Transformation of Functional Programs into Data Flow Graphs implemented with PVM

Ralf Ebner

Alexander Pfaffinger

Technische Universität München

1. Functional Programming and Parallelization
2. From FASAN to a Data Flow Graph
3. Distributed Data Flow Graphs with PVM
4. Runtime Observations and Conclusions

Functional Programming and Parallelization

Mathematical function notion: $y = sin(x)$

- Functions describe mapping from input to output data
  $\Rightarrow$ no **side effects**
- Identifiers keep their assigned value
  no „procedure variable“, no pointers
  $\Rightarrow$ no **implicit global computation state**
- Functions can be evaluated whenever all input data is available
  $\Rightarrow$ no **explicit execution sequence**

Therefore:

- clear dependence of functions
- programs easier to understand
- soundness proofs
- automatic detection of potential parallelism
- free of deadlocks

FASAN

"**F**unctions **A**pplied to **S**treams
in **A**lgorithms of **N**umerical Computing "

Requirements:

- **comfortable**
- **reusable**
- **efficient**
- **portable**

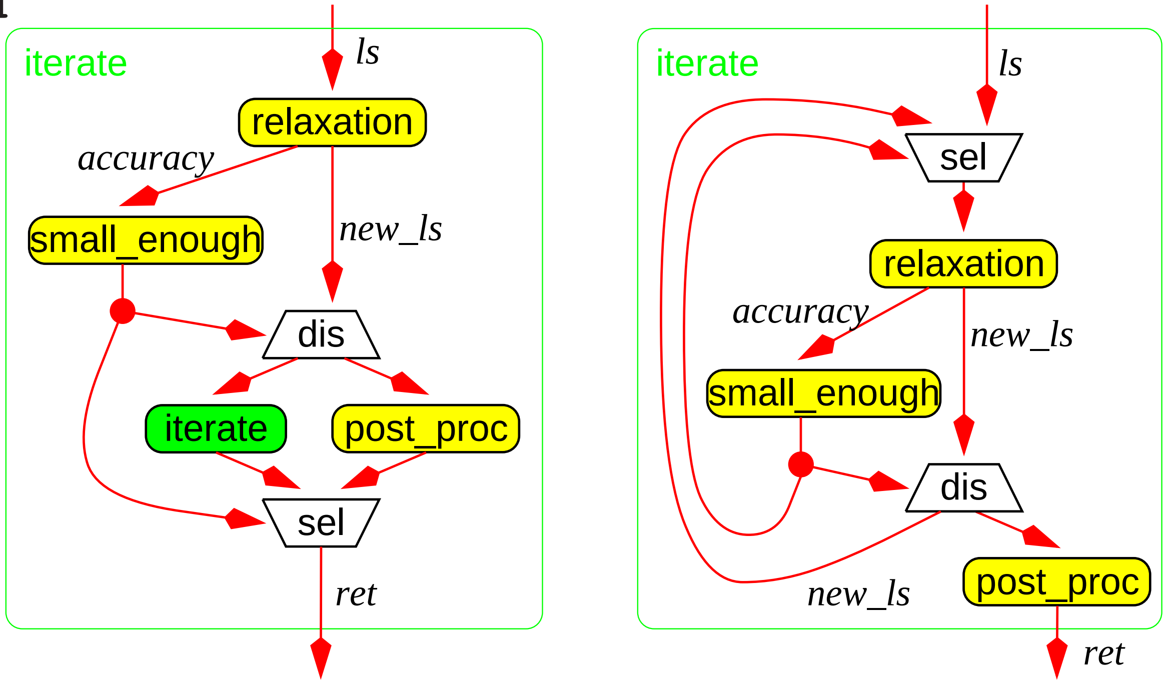
Concepts of FASAN:

- functional layer for parallel configuration of sequential modules
- linking with functions written in procedural languages (C, Fortran, ...) or libraries
- data flow computation model, streams of data
- efficient implementation of tree-like data structures
- built upon portable parallel system: PVM

Iterations

Technique: tail recursion

```
function iterate (ls): ret;
begin
  new_ls, accuracy := relaxation (ls);
  if small_enough (accuracy)
    then ret := post_proc (new_ls);
    else ret := iterate (new_ls);
  fi
end
```

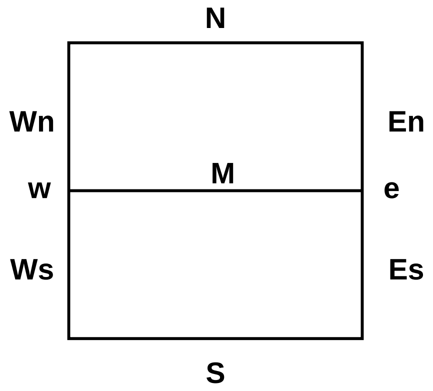


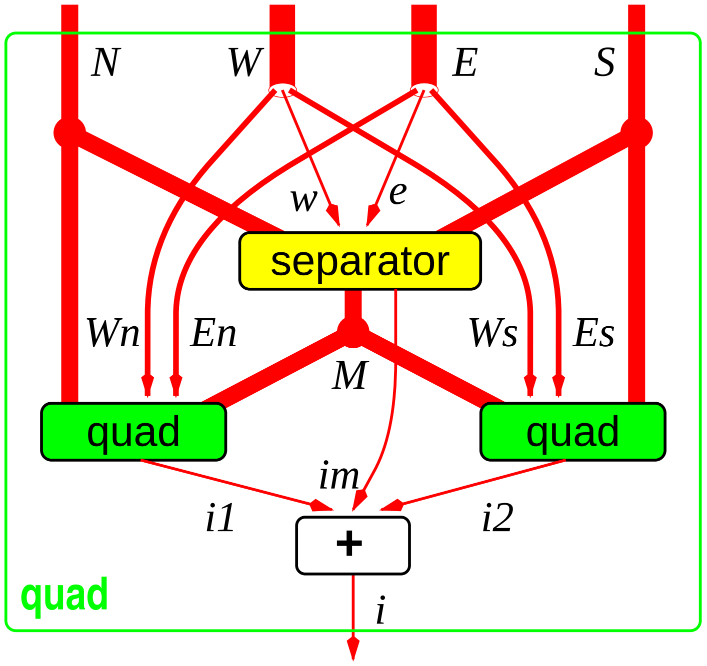
- reuse of existing data flow graph
- experience from preceeding iterations

Divide & Conquer

Technique: cascade recursion on tree structures

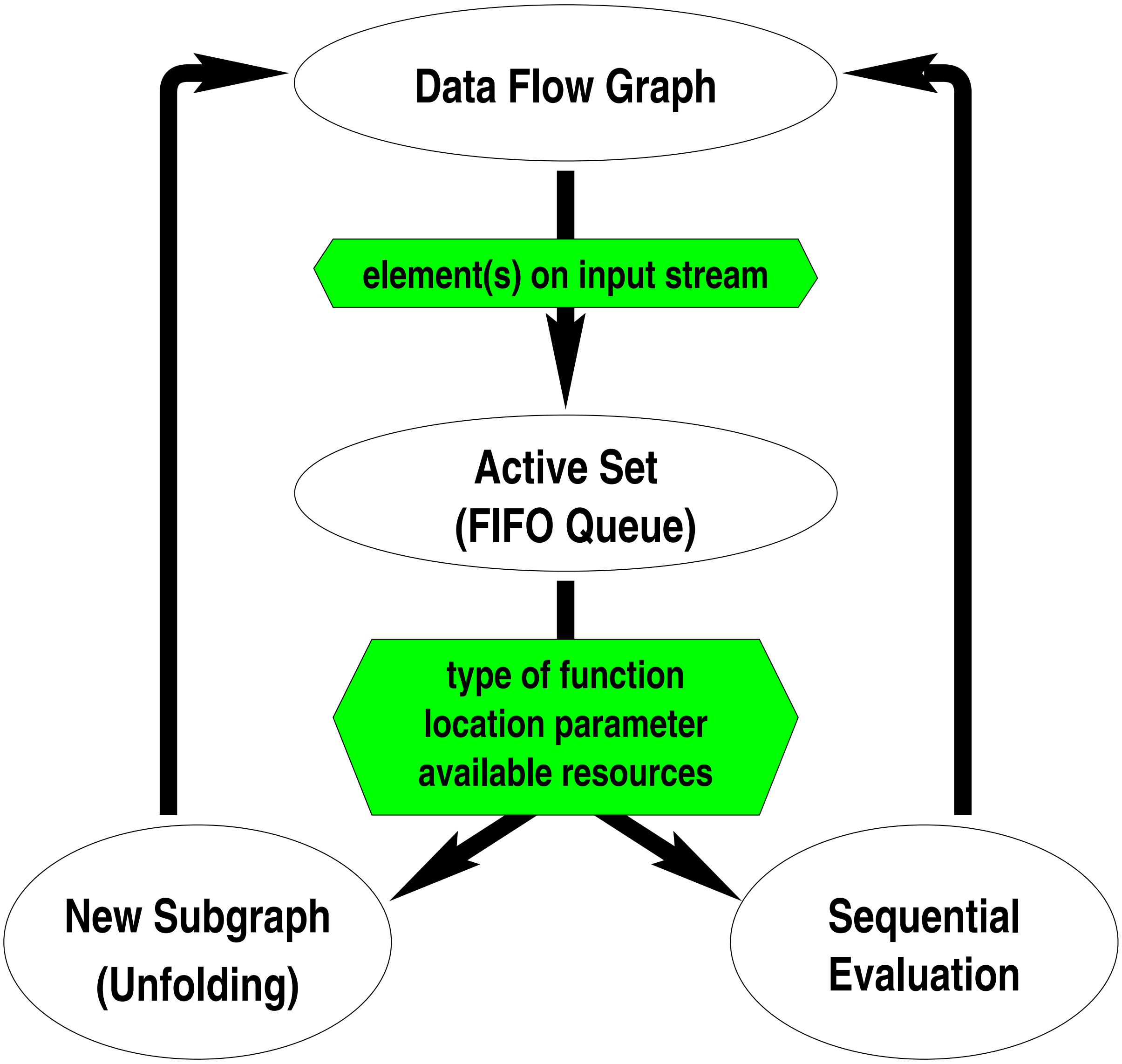
```
function quad (N, S, W, E): i;
begin
  ⟨ Wn, w, Ws ⟩ := W;
  ⟨ En, e, Es ⟩ := E;
  im,M := separator(N,S,w,e);
  i1 := quad (N,M,Wn,En);
  i2 := quad (M,S,Ws,Es);
  i := im + i1 + i2;
end
```



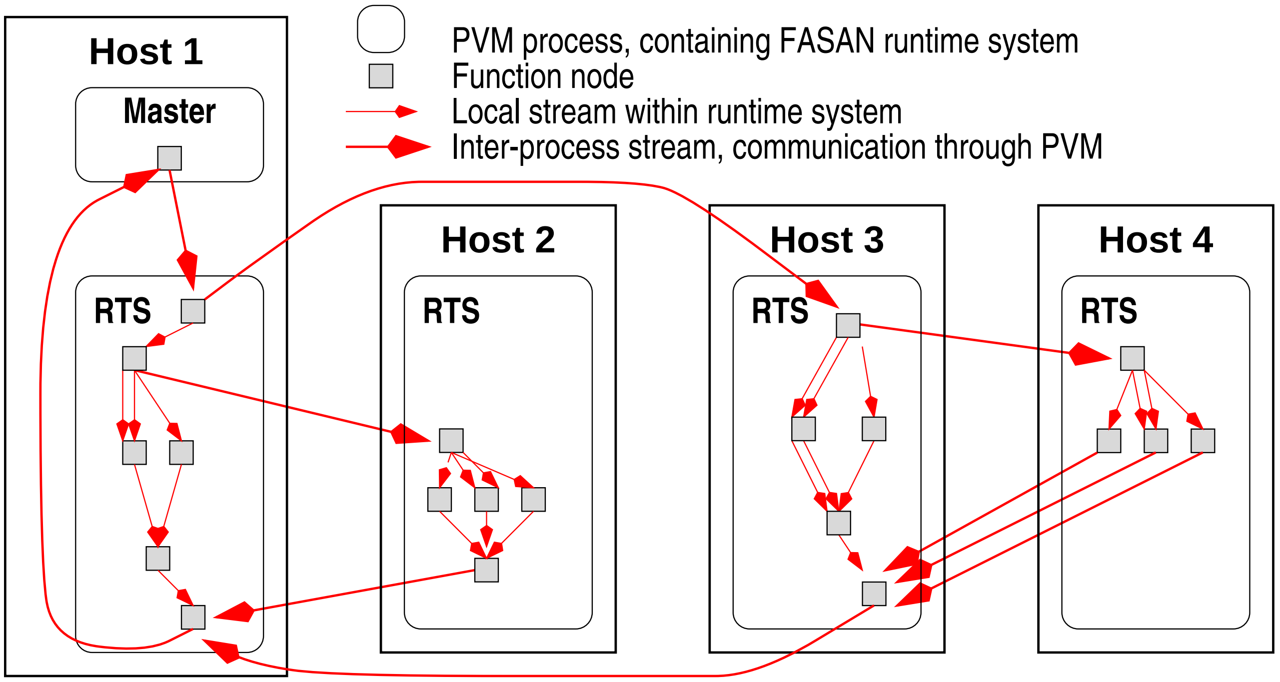


- exploitation of implicit maximal parallelism
- less communication

Life cycle of a function node

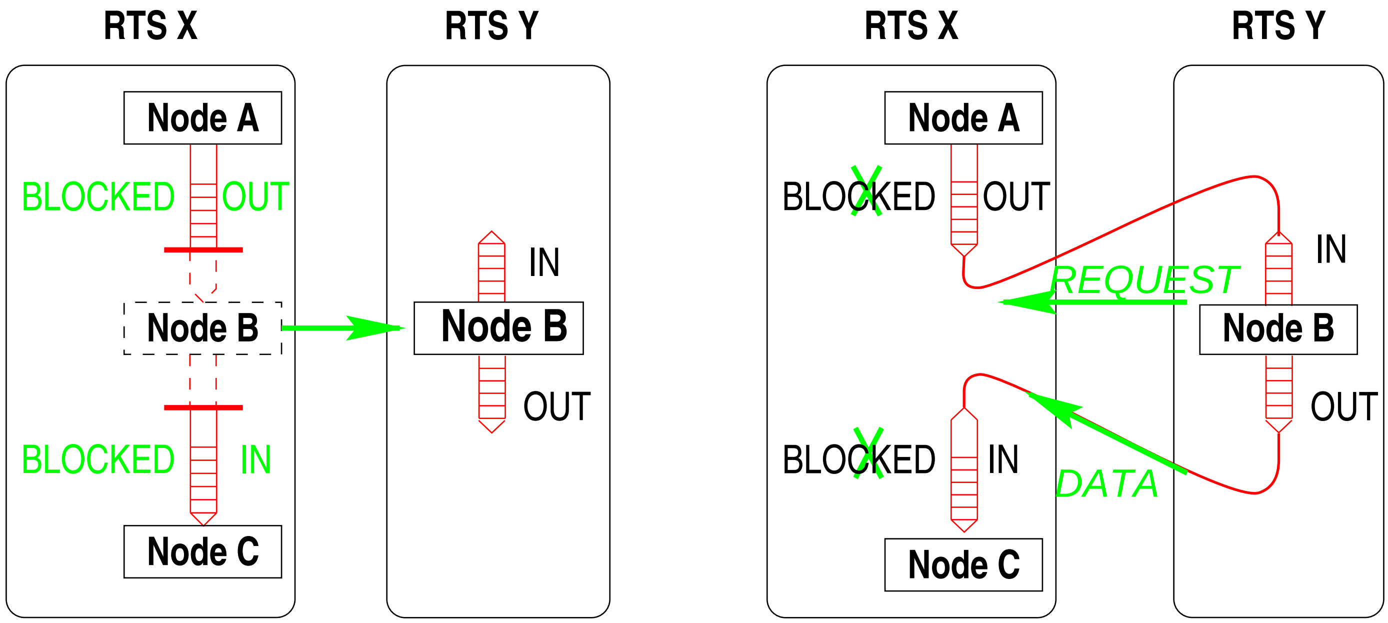


Distribution of a Data Flow Graph



- Master spawns all FASAN runtime systems as PVM processes
- Node migration between runtime systems according to user defined migration schemes

Migration of Function Nodes



Protocol on top of PVM for:

- node migration
- stream splitting
- data element transfer
- control message propagation

Runtime Observations

Adaptive numerical quadrature with **aqho**
(Th. Bonk)

Computation of

$$\int_{-1}^{1}\int_{-1}^{1} |\,xy + 0.05|^{-3.5}\, dx\, dy$$

on 16 workstations of type HP9000/720:

#hosts	accuracy ε	#graph nodes	extrapolation data	runtime	active time (sum)	fixed time speedup
1	10^{-5}	15	5.31 MB	51.48 s	49.39 s	0.96
2	10^{-6}	1018	10.83 MB	59.20 s	82.84 s	1.40
4	10^{-8}	2878	23.88 MB	75.11 s	168.16 s	2.24
8	10^{-10}	7502	56.06 MB	87.43 s	330.21 s	3.78
16	10^{-12}	18558	80.91 MB	99.10 s	511.73 s	5.16

Conclusions

- **Reusability**
  parallel configuration of sequential routines written in standard languages and libraries
- **Comfort**
  same code for sequential and parallel program
- **Portability**
  data flow and migration protocol based on PVM
- **Efficiency**
  functional paradigm and data flow model: overhead, but
  - reduced communication thanks to hierarchical streams
  - maximal parallel potential through less sequential dependencies
  - project: automatic dynamic load balance