Functional Meta-Programming in the Construction of Parallel Programs

Christoph A. Herrmann
University of Passau, Germany

http://www.fmi.uni-passau.de/~herrmann

Objectives

- efficient parallel target programs
  - predefined, parameterized parallel patterns
  - meta-programming to avoid overhead

Objectives

- efficient parallel target programs
- short development time
  - automatic program generation
  - use like a library

Objectives

- efficient parallel target programs
- short development time
- appropriate for application programmer without experience of parallelism
  - focus on function
  - hide operational view

Objectives

- efficient parallel target programs
- short development time
- appropriate for application programmer without experience of parallelism
- software quality
  - semantic definition
  - cost model

Own and Related Approaches

- skeleton idea (Cole)
- skeleton/ML compiler (Bratvold)
- C macros
- C++ skeletons (Kuchen)
- SAT (Gorlatch)
- HDC compiler
- PSML comp. (Michaelson)
- meta-progr.

1990 1995 2000 2005

New Approach (since 2002): Functional Meta-Programming (1)

Def. (meta-programming):

- analysis
- transformation
- generation

of object-programs by meta-programs

New Approach (since 2002): Functional Meta-Programming (2)

motivation for meta-programming:

- keep skeleton approach, but

  (1) use existing compiler technology

  (2) avoid administrative overhead

New Approach (since 2002): Functional Meta-Programming (3)

- CMPP 2002
  - domain-specific object-language
  - cost model
  - meta-language: Haskell

New Approach (since 2002): Functional Meta-Programming (3)

- CMPP 2002
  - domain-specific object-language
  - cost model
  - meta-language: Haskell
- CMPP 2004
  - meta-language: MetaOCaml

  \+ simple code-generation

Programming Layers

(1) domain-specific language for parallelism

- cost calculator
- SPMD code generator
- sequential/parallel task structure

Programming Layers

(1) domain-specific language for parallelism

(2) skeletons expressed in this language

- written by a parallelism expert
- handle the communications
- can exploit the parallel machine

Programming Layers

(1) domain-specific language for parallelism

(2) skeletons expressed in this language

(3) application programming

- based on skeletons
- no knowledge in parallelism required

Design Alternatives

for the parallel language

- small implementation effort

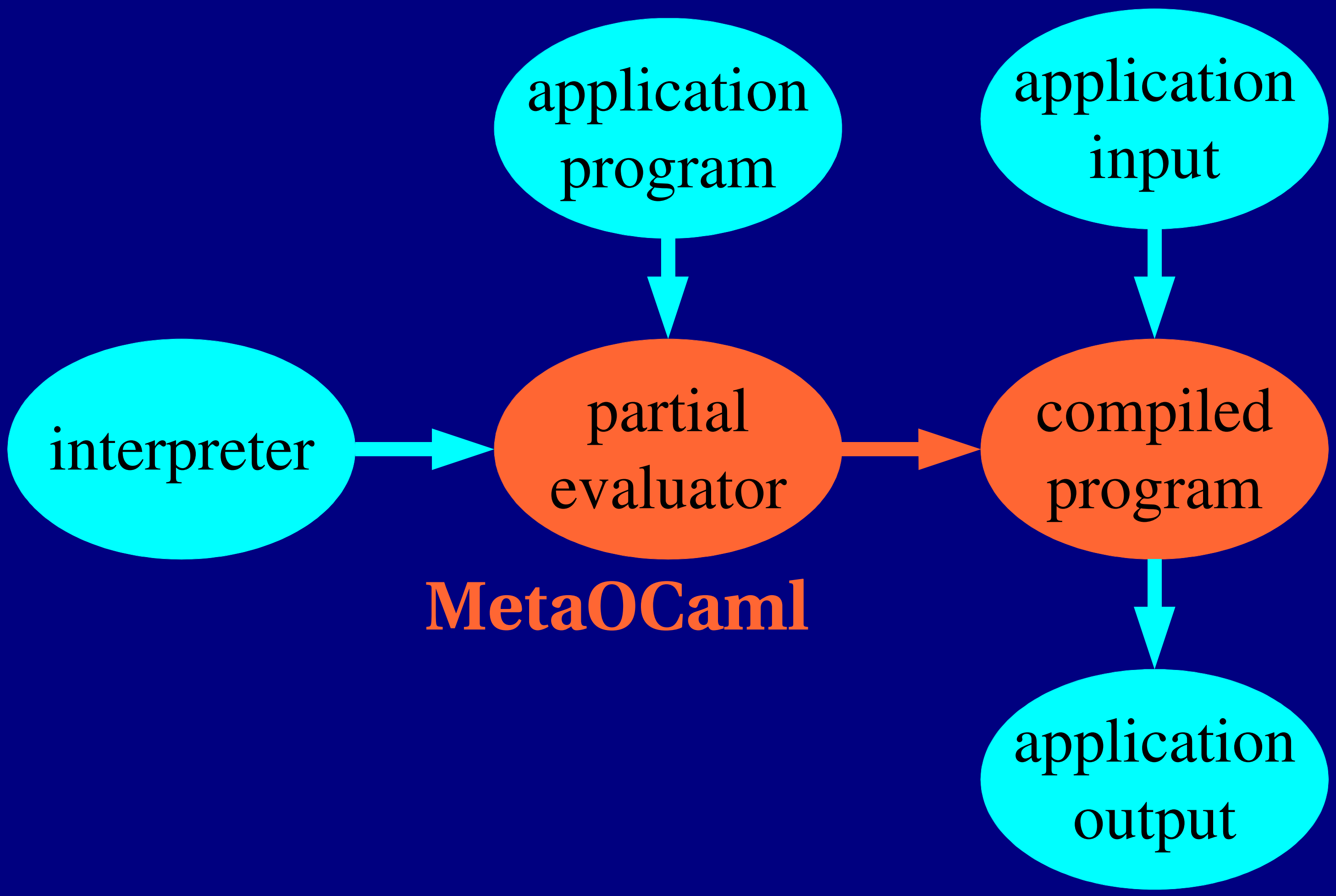
  → interpretation

- good efficiency

  → compilation

?

Interpretation
+ Partial Evaluation = Compilation



MetaOCaml

- developed by Walid Taha (Rice Univ.)
- based on Objective Caml
- run-time program specialization
- meta-programming extensions

Meta-Programming Extensions

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

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

Meta-Programming Extensions

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

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 (`.!`): executes object program

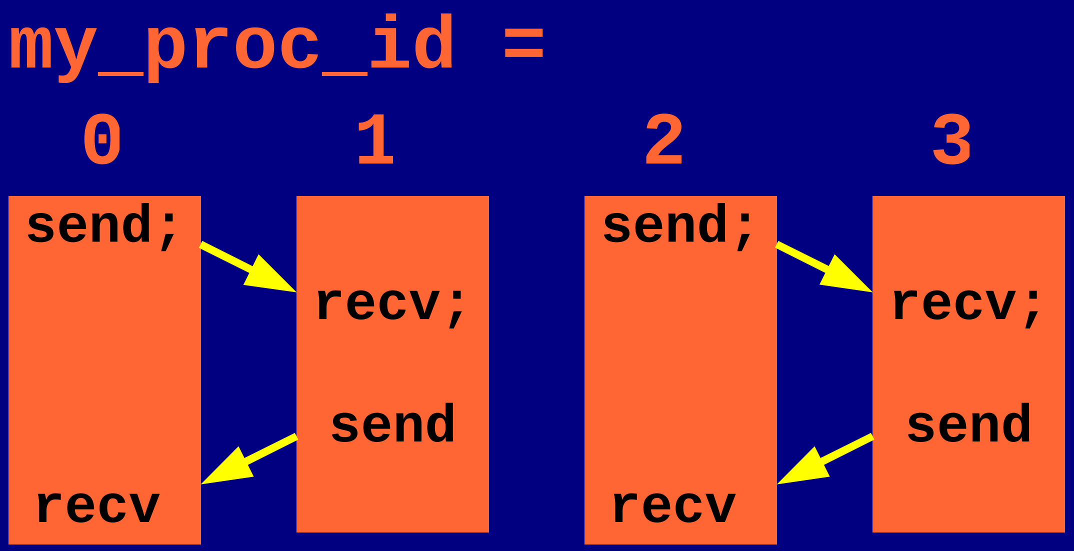
```
# let c = .!b ;;
val c : int = 17
```

Use of Meta-Programming

<u>one</u> meta-program → <u>many</u> object-programs: one for each process

simple example

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



Meta-Programming Actions

- combination of atomic parts of the specification

Meta-Programming Actions

- combination of atomic parts of the specification
- removal of interpretation overhead
  - analysis of specification

Meta-Programming Actions

- combination of atomic parts of the specification
- removal of interpretation overhead
  - analysis of specification
  - case distinctions and addressing calculations for
    - process identifier
    - block of distributed data

Meta-Programming Actions

- combination of atomic parts of the specification
- removal of interpretation overhead
  - analysis of specification
  - case distinctions and addressing calculations for
    - process identifier
    - block of distributed data
- domain-specific optimization

Specification Language

(1) atomic computation: **`Atom f`**

- **`f`** is a function in the host language

Specification Language

(1) atomic computation: **`Atom f`**

(2) sequential composition: **`Seq(n,f)`**

- **`n`**: the number of parts in the sequence
- **`f`**: mapping from index **`i`** to part **`f(i)`**

Specification Language

(1) atomic computation: **`Atom f`**

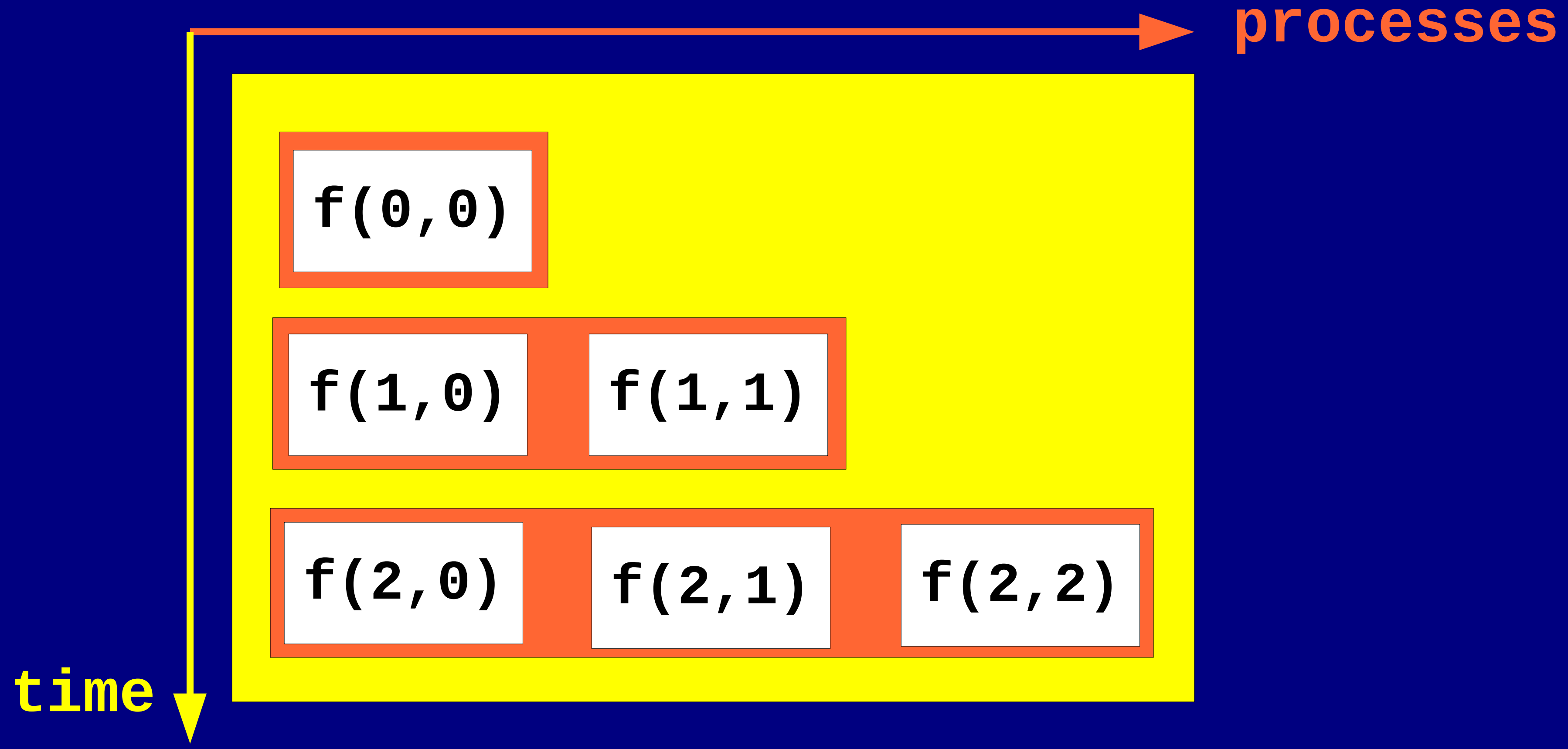
(2) sequential composition: **`Seq(n,f)`**

(3) parallel composition: **`Par(n,f)`**

- **`n`**: the number of parallel parts
- **`f`**: mapping from index **`i`** to part **`f(i)`**

Specification Example

```
let single s p = Atom (f (s,p)) in
let step s = Par (s+1, fun _ p -> single s p)
in Seq (3,fun s -> step s)
```

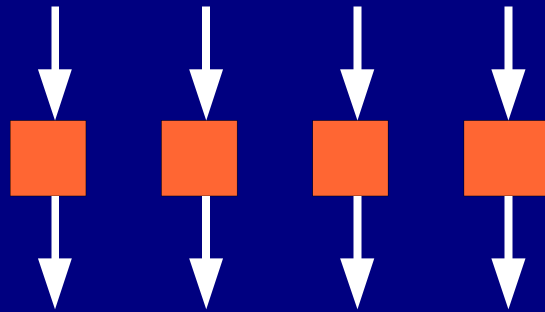


Cost Model

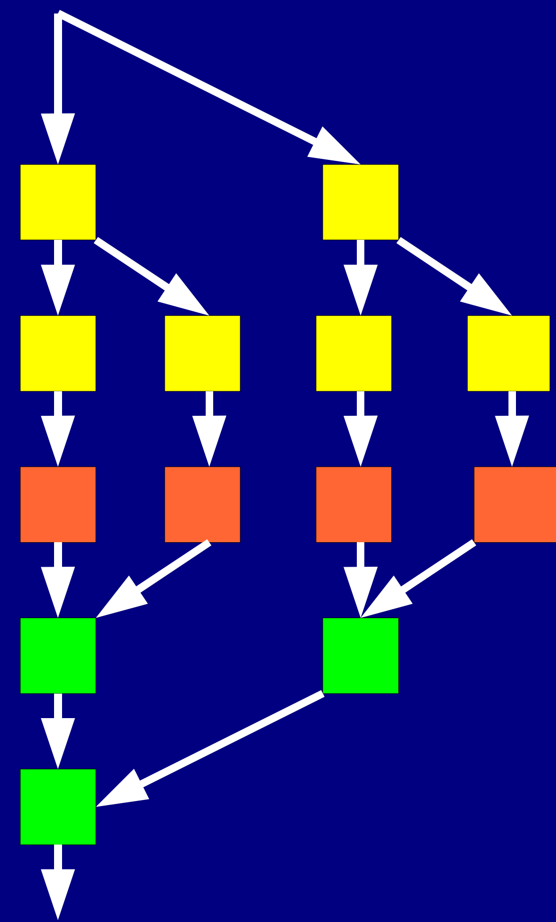
	w: work	d: depth	u: used PEs
`Atom` f	1	1	1
`Seq(`n, f`)`	$\mathrm{sum}_{0 \le i < n}\ \mathrm{w}(f\ i)$	$\mathrm{sum}_{0 \le i < n}\ \mathrm{d}(f\ i)$	$\max_{0 \le i < n}\ \mathrm{u}(f\ i)$
`Par(`n, f`)`	$\mathrm{sum}_{0 \le i < n}\ \mathrm{w}(f\ i)$	$\max_{0 \le i < n}\ \mathrm{d}(f\ i)$	$\mathrm{sum}_{0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

Skeletons

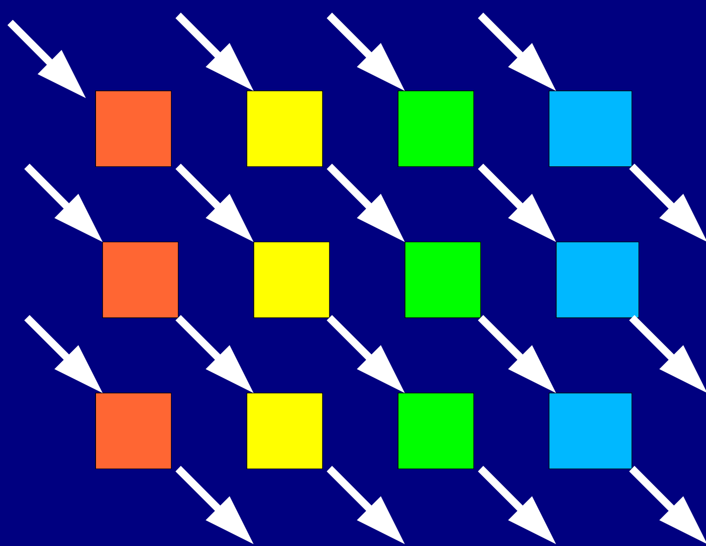
map



divide & conquer



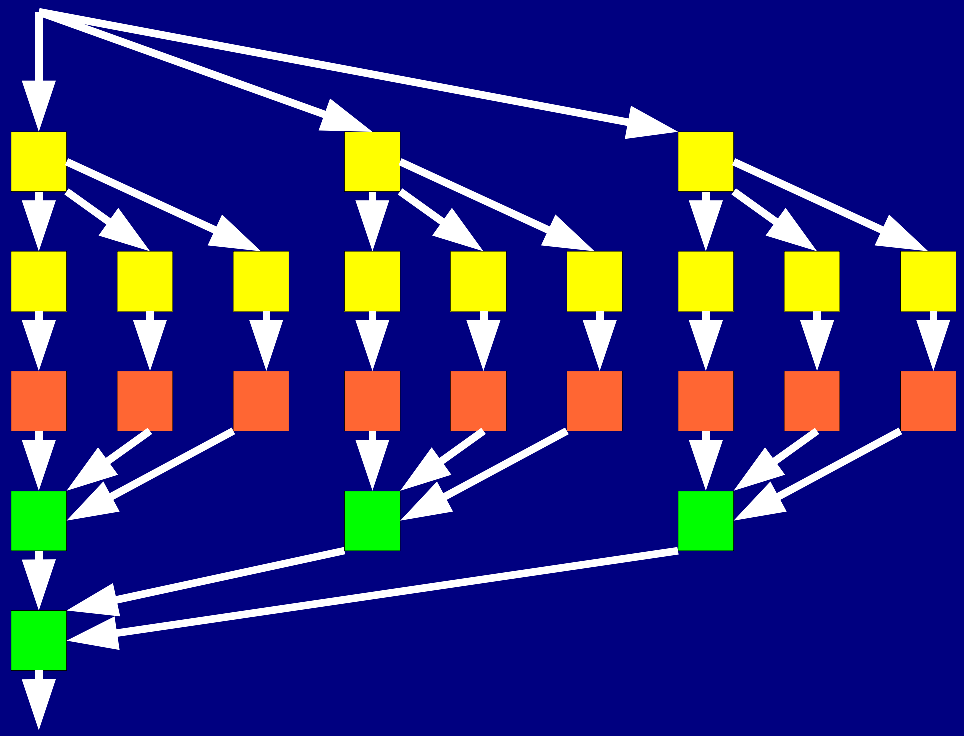
pipeline/systolic



Divide & Conquer (1)

example
- recursion depth = 2
- division degree = 3

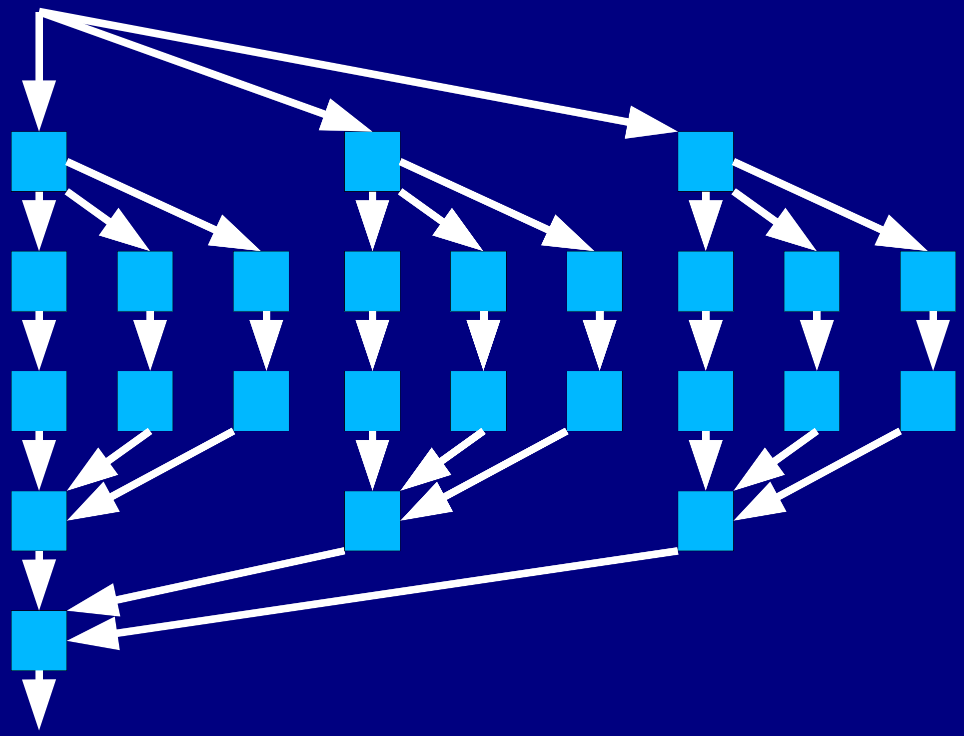
```
Par (3, ...
     Seq (_,...
          Par (3, ...
               Seq (_, ...)
               )))
```



Divide & 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+1) = Par (degree, ...Seq(_,...dc (depth)...)...)
```

Divide & Conquer (3)

- structural parameters (influence parallelism)
  - `depth`
  - `degree`
- customizing functions
  - `basic`
  - `divide`
  - `combine`

Divide & Conquer (4)

meta-program

```
let rec dc degree basic divide combine depth =
  if depth=0
  then Atom (fun x -> (.< let y = .~x
                          in basic y
                       >.))
  else
  Par (degree, subtask)
```

```
... send/recv ... divide ...

dc degree basic divide
   combine (depth-1)

... send/recv ... combine ...
```

Divide & Conquer (4a)

```
let rec dc degree basic divide combine depth =
  if depth=0
  then Atom (fun x -> (.< let y = .~x
                         in basic y
                      >.))
  else
  Par (degree, (* subtask *)
       fun partners mypart
           -> if mypart=0
              then master partners
              else worker mypart partners.(0))
```

Divide & Conquer (4b)

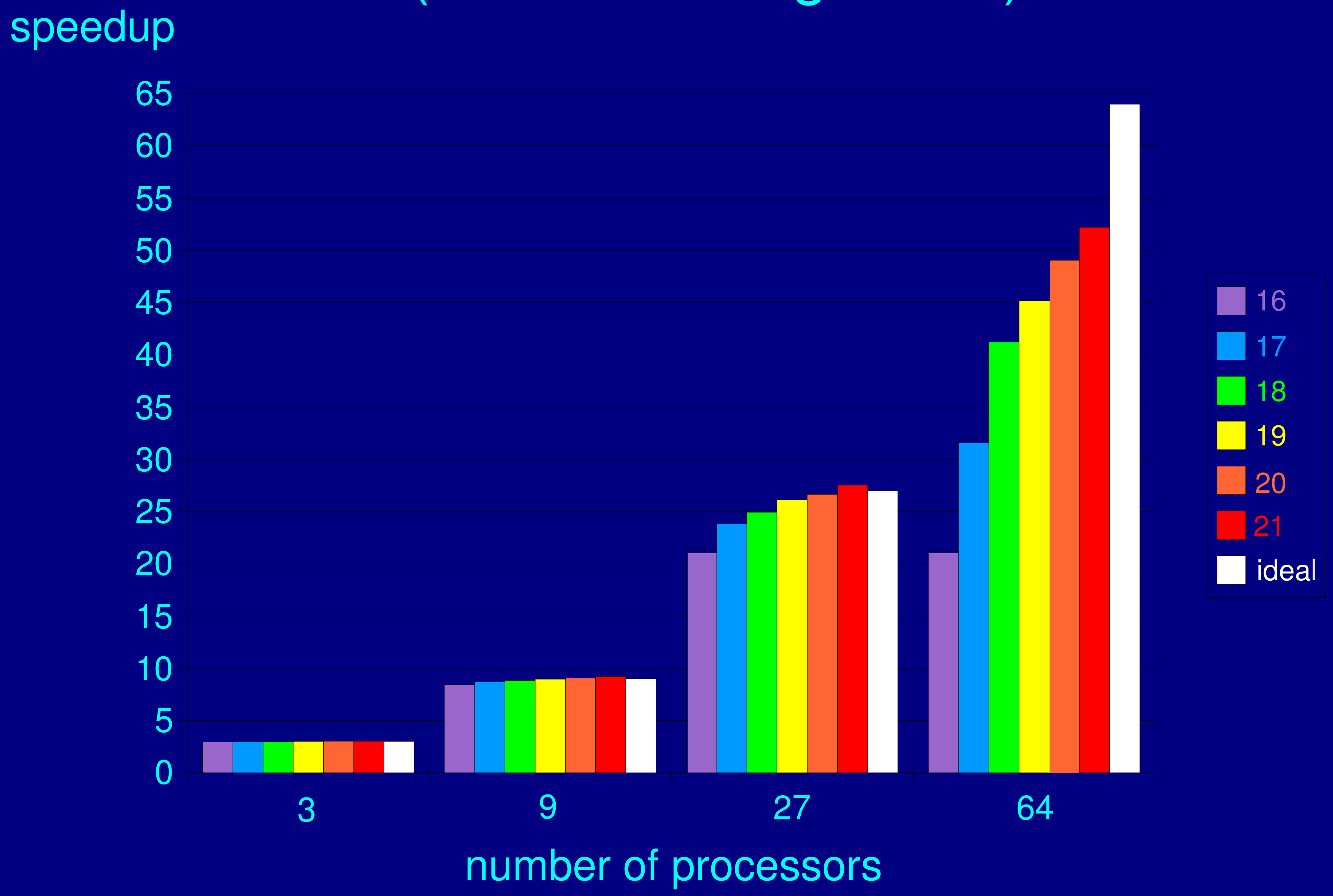
```
master partners = cseq
 [ Atom (fun x ->
          .< let y = .~x in
             for i=1 to degree-1 do
                send y (partners.(i)) depth
             done;
             divide 0 y
          >.);
   dc degree basic divide combine (depth-1);
   Atom (fun x ->
          .< let y = .~x in
             let tmpdata = Array.make degree [||] in
              tmpdata.(0) <- y;
             for i=1 to degree-1 do
                tmpdata.(i) <- receive (partners.(i))
                                      depth
             done;
             combine tmpdata
          >.)
 ]
```

Divide & Conquer (4c)

```
worker mypart partner = cseq
 [ Atom (fun x ->
          .< begin
              .~x;
              let y = receive partner depth in
              divide mypart y
             end
          >.);
   dc degree basic divide combine (depth-1);
   Atom (fun x ->
          .< let y = .~x in
             send y partner depth;
             y
          >.)
 ]
```

Experimental Results

D&C (Karatsuba Algorithm)



Experimental Results

- abstraction penalties
  - partial evaluation: **negligible** (few ms)
  - bytecode interpretation: **serious** (factor 3-8)

Conclusions

- functional meta-programming: useful for
  - organization of collective communications
  - definitions of parallel skeletons
- partial evaluation time is negligible!
- improvements of sequential parts necessary
  - short-term:
    - linking with compiled native code
    - call to external functions in C/Fortran
  - long-term:
    - just-in-time compilation

sources soon available via:

www.fmi.uni-passau.de/~herrmann

thank you for your attention!

questions?