Skeleton based programming (models)

CCP'09

M. Danelutto

The principle

- The new system presents the user with a selection of independent “algorithmic skeleton”, each of which describes the structure of a particular style of algorithm, in the way in which “higher order functions” represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.

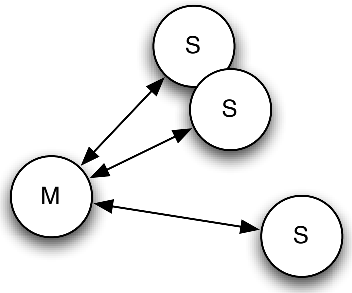
  (Cole 1988)

The principle (rephrased)

- Abstract parallelism exploitation pattern by parametric code (higher order function)
- Provide user mechanism to specify the parameters (sequential code, extra parameters)
- Provide (user protected) state-of-the-art implementation of each parallelism exploitation pattern
- In case, allow composition
  - Fundamental, property not present in first skeletons systems

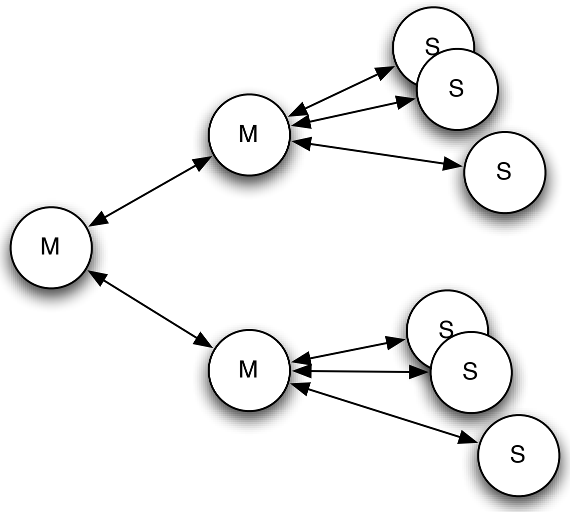
Sample pattern: the task farm

- Parameters:
  - Worker code
  - Parallelism degree (computed?)

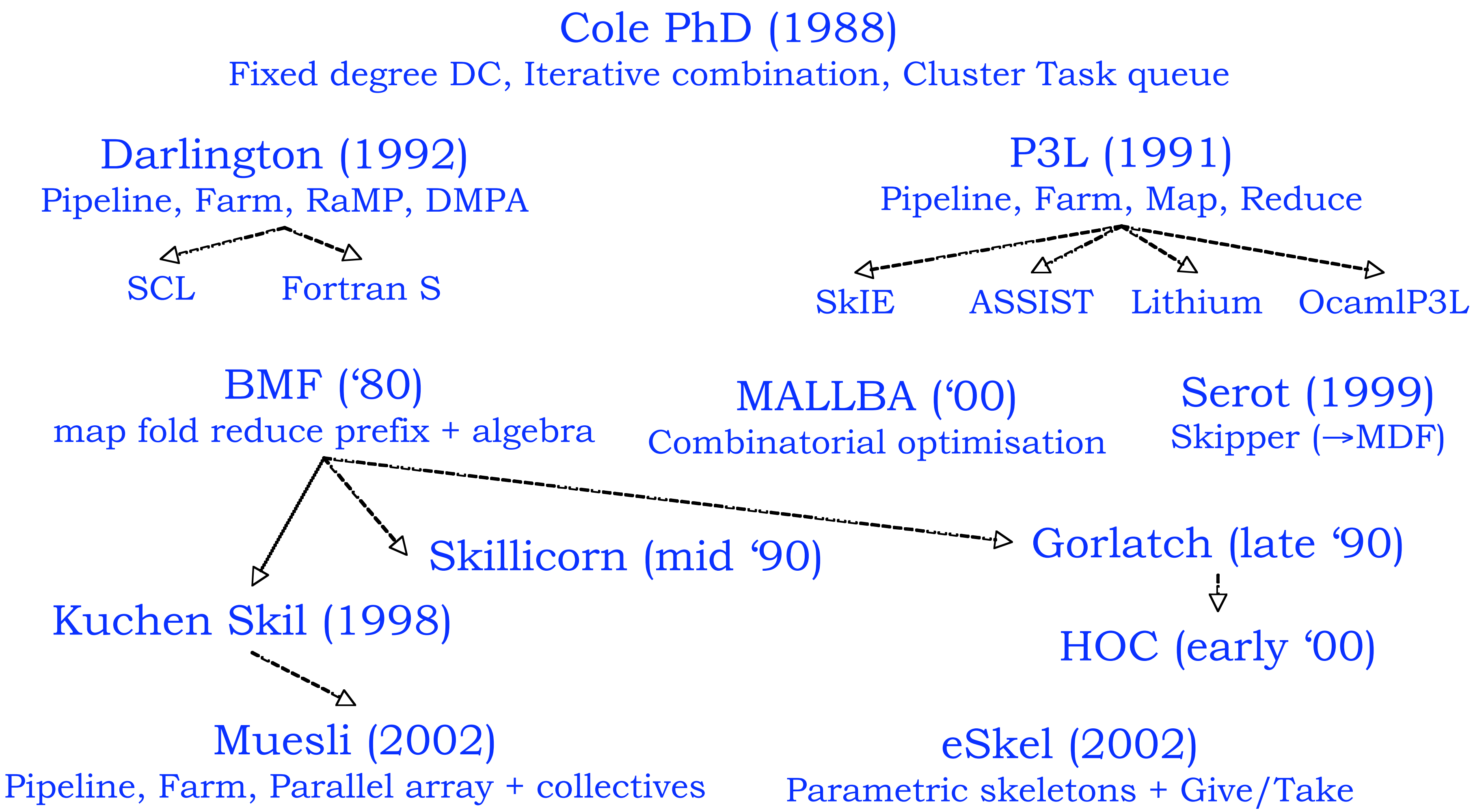


- Known implementation
  - Master slave pattern
  - Possibly distributed master

- Composite worker
  - Master to master optimizations



Skeleton evolution



Cole PhD Thesis

- Fixed degree D&C, Iterative Combination (2 “best” items in the set combined, iterated), Cluster Skeleton (abstrac machine rather than algorithm), Task Queue
- Lot of usage examples and analytical evaluation of skeletons
- Seminal work in the area
  - Due to the motivations
  - More that to the skeletons discussed
- Hierarchical composition later on (‘95 PARCO)

Darlington IC

- Coordination comes in

This is in contrast to the low level parallel extensions to languages where both tasks must be programmed simultaneously in an unstructured way. The coordination approach provides a promising way to achieve the following important goals:

- **Reusability of Sequential Code**: Parallel programs can be developed by using the coordination language to compose existing modules written in conventional languages.
- **Generality and Heterogeneity**: Coordination languages are independent of any base computational language. Thus, they can be used to compose sequential programs written in any language and can, in principle, co-ordinate programs written in several different languages.
- **Portability**: Parallel programs can be efficiently implemented on a wide range of parallel machines by specialised implementations of the compositional operators for target architectures.

- Darlington et al. Functional Skeletons for Parallel Coordination (Europar '95)

Darlington (2)

- Initially ('91)
  Farm, Pipeline, RaMP, DMP
- Then ('95):
- Coordination (see before)
- Clearer data parallel asset
- Control parallel skeletons
  (Farm, SPMD)
- Transformations !
- Fortran embedding !

```
map ::  (α → β) → ParArray index α → ParArray index β
map f << x0,...,xn >> = << f x0,...,f xn >>

imap ::  (index → α → β) → ParArray index α → ParArray index β
imap f << x0,...,xn >> = << f 0 x0,...,f n xn >>

fold ::  (α → α → α) → ParArray index α → α
fold (⊕) << x0,...,xn >> = x0 ⊕ ··· ⊕ xn

scan ::  (α → α → α) → ParArray index α → ParArray index α
scan (⊕) << x0,x1 ...,xn >> = << x0,x0 ⊕ x1,...,x0 ⊕ ··· ⊕ xn >>

farm ::  (α → β → γ) → α → ParArray index β → ParArray index γ
farm f env = map ( f env )

SPMD []              = id
SPMD (gf, lf) :  fs  = (SPMD fs) ∘ (gf ∘ (imap lf ))

map f ∘ map g = map (f ∘ g)

foldr1 (f ∘ g) = fold f ∘ map g

send f ∘ send g = send (f ∘ g)
fetch f ∘ fetch g = fetch (g ∘ f)

matrixAdd p A B = (gather ∘ map SEQ_ADD) (distribution fl dl)
   where
      C = SeqArray ((1..SIZE(A,1)), (1:SIZE(A,2)))
      fl = [((row_block p),id), ((row_block p),id), ((row_block p),id)]
      dl = [A, B, C]
```

Kuchen : Muesli

- Clearly separetes data and control parallelism exploitation

- Builds on top of MPI
- Inheriths two tier model from P3L:
- Arbitrary control parallel nestings
- With data parallel or sequential leaves
-

Muesli : nesting

```
int main(int argc, char **argv){
try{
    InitSkeletons(argc,argv);

    Initial<int>        p1(init);
    Atomic<int,int>     p2(square,1);
    Process*            p3 = NestedFarm<int,int>(p2,4);
    Final<int>          p4(fin);
    Pipe                p5(p1,*p3,p4);

    p5.start();

    TerminateSkeletons();}
  catch(Exception&){cout << "Exception" << endl << flush;}
}
```

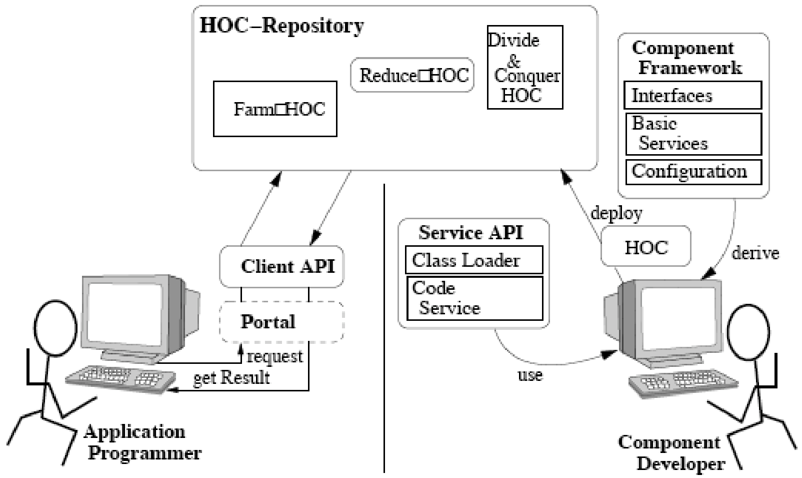
Muesli : data parallel

```
template <class C> // using algorithm of Gentleman based on torus topology
DistributedMatrix<C> matmult(DistributedMatrix<C> A,DistributedMatrix<C> B){
    A.rotateRows(& negate);
    B.rotateCols(& negate);
    DistributedMatrix<C> R(A.getRows(),A.getCols(),0,
             A.getBlocksInCol(), A.getBlocksInRow());
    for (int i=0; i< A.getBlocksInRow(); ++i){
      typedef C (*skprod_t)(const DistributedMatrix<C>&,
                           const DistributedMatrix<C>&, int, int, C);
      R.mapIndexInPlace(curry((skprod_t)skprod<C>)(A)(B));
      A.rotateRows(-1);
      B.rotateCols(-1);}
    return R;}

int main(int argc, char **argv){
try{
    InitSkeletons(argc,argv);
    DistributedMatrix<int> A(Problemsize,Problemsize,& add,sqrtp,sqrtp);
    DistributedMatrix<int> B(Problemsize,Problemsize,& add,sqrtp,sqrtp);
    DistributedMatrix<int> C = matmult(A,B);
    TerminateSkeletons();}
  catch(Exception&){cout << "Exception" << endl << flush;};
}
```

Gorlatch: HOC

- Inherits from Lithium
- Exploiting Web Services
- Higher order components
  - Farms, pipelines
- Developed in Muenster
- Joint works with
  - Caromel, Cole, Danelutto




```
public interface Worker {
    public double[] compute(double[] input);
}
public interface Master {
    public double[][] split (double[] input, int numWorkers);
    public double[] join(double[][] input);
}
```

```
farmHOC =farmFactory.createHOC();
farmHOC.setMaster("masterID"); // web service invocation in Java
farmHOC.setWorker("workerID");
String[] targetHosts = {"masterH", "workerH1", ...};
farmHOC.configureGrid(targetHosts); // deployment of the farmHOC on the Grid
farmHOC.compute(input);
```

Cole eSkel

- Local data or Spread data processing
- Implicit or explicit interaction mode
- Transient and persistent skeleton calls in a skeleton
- Pipeline, Deal (cyclic distrib farm), Farm, Butterfly
- MPI (rather glossy interface)

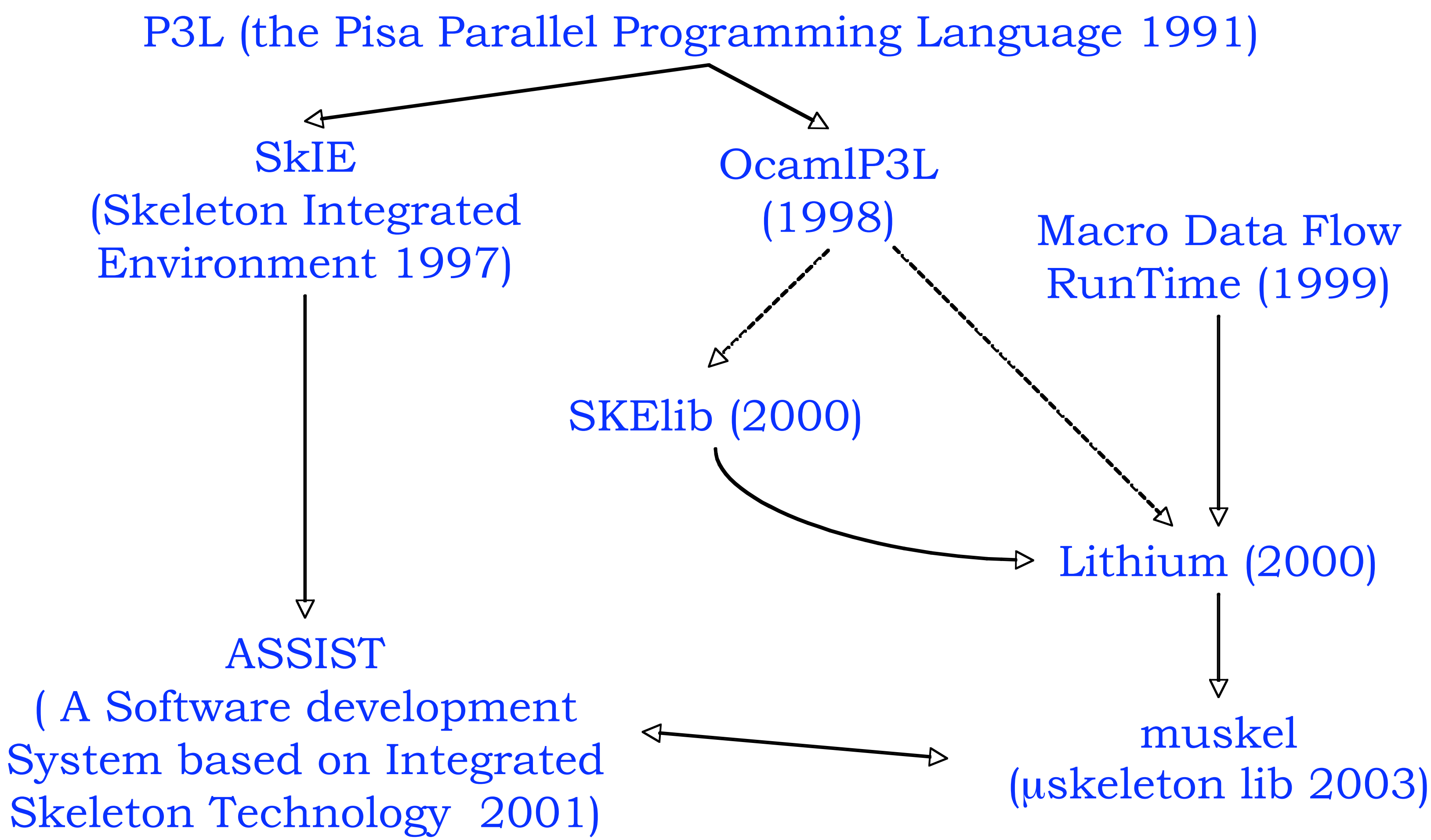
The Edinburgh Skeleton Library

- Introduction to eSkel
- eSkel's downloads NEW
- eSkel's publications
- Links

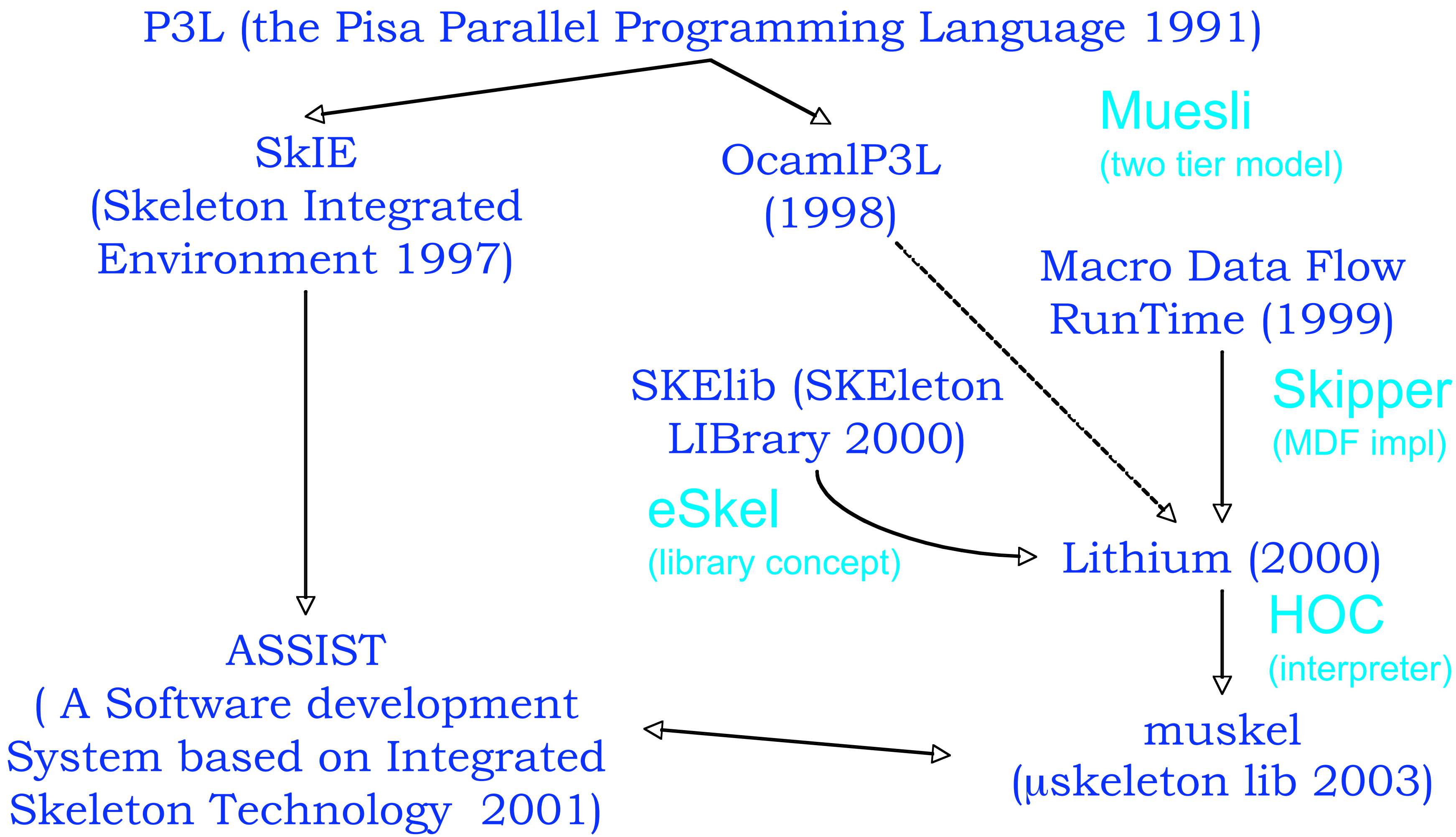
MALLBA

- Combinatorial optimization through skeletons
- Fairly “unconventional” set of skeletons
  - D&C, B&B, Dynamic Programming, Hill Climbing, Metropolis, Simulated Annealing (SA), Tabu Search (TS) and Genetic Algorithms (GA)
- C++ implementation
  - *provided* classes (fixed implementation) + *required* classes (user supplied, problem dependent code)
- Related work on performance models
- Excellent speedups on (heterogeneous CPU) clusters as well as on WAN

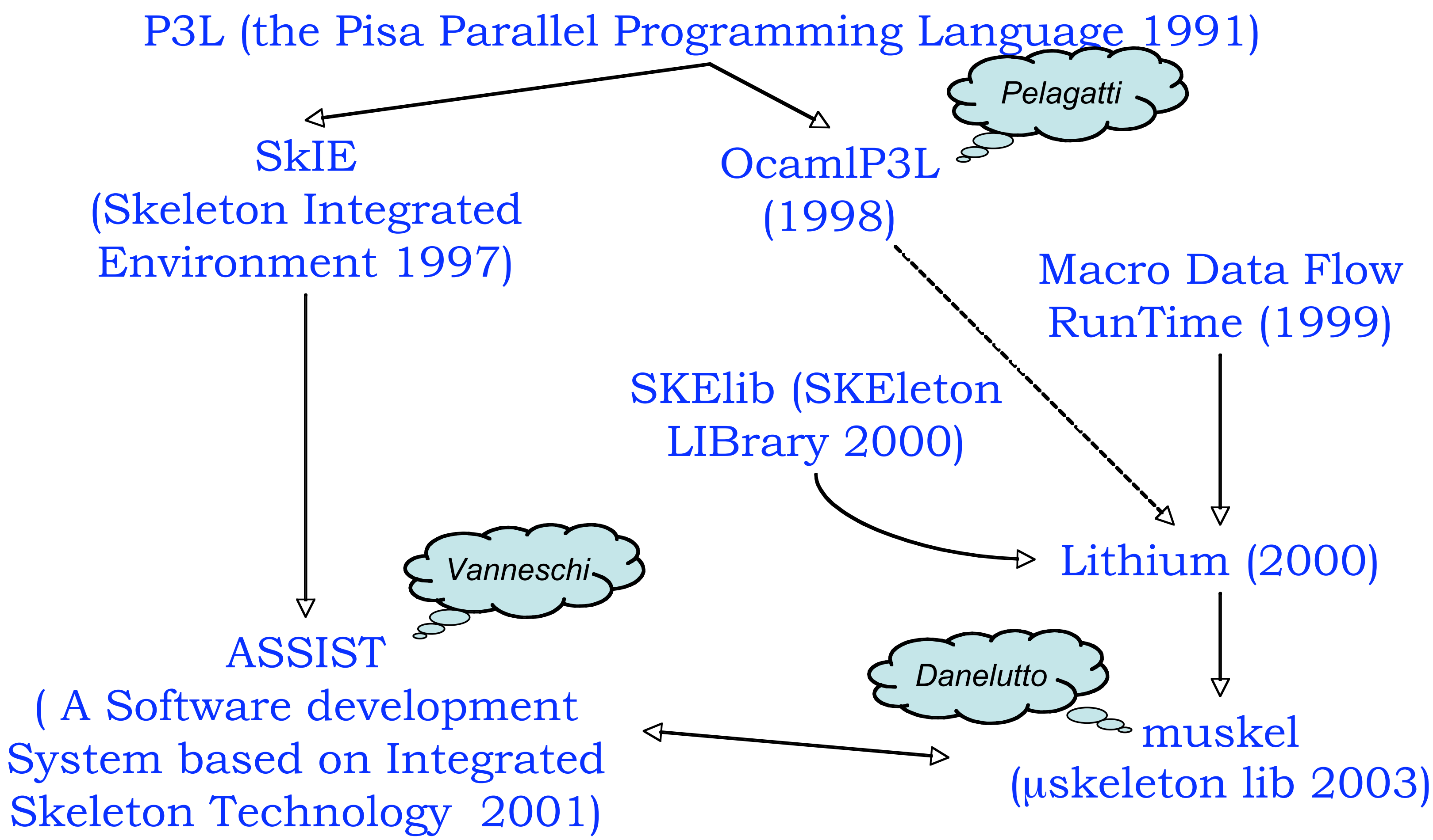
The Pisa Picture



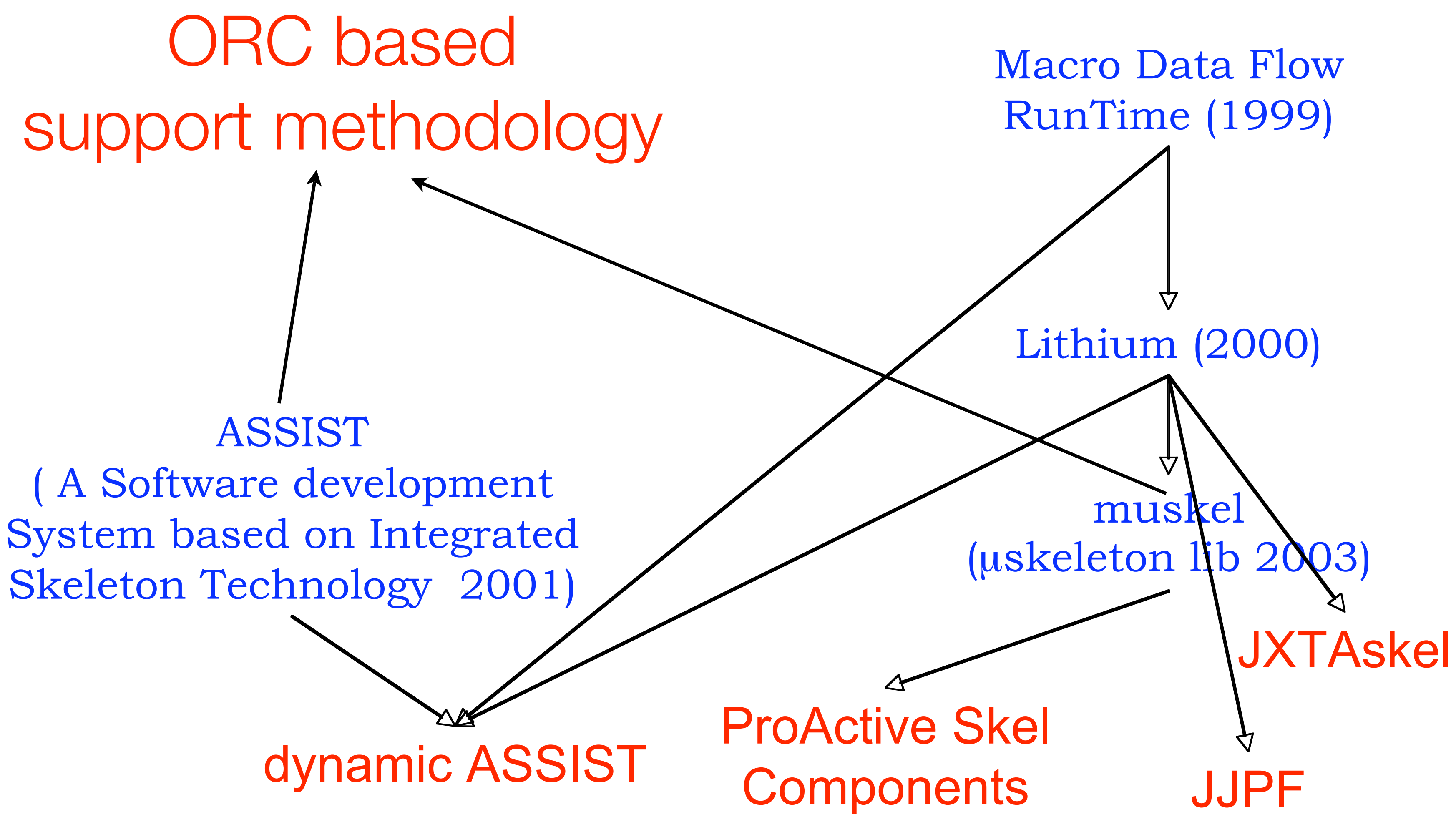
The Pisa picture



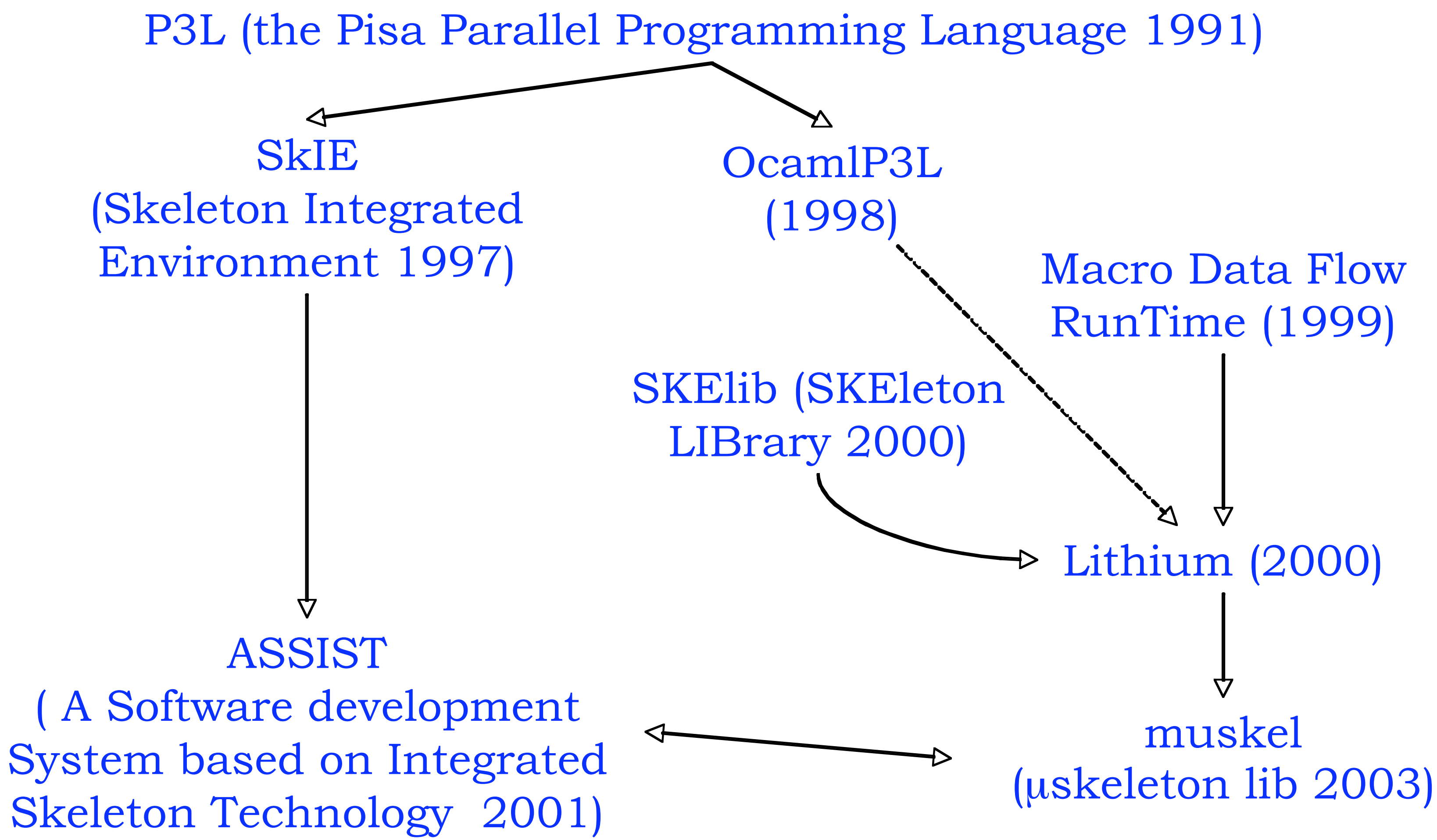
The Pisa picture: alive projects



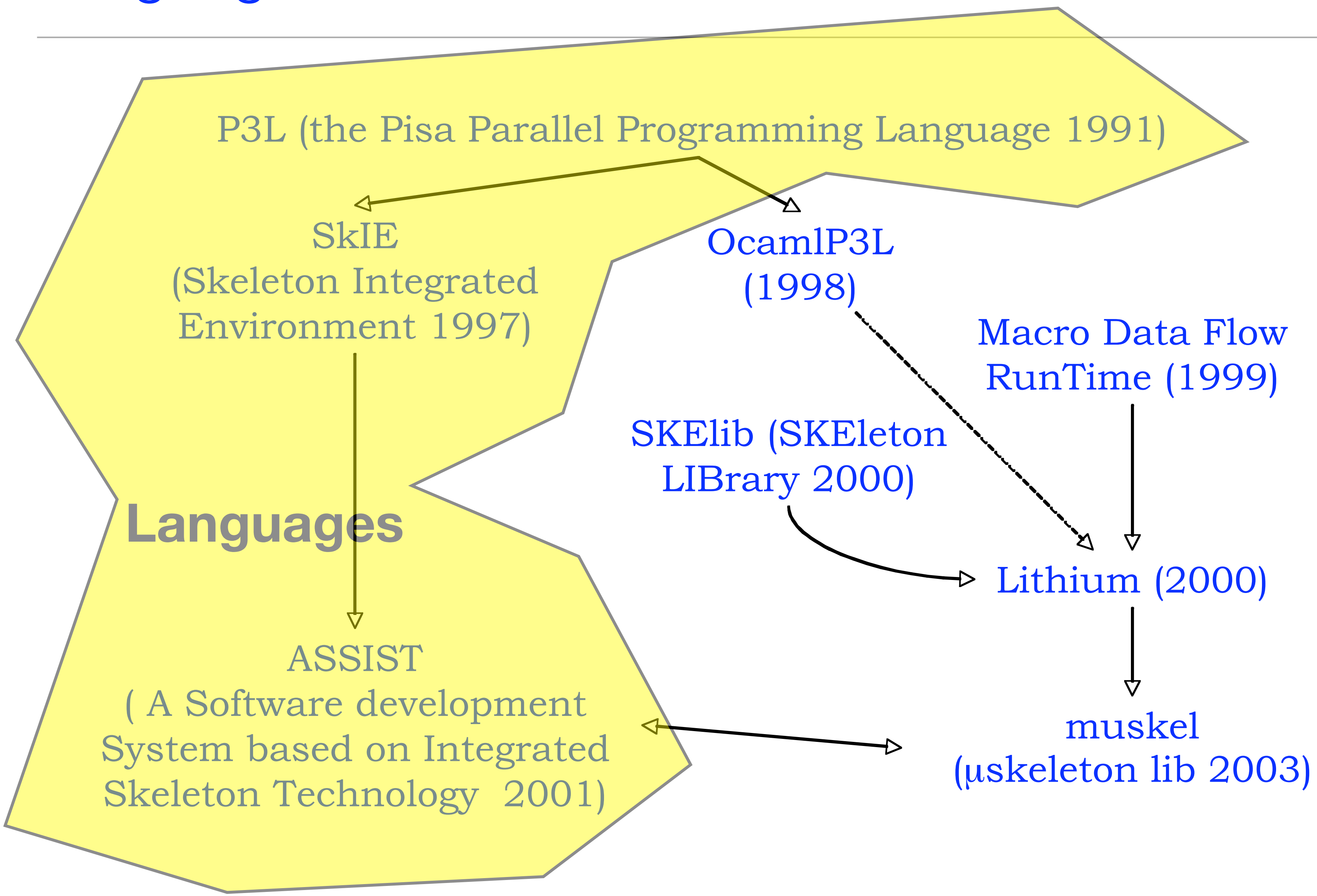
The Pisa picture : side effects ...



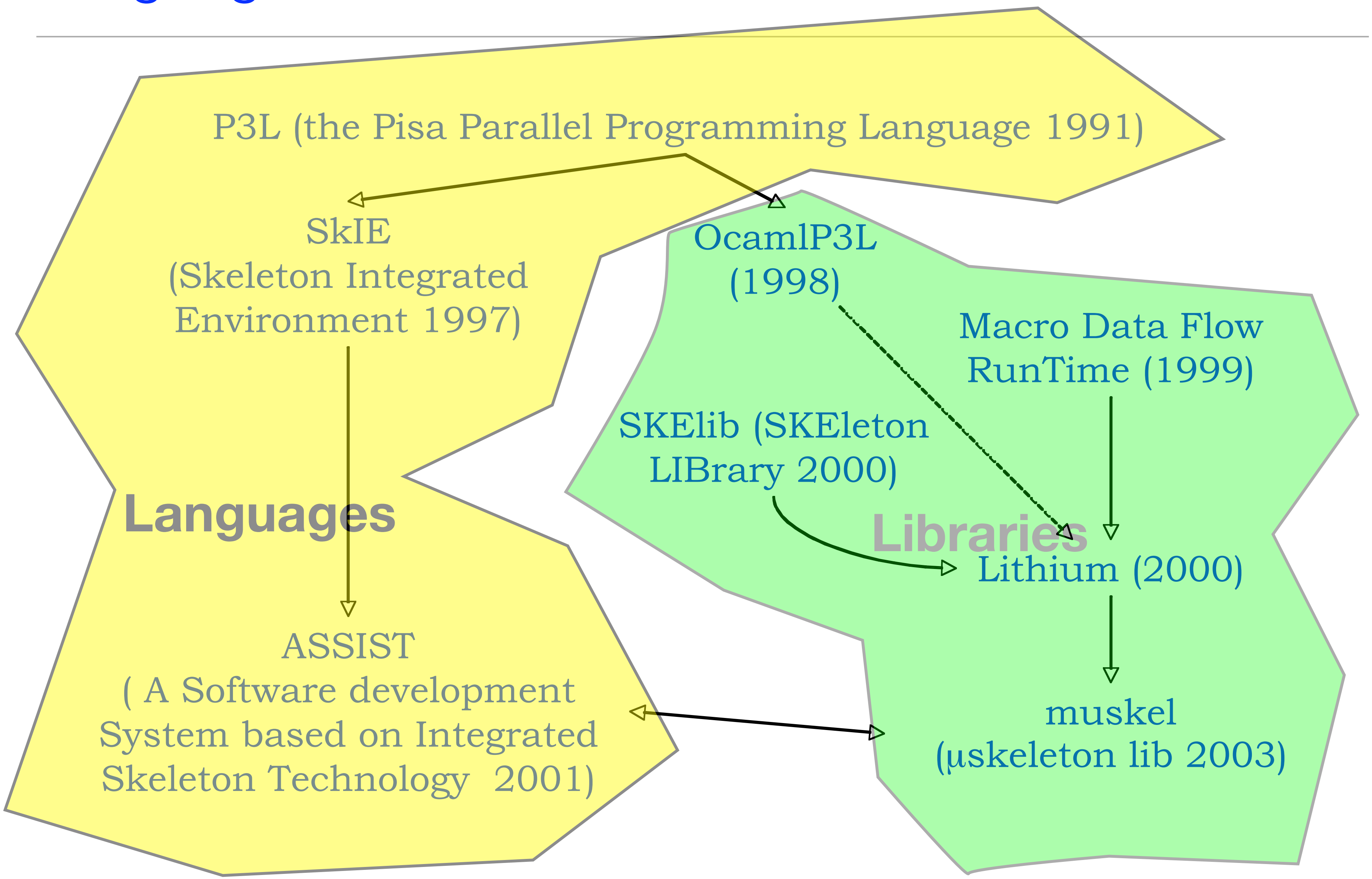
Languages vs. libraries



Languages vs. libraries



Languages vs. libraries



Languages

- Completely new language
  - Coordination language
  - Pragmas to “drive” implementation (data distribution, parallelism degree, reordering, load balancing)
- Compiler (static properties)
  - Generates (high level) source code
  - Targets specific parallel model (threads, commlibs)
  - Performs known optimizations (e.g. parallelism degree, n2m communication optimization, …)
- Run time (dynamic properties)
  - Load balancing
  - Fault tolerance

Libraries

- Library calls:
  - Declare patterns
  - Instantiate parameters
  - Drive implementation
- Implementation
  - Completely at run time (or JIT)
  - Relies on library communication facilities
  - Usually more efficient on dynamic properties handling
- “User friendly” approach (perceived)
  - No need to learn a new language

A code comparison

- P3L

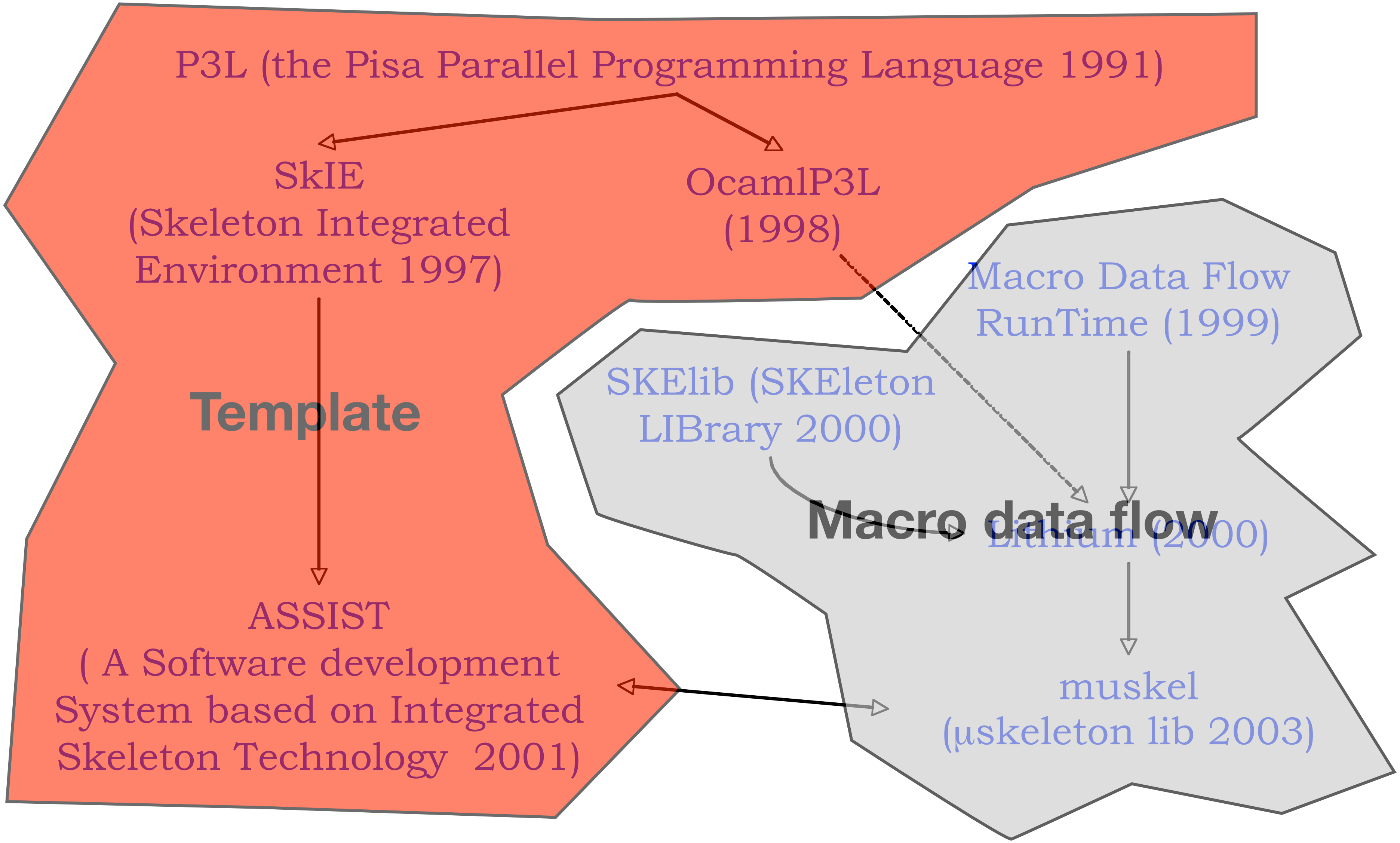
```
f in(T1 a) out(T2 b)
$c{ /* c code here … */
  b = …;
}c$

farm main in(T1 a) out(T2 b)
  nw 2
  f in(a) out(b)
end farm
```

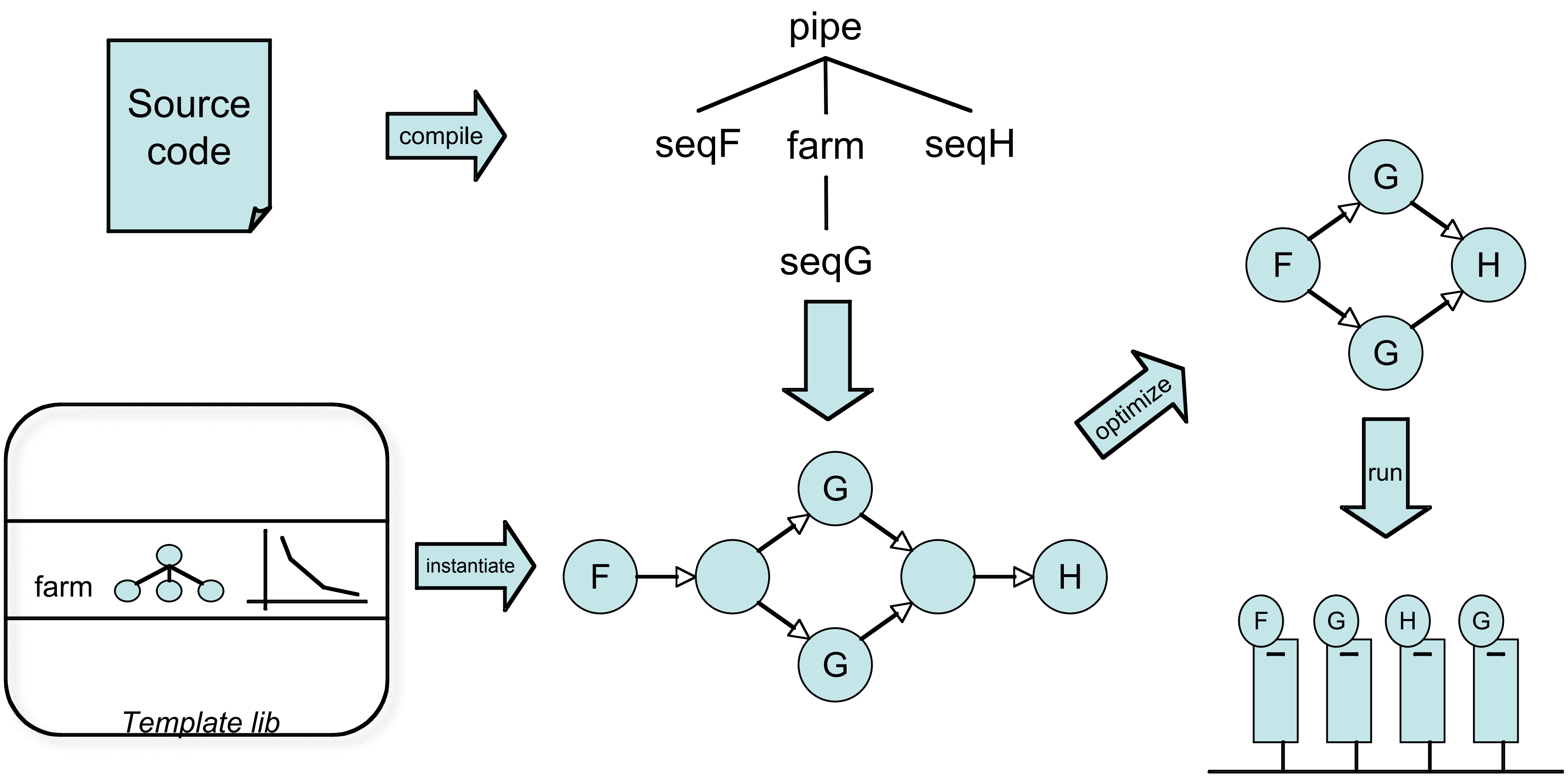
- **muskel**

```
public static void
    main(String [] args) {
  Skeleton worker = new F();
  Farm main = new Farm(f);
  Manager m = new Manager();
  m.setInputFile("…");
  m.setOutputFile("…");
  m.setProgram(main);
  m.setContract(new ParDegree(2));
  m.compute();
}
class F implements Skeleton {
  public T2 compute(T1 task) {
     T2 result = null;
    result = … ;
    return result;
   }
}
```

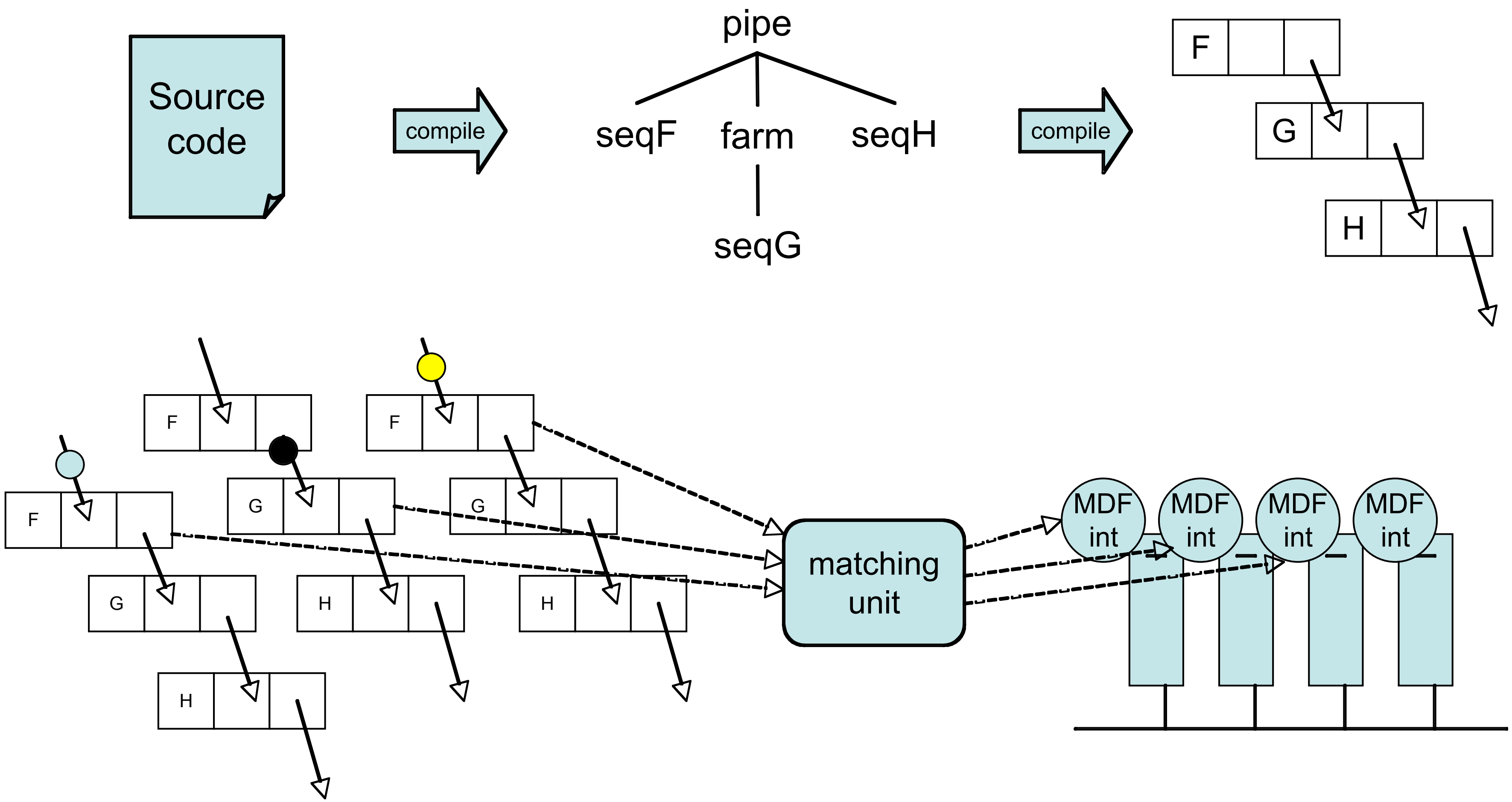
Template vs. macro data flow



Template based implementation



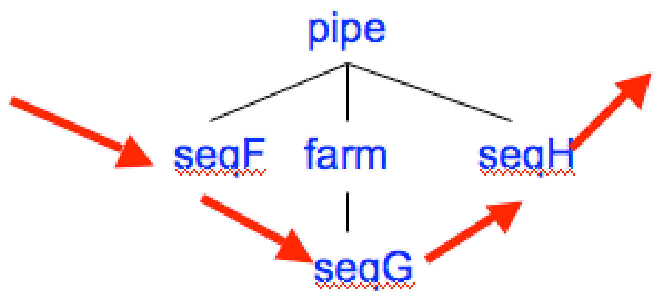
Macro data flow implementation



Optimizations: normal form

- Improve service time
- Stream parallel computations
- Coarser grain remote computations
- Automatic transformation tool (source2source)
- Proven correct, efficient (theoretically & experimentally)

- Step 1: take the frontier

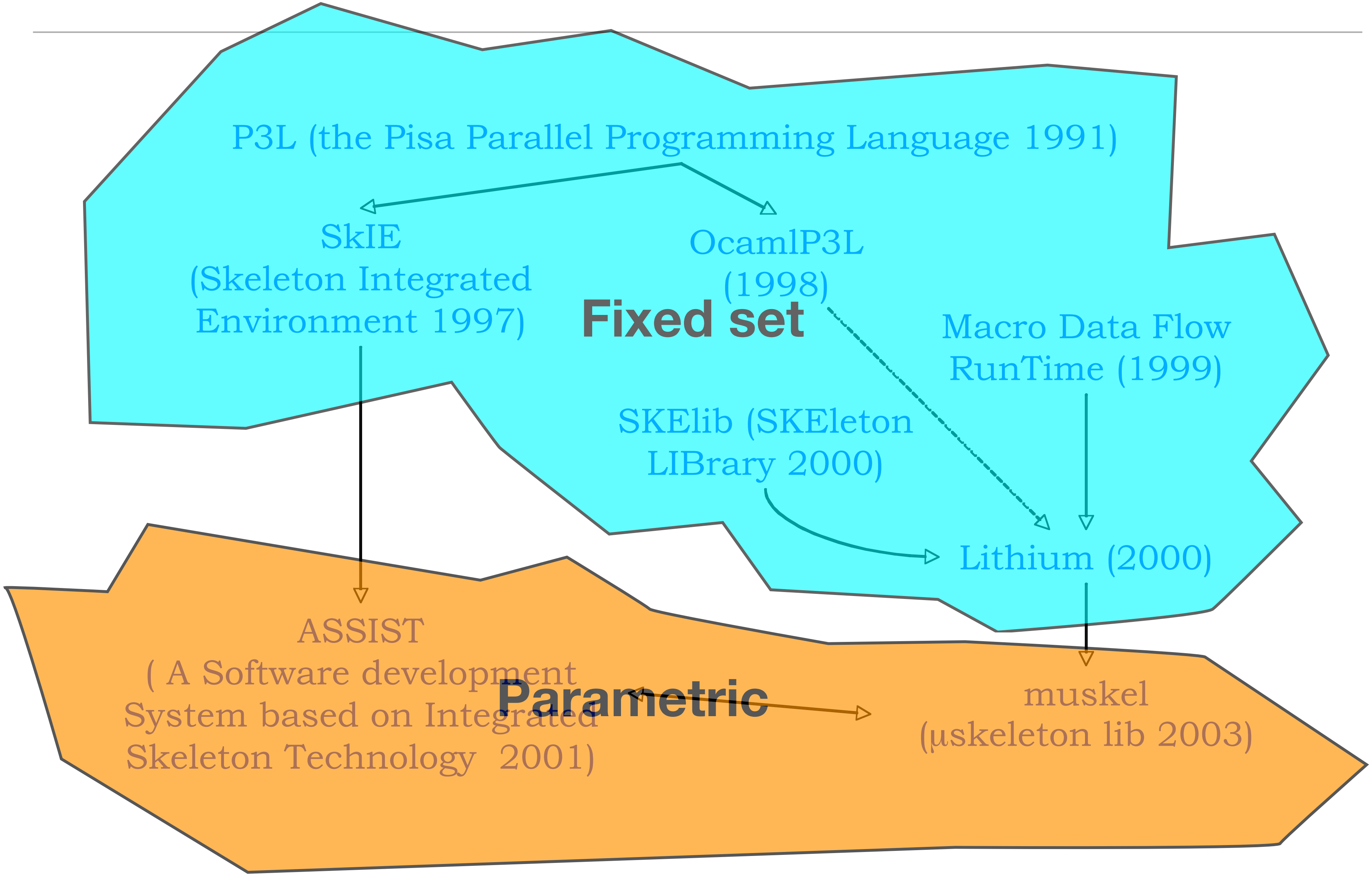


- Step 2: make a (big) seq

seqF;seqG;seqH

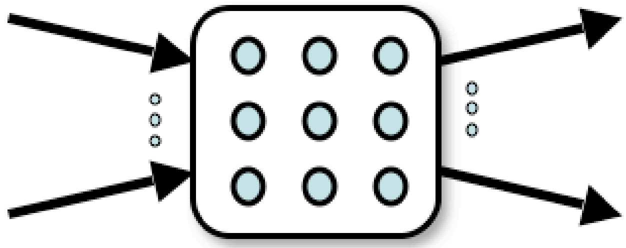
- Step 3: farm it

Fixed skeleton set vs. parametric



Parametric skeletons: the ASSIST way

- Set of virtually parallel activites (named + code)
- Shared state among virtual activities (if needed)
- Cycle control over virtual parallel activities
- Input data from data flow streams to virtual activities and state with non deterministic control
- Data from virtual activities and state to output streams



```
parmod matrix_mul (input_stream long M1[N][N], long M2[N][N]
                                output_stream long M3[N][N])
{
topology array [i:N][j:N] Pv;
attribute long A[N][N] scatter A[*ia][*ja] onto Pv[ia][ja];
attribute long B[N][N] scatter B[*ib][*jb] onto Pv[ib][jb];
stream long ris;
do input_section {
             guard1: on , , M1 && M2 {
             distribution M1[*i0][*j0] scatter to A[i0][j0];
             distribution M2[*i1][*j1] scatter to B[i1][j1];
             }
} while (true)

virtual_processes {
             elab1 (in guard1 out ris) {
               VP i, j { f_mul (in A[i][], B[][j]
                                output_stream ris);
               }
             }
}
```

User defined skeleton: the **muskel** way

- Skeleton tree ☞ normal form ☞ data flow
- User defined data flow graphs
- User friendly ways to connect them
- User programs non skeleton parallel code
- Macro data flow interpreter interprets it as plain skeleton derived code