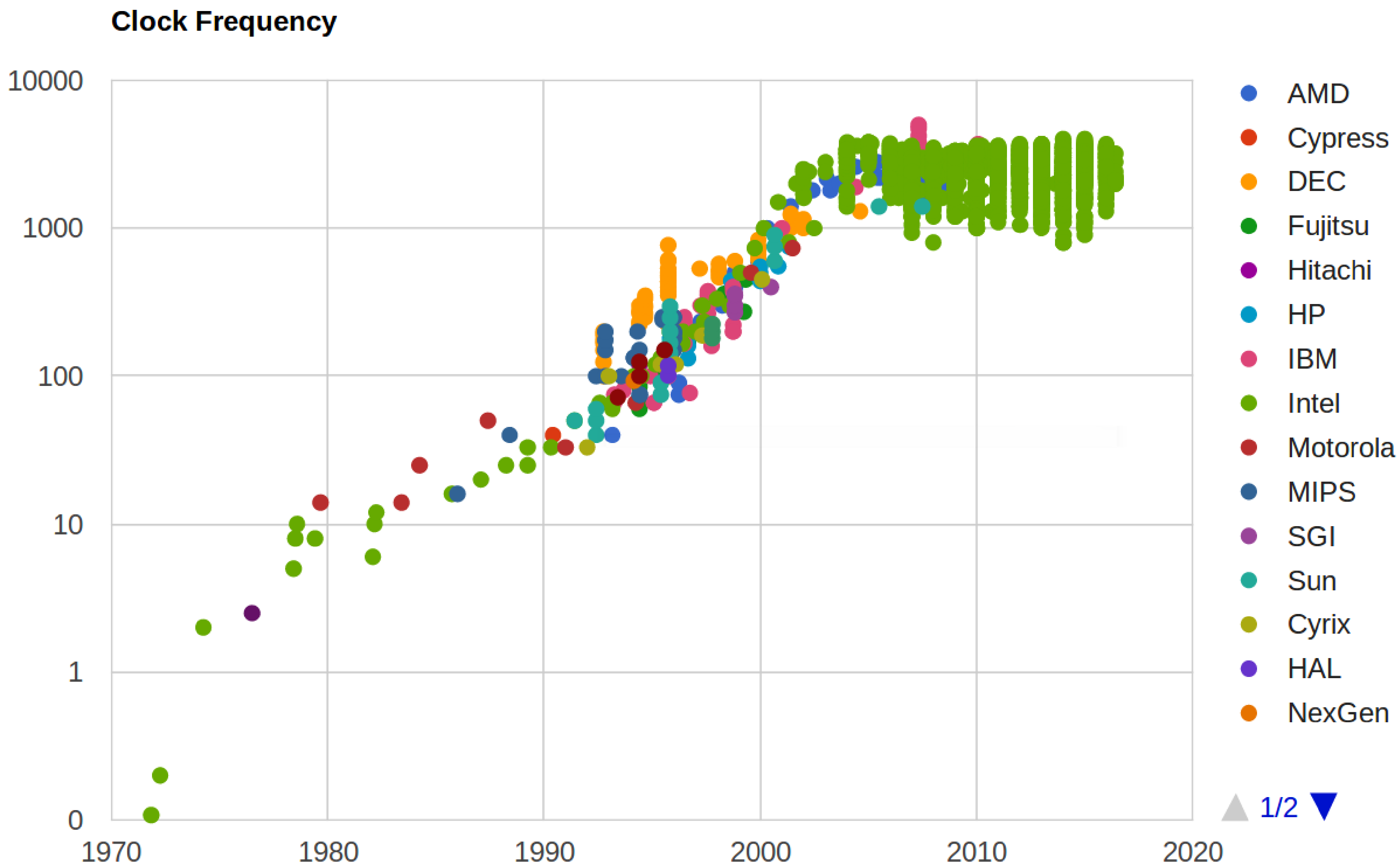
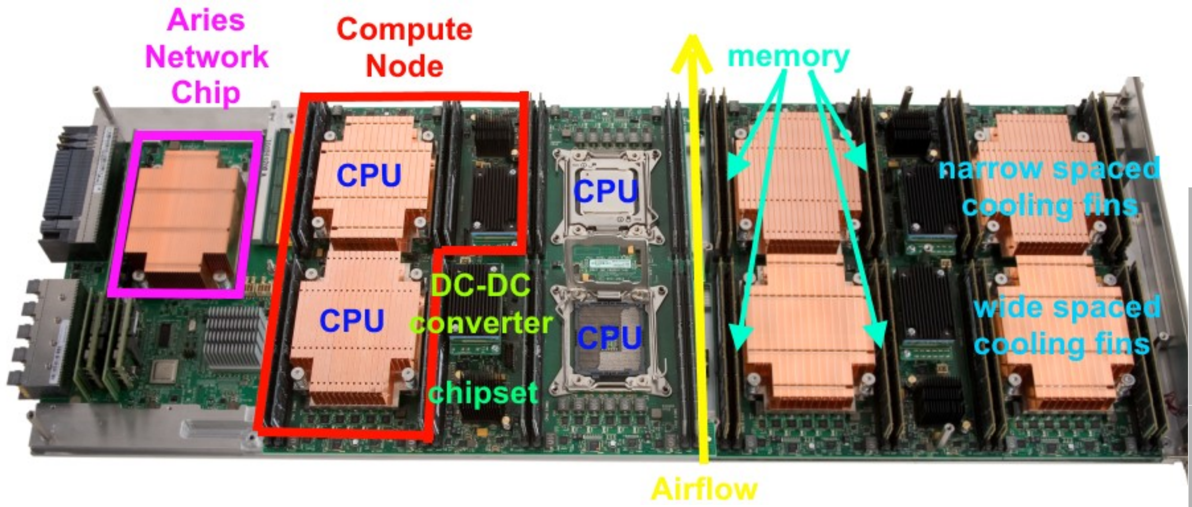
# Towards Augmenting Existing Procedural HPC Application Codes with Functional Semantics

Daniel Rubio Bonilla
HLRS – University of Stuttgart

# CPU Evolution



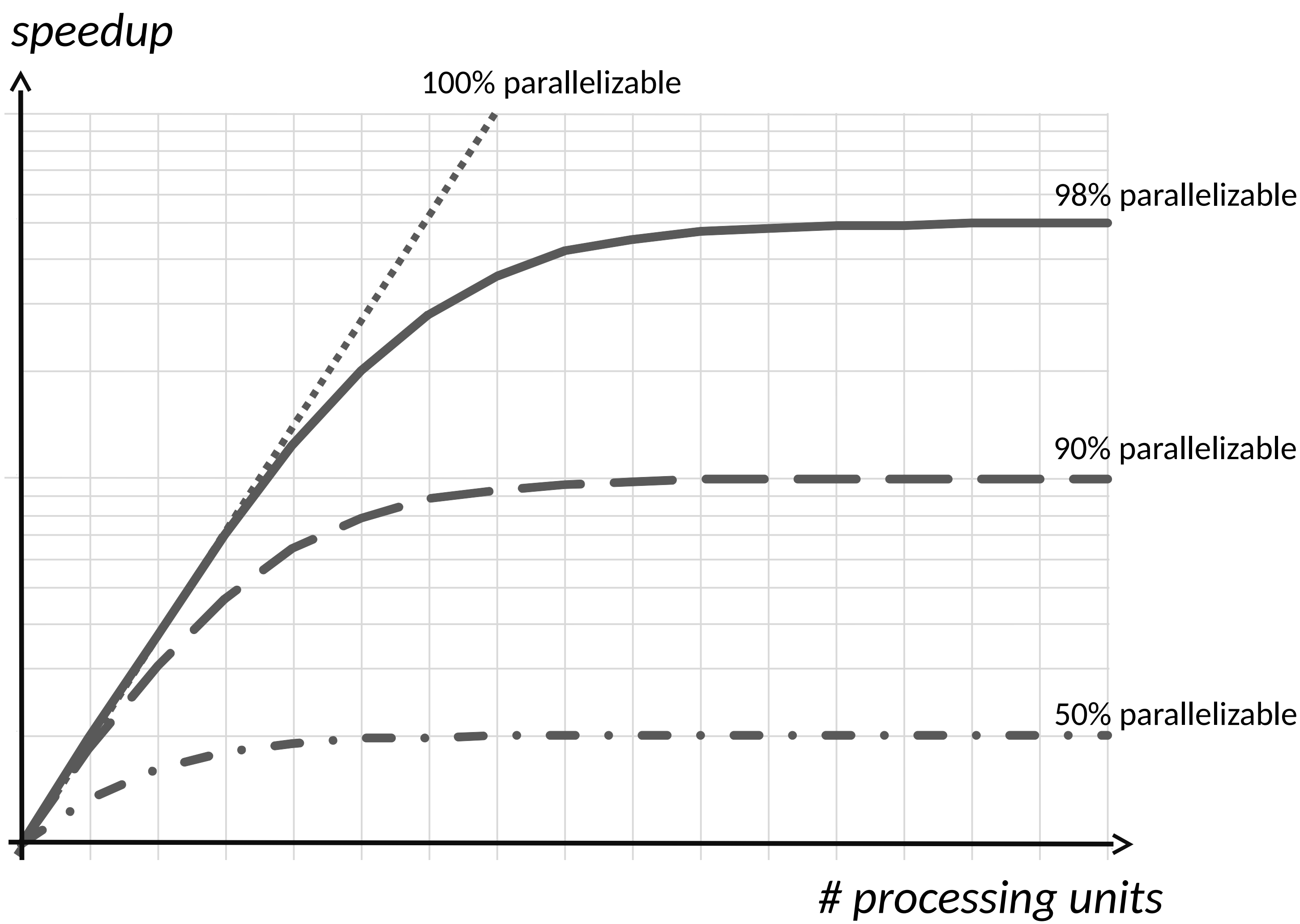
# Hazel Hen



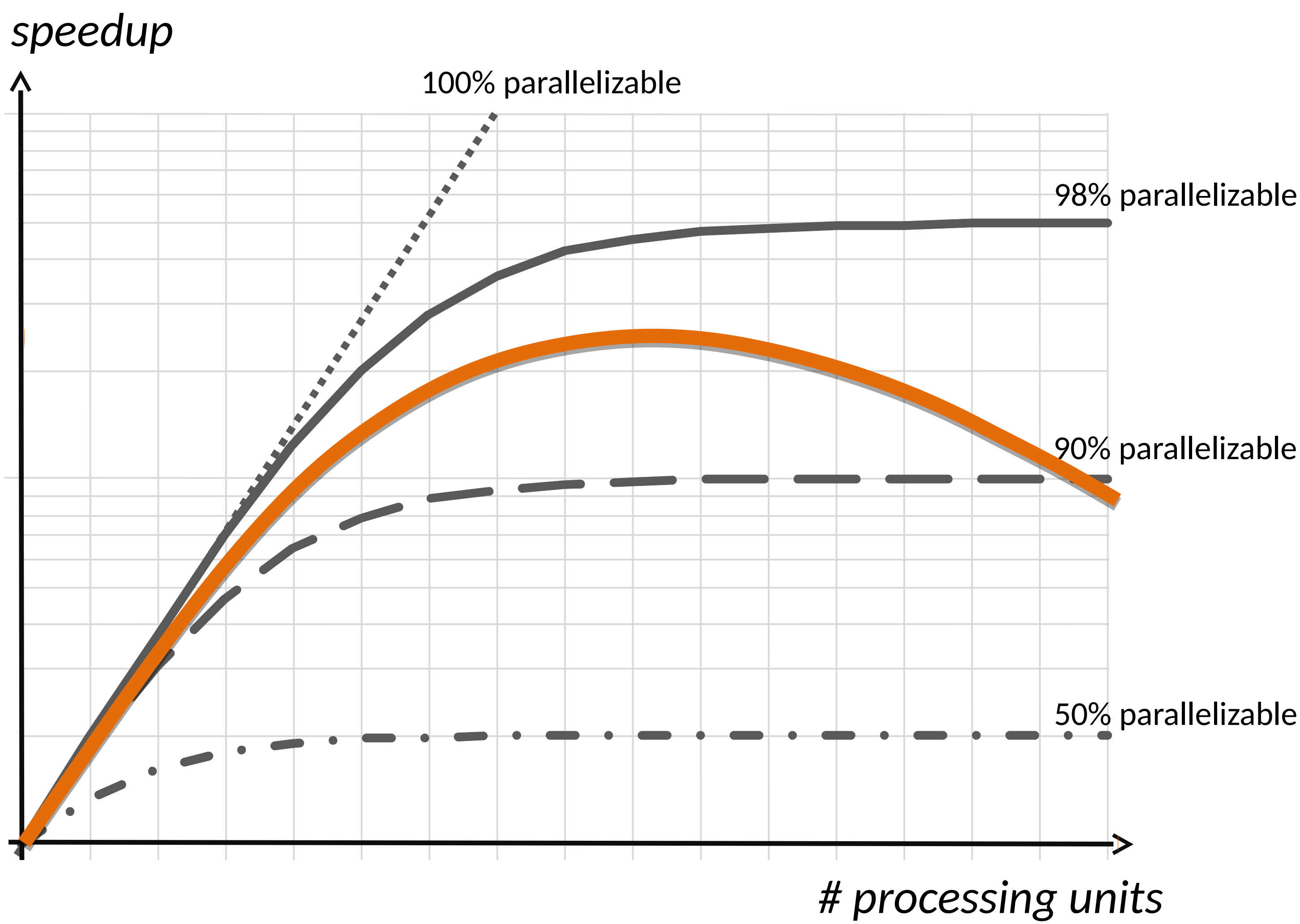

| | |
|---|---|
| CPU | E5-2680 v3<br>12 Cores<br>30MiB Cache<br>2.5 GhZ |
| Node | 2 CPUs – 24C<br>128 GB |
| Comp. Nodes | 7712 |
| Total Cores | 185,088 |
| Performance | 7420 TFlops |
| Storage | ~10 PB |
| Weight | 61.5 T |
| Power | 3200 KW |

# Amdahl Law




speedup
100% parallelizable
98% parallelizable
90% parallelizable
50% parallelizable
# processing units

# Real Amdahl Law



## In High Performance Computing…

- Performance is increased by
  - Integrating more cores (millions!?)
  - Using heterogeneous accelerators (GPU, FPGA, …)

- Issues
  - Programmability
  - Portability

# Different Programming Model

- Focused on mathematical problems
  - Engineering
  - Science

- To enable:
  - Parallelization and concurrency
  - Portability across different hardware and accelerators

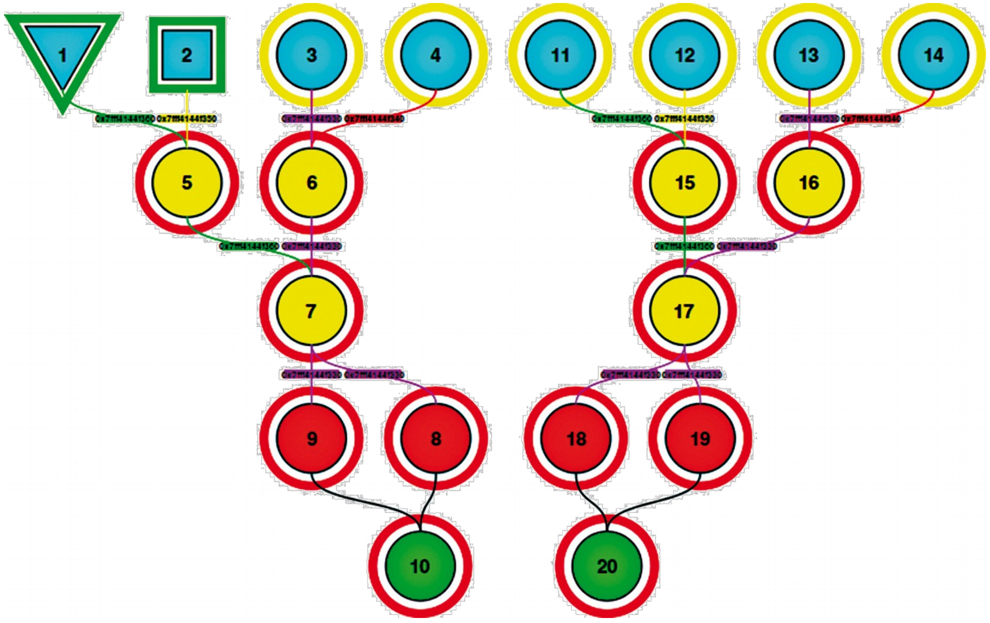
# Current Parallel Programming

- Directly introducing parallel structures
  - Threads (pthreads)
  - Message Passing (MPI)
- Offering semantically enhancements
  - Common global address space (UPC, F--)
- Introducing structural information
  - OpenMP
  - OmpSs family

using **structural information** to obtain **parallelism** and **concurrency**

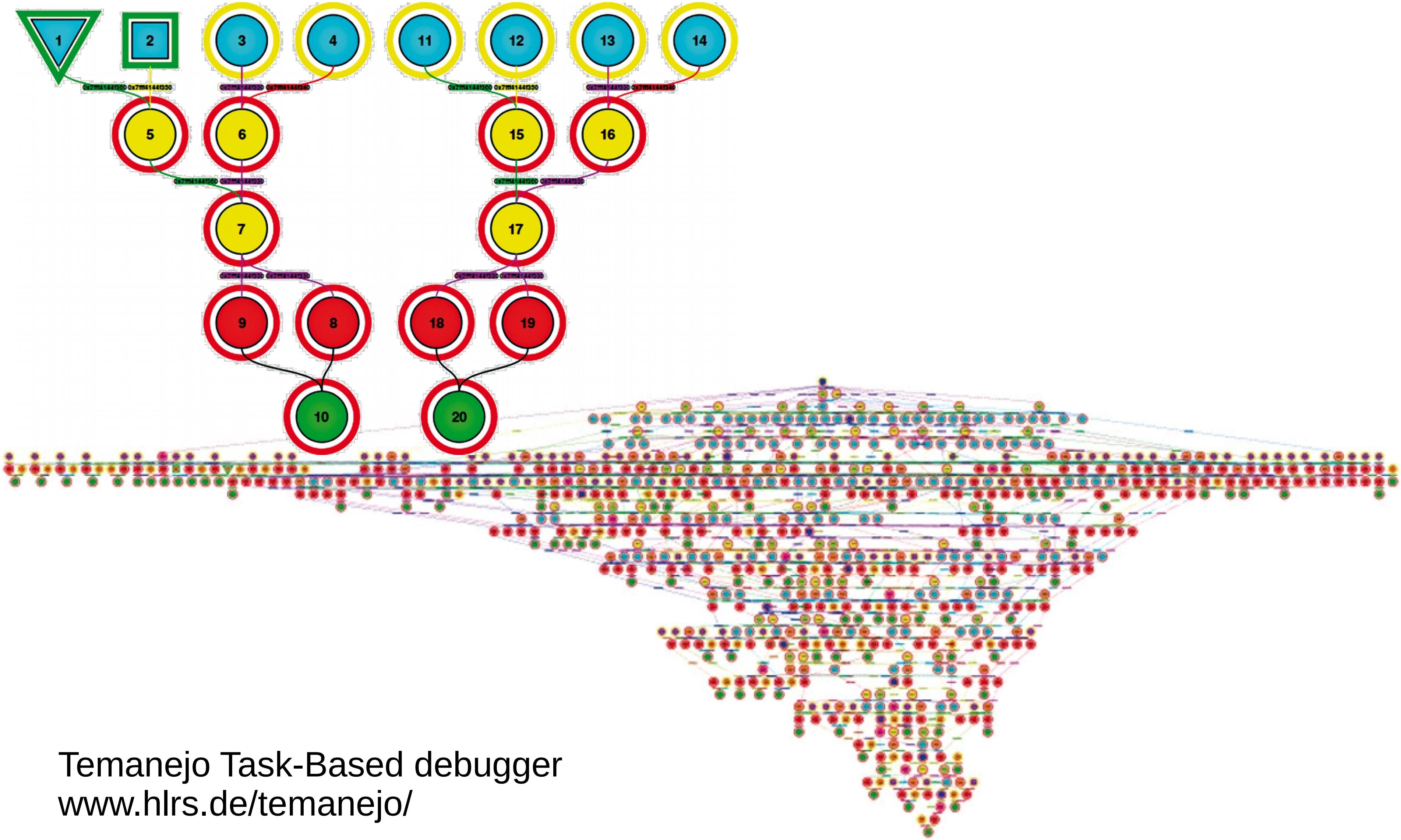
- Successful examples: OpenMP & OmpSs, but…
  - Very simple structural information
  - Lack of control of computational weight
- OpenMP
  - Only runs in cache coherent shared-memory CPUs
- OmpSs:
  - Allows for the annotation of data interface of tasks
  - Runtime scheduling

# Current Approaches (OmpSs)




Temanejo Task-Based debugger
www.hlrs.de/temanejo/

# Current Approaches (OmpSs)




Temanejo Task-Based debugger
www.hlrs.de/temanejo/

To obtain the **structural information** of the application by **annotating** the imperative code with a **functional-like directives** (mathematical / algorithmic structure)

- The main difficulty in this approach are:
  - “deriving” the structure of the application
  - matching the structure to the source code

# Higher Order Functions

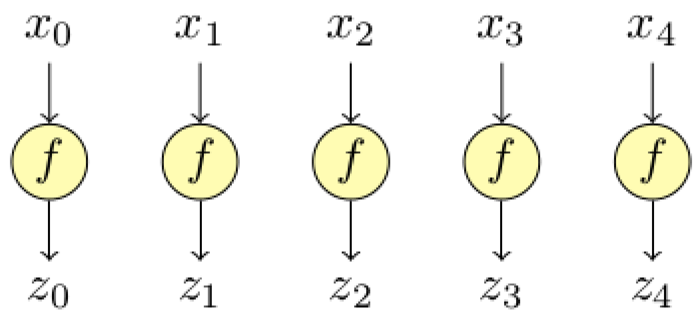
- Higher Order functions are mathematical functions
  - Takes one or more function as an argument
  - Can return a function as a result
- Clear repetitive execution structure
- These structures can be transformed to equivalent ones
  - But with different non-functional properties

# Higher Order Functions

- Apply to all:

    map :: (a -> b) -> [a] -> [b]

    map (*2) [1,2,3,4] = [2,4,6,8]

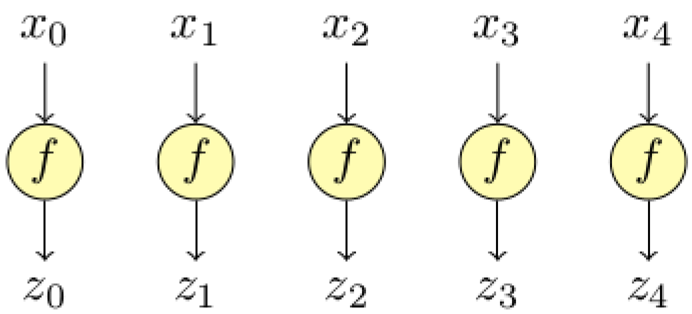


# Higher Order Functions

- Apply to all:

  *map* :: (a -> b) -> [a] -> [b]

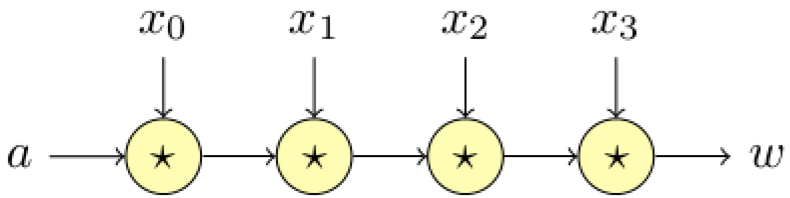
  *map* (*2) [1,2,3,4] = [2,4,6,8]



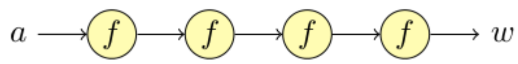
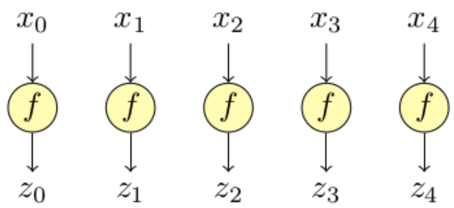
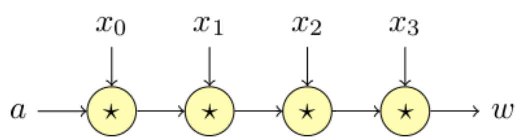
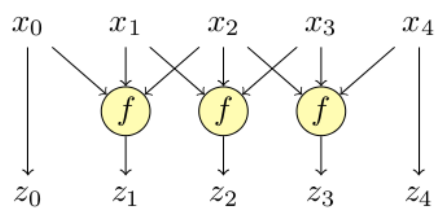

- Reduction:

  *foldl* :: (a -> b -> a) -> a -> [b] -> a

  *foldl* (+) 0 [1,2,3,4] = 10

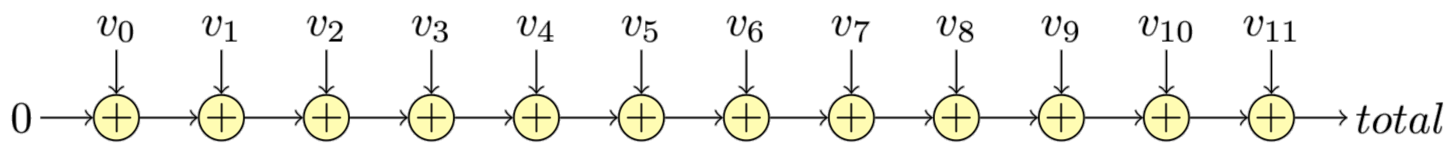


# Other Higher Order Functions

| | | |
|---|---|---|
| *itn* |  | $w = itn\ f\ a\ n$ |
| *map* |  | $zs = map\ f\ xs$ |
| *foldl* |  | $w = foldl\ (\star)\ a\ xs$ |
| *stencil1D* |  | $zs = stencil1D\ f\ w\ xs$ |

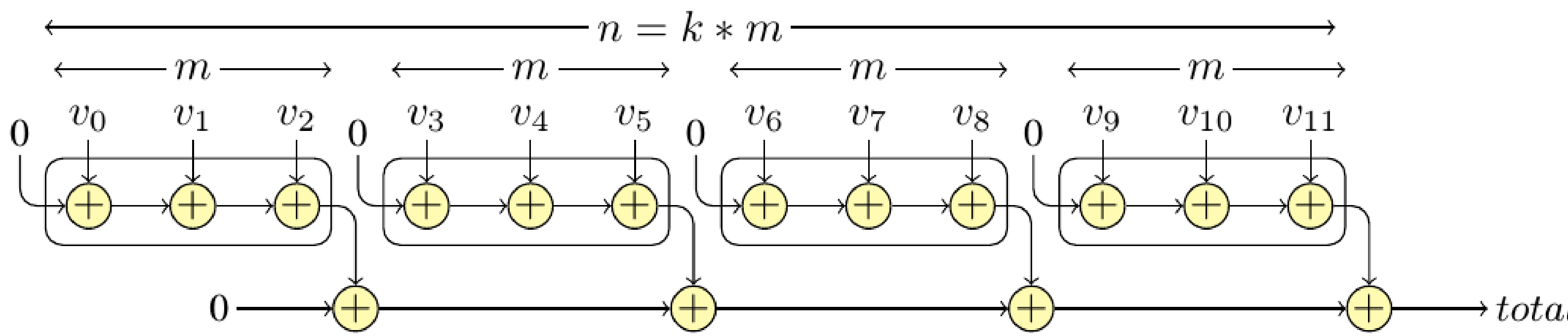
# Higher Order Functions Transformations

total = *foldl* (+) 0 vs




**One possible transformation**

**Only if the operation is associative and we know its neutral element**



# Transformations

- Changes in the mathematical formulation
  - Or the algorithm execution

- Produce equivalent code
  - Change computing load
  - Change memory distribution
  - Modify communication

- Allow adaptation to different architectures

- While maintaining correctness!

# Hierarchical Structures

- Functional annotations allow the construction of multiple structural levels:
  - Emerging complexity of the structural information

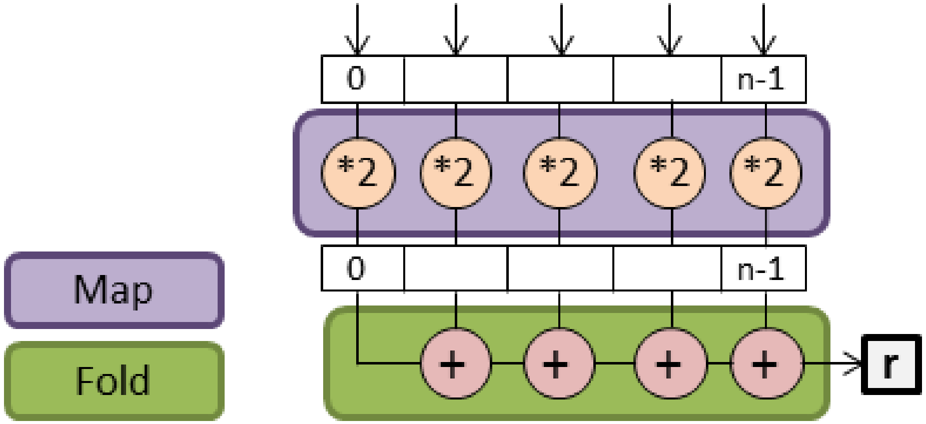
- We distinguish between:
  - Output of one Higher Order Function is input of another
    - This can be achieved by analyzing the data dependencies between the functions
  - The operator of one (Higher Order) Function is composed of other functions

# Flat Structure

Graph of a Complex Structure of two same level Higher Order Functions (HOFs)

- The output of one HOF is the input for another HOF

foldl (+) 0 (map (*2) [0..n-1])




foldl :: (a -> b -> a) -> a -> [b] -> a

map :: (a -> b) -> [a] -> [b]

# Flat Structure

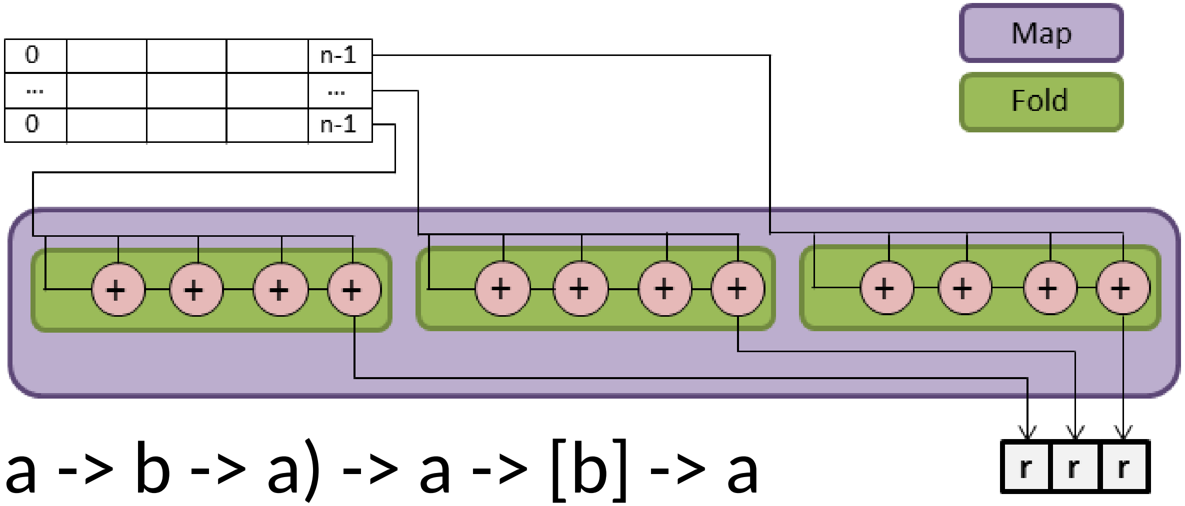
```
int va[NELEM];
int r; int i;

for(i = 0; i < NELEM; i++) va[i] = i;

#pragma polca map DOUBLE va va
for(i = 0; i < NELEM; i++) {
#pragma def DOUBLE
#pragma inout va[i]
  va[i] *= 2;
}
#pragma polca foldl PLUS 0 va r
{
  r = 0;
  for(i = 0; i < NELEM; i++) {
#pragma polca def (PLUS)
#pragma inout r
#pragma in va[i]
  r += va[i];
}
```

# Hierarchical Structure

Graph of a Complex Hierarchical Structure of two different level Higher Order Functions (HOF)

- The operator of one HOF is another HOF

map (foldl (+) 0) [[..]..[..]]




foldl :: (a -> b -> a) -> a -> [b] -> a

map :: (a -> b) -> [a] -> [b]

# Hierarchical Structure

H L R S

```
int vas[NELEM * NSELEM]; int vr[NELEM];
int i, j;

for(i = 0; i < NELEM*NSELEM; i++) vas[i] = i;

#pragma polca map OP vas vr
for(i = 0; i < NELEM; i++) {
#pragma polca OP
#pragma polca foldl PLUS 0 va vr[i]
  {
    vr[i] = 0;
    for(j = 0; j < NSELEM; j++) {
#pragma polca def PLUS
#pragma polca input vr[i] vas[(i*NELEM) + j]
#pragma polca output vr[i]
      vr[i] += vas[(i*NELEM) + j];
    }
  }
}
```

# Binding

- A strong binding is required between annotations and the source code. Needs to be specifically identified:
  - The type of Higher Order Function
  - The arguments (data and operators)
  - The result / output

```
int addAll(int *v, size_t n) {
int total = 0;
#pragma polca foldl PLUS total v total
  {
    for(int i=0; i<n; i++) {
#pragma polca def PLUS
#pragma polca inout total
#pragma polca in i
      total += v[i];
    }
  }
  return total;
}
```

# Mathematical Properties

- A **ring** is a set R with binary operations + and *

- R is an *abelian group* under addition
  - Additive operation is associative and commutative
  - There is an additive identity element
  - There is an additive inverse

- R is a *monoid* under multiplication
  - Multiplication is associative
  - There is a multiplication identity element

- Multiplication is distributive with respect to addition

```
#pragma ring_prop (+, 0, -, *, 1) int
```

# Data / Memory Organization

- Data type
  - Size
  - Data representation
- Organization
  - Endianness
  - Array size

```
#pragma polca memAlloc (sizeof(type)) ARRAY_ELEM myArray
type* myArray = (type*) malloc(sizeof(type)*ARRAY_ELEM);

...

#pragma polca memcopy array1 N_ELEM array2
malloc((void*) array1, (void*) array2, N_ELEM * sizeof(type));
```

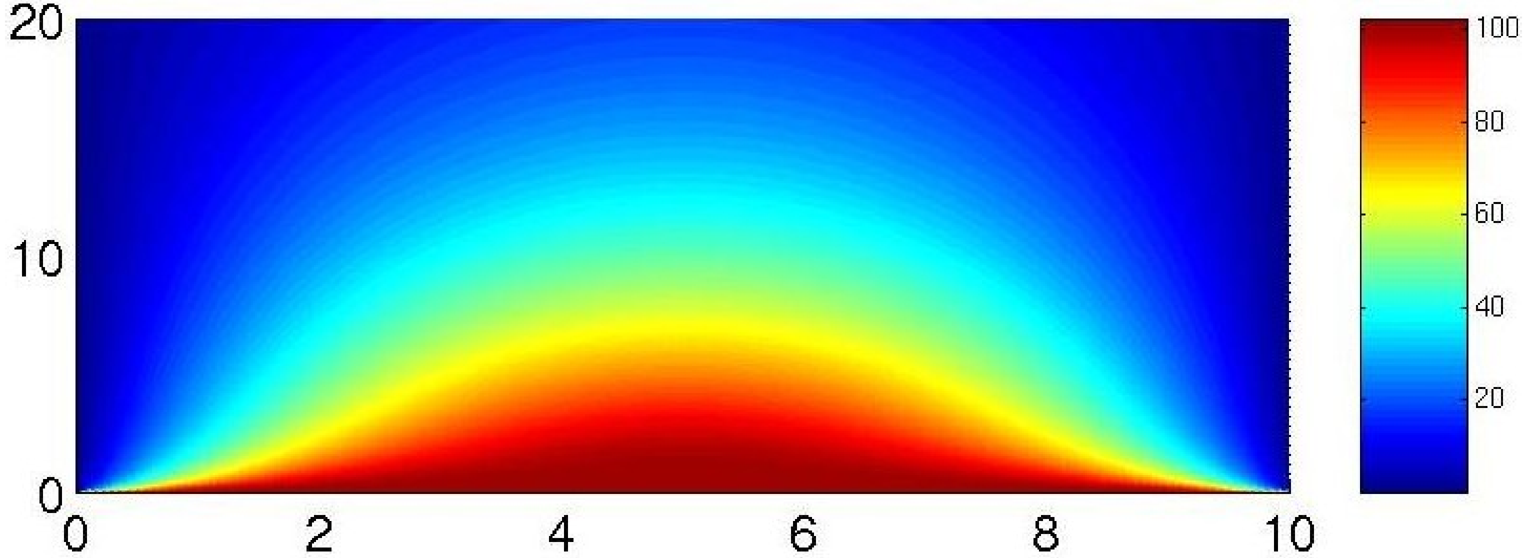
# Example - Mathematics

**Large scale tasks are mostly mathematical**

example:

$$\frac{\partial T(x,\mathrm{t})}{\partial t} - \propto \cdot \nabla^2 T(x,t) = 0$$

1-D heat dissipation function



# Mathematics - Discretization

① $$\frac{\partial T(x,\mathrm{t})}{\partial t} - \propto \cdot \nabla^2 T(x,t) = 0$$

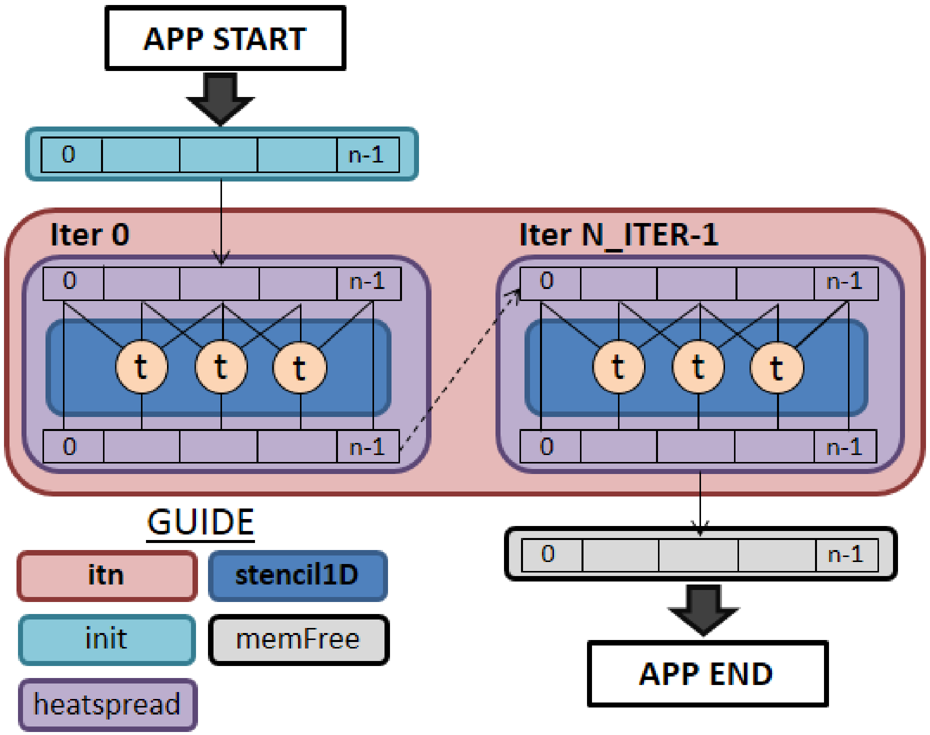
② $$\lim_{\Delta t \to 0} \frac{T(t+\Delta t, x) - T(t,x)}{\Delta t} = \propto \cdot \lim_{\Delta x \to 0} \frac{T(t, x-\Delta x) - 2T(t,x) + T(t, x+\Delta x)}{(\Delta x)^2}$$

...

③ $$z_i' = z_i + c \cdot (z_{i-1} - 2z_i + z_{i+1})$$

# Structure

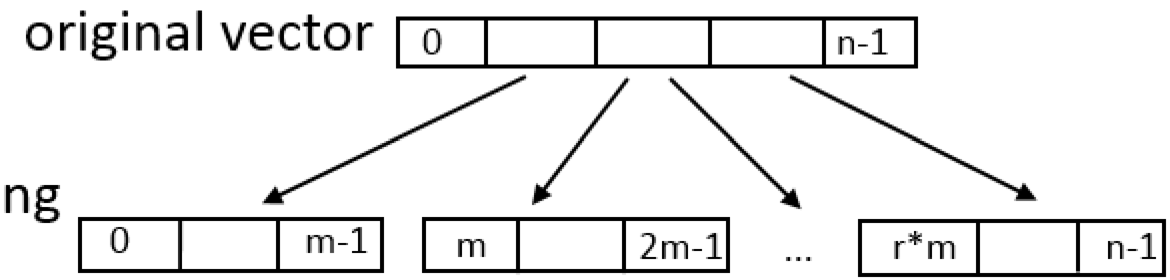
```
let heatDiffussion = itn HEATTIMESTEP hm_array N_ITER
HEATTIMESTEP v = stencil1D TKernel 1 v
  where TKernel x y z = y + K * (x - 2*y + z)
```




```
#pragma polca stencil1D \
         TKernel 1 hm hm_tmp
for(i=1; i<n_elem-1; i++){
#pragma polca def Tkernel
  hm_tmp[i] = hm[i] + K*(hm[i-1] \
          +hm[i+1]-2*hm[i]);
}
```

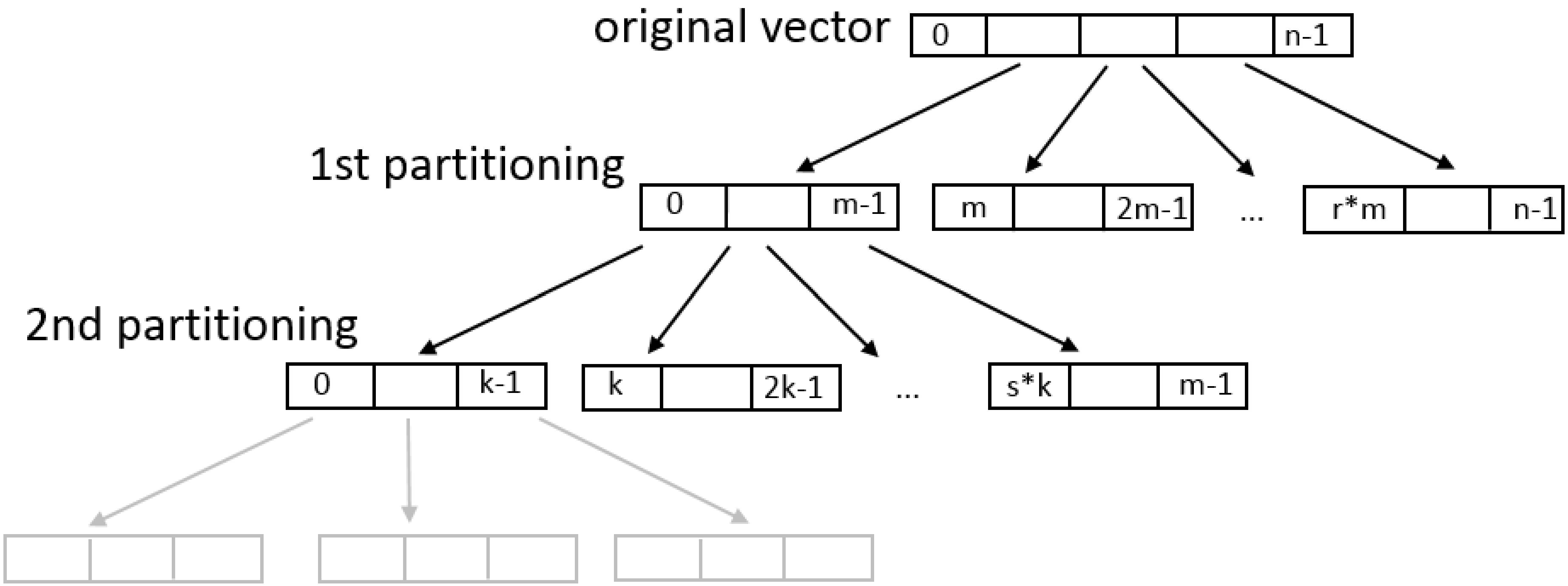
# Transformations – Partitioning 1

```
let heatDiffusion = itn HEATTIMESTEP hm_array N_ITER
PAR1 v = stencil1D TKernel 1 v
  where TKernel x y z = y + K * (x - 2*y + z)
HEATTIMESTEP vs = map PAR1 vs
```



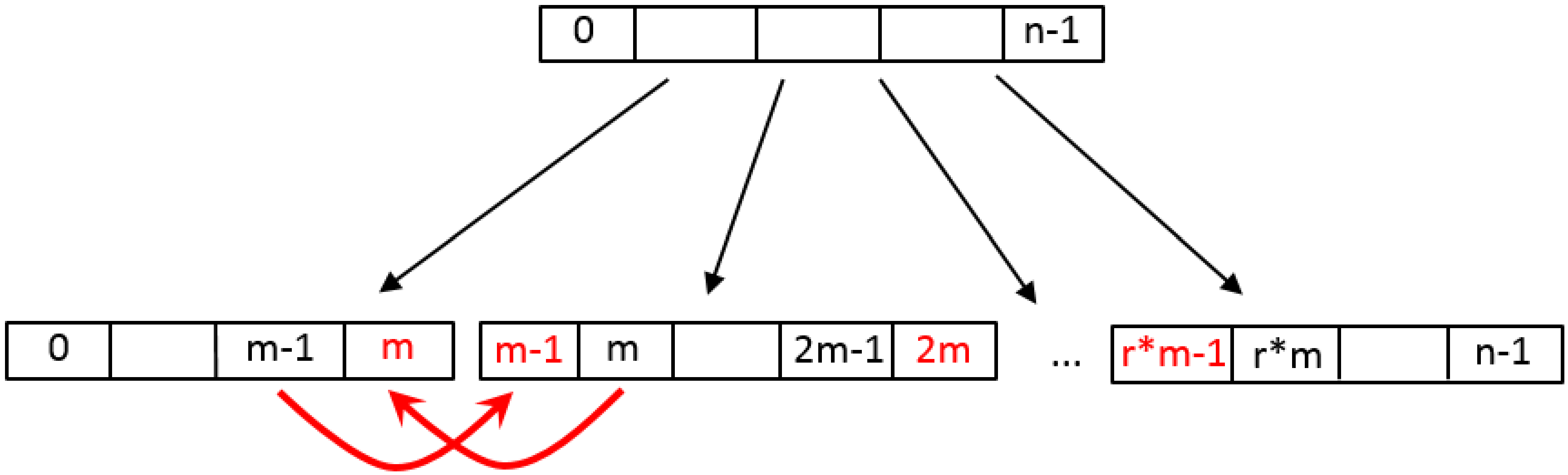
# Transformations – Partitioning 2

```
let heatDiffusion = itn HEATTIMESTEP hm_array N_ITER
PAR2 v = stencil1D TKernel 1 v
  where TKernel x y z = y + K * (x - 2*y + z)
PAR1 vs = map PAR2 vs
HEATTIMESTEP vss = map PAR1 vss
```



# Platform Specific Transformations

- OpenMP:
  - Relatively straightforward
- MPI:
  - Communication
  - Halos



# Transformed Code

```
if (rank < size - 1)
  MPI_Send(&hm[LOCAL_N_ELEM],1, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD);
if (rank > 0)
  MPI_Recv(&hm[0], 1, MPI_FLOAT, rank-1, 0, MPI_COMM_WORLD, &status);
if (rank > 0)
  MPI_Send(&hm[1], 1, MPI_FLOAT, rank-1, 1, MPI_COMM_WORLD );
if (rank < size - 1)
  MPI_Recv(&hm[LOCAL_N_ELEM+1],1,MPI_FLOAT, rank+1, 1, MPI_COMM_WORLD, \
                                                                    &status);
```

```
#pragma polca stencil1D 1 TKernel hm hm_tmp
#pragma omp parallel for
for(i=1; i<LOCAL_N_ELEM+1; i++) {
#pragma polca Tkernel
#pragma polca input hm[i-1] hm[i] hm[i+1]
#pragma polca output hm_tmp[i]
  hm_tmp[i] = hm[i] + K * (hm[i-1] + hm[i+1] - 2 * hm[i]);
}
```

# Heterogeneous Hardware

HLRS

- FPGAs
  - Functional Annotations → Clash → VHDL
  - Need a full specification
- OpenCL
  - Operations similar to C
  - Need to add communication
  - No recursion

```
    ,eta_i2     => repANF_1);

indexVec_n_12 : block
  signal n_13 : array_of_signed_16(0 to 2);
  signal n_14 : integer;
begin
  n_13 <= eta_i1;
  n_14 <= 2;
  -- pragma translate_off
  process (n_13,n_14)
  begin
    if n_14 < n_13'low or n_14 > n_13'high then
      assert false report ("Index: " & integer'image(n_14) & "
       integer'image(n_13'low) & " to " & integer'image(n_13'
      tmp_11 <= (others => 'X');
    else
    -- pragma translate_on
      tmp_11 <= n_13(n_14);
    -- pragma translate_off
    end if;
  end process;
  -- pragma translate_on
end block;

repANF_3 <= tmp_11;

satPlus_6_repANF_4 : entity satPlus_6
  port map
    (bodyVar_o => repANF_4
    ,eta_i1     => repANF_2
    ,eta_i2     => repANF_3);
```

# Thank you!

**Contact:**
rubio@hlrs.de

**Projects:**
POLCA www.polca-project.eu
Smart-Dash www.dash-project.org
CλaSH www.clash-lang.org