example staging in parallel programming
  1. configuring the program for the machine
  2. specializing for a new kind of input data
  3. fast processing of several sets of input data
- Only type-correct code can be generated

Compile-time Metaprogramming with the C++ Template Sublanguage

- Advantages
  - Availability of C++ compilers and libraries (MPI)
  - Direct access to HW-close language parts (C)
  - Acceptance by potential commercial partners
- Disadvantages
  - No type safety
  - Compilation of the generated program can fail
  - Template resolving is slow
  - No I/O possible, potential need for preprocessing
  - No modelling of domain-specific types

Thank You for Your Attention!

Questions ?

Project page

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