

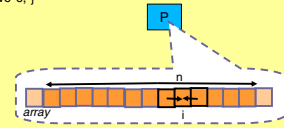


# Programming and Parallelization with Algorithmic Skeletons

## An Introduction

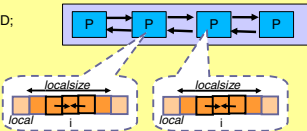
### Example: 1D smoothening in C

```
float filter (float a, b, c) { return wa*a + wb*b + wc*c; }
void main ( int argc, char *argv[] )
{
    float *array = new_FloatArray( n+2 );
    float *tmp = new_FloatArray( n+2 );
    while ( globalerr > 0.1 ) {
        for (i=1; i<=n; i++)
            tmp[i] = filter( array[i-1], array[i], array[i+1] );
        globalerr = 0.0;
        for (i=1; i<=n; i++)
            globalerr = fmax( globalerr, fabs(array[i] - tmp[i] ) );
        for (i=1; i<=n; i++)
            array[i] = tmp[i];
    }
}
```



### Example: 1D smoothening in C + MPI

```
void main ( int argc, char *argv[] ) {
    MPI_Comm com = MPI_COMM_WORLD;
    MPI_Init ( &argc, &argv );
    MPI_Comm_size ( com, &np );
    MPI_Comm_rank ( com, &me );
    localsize = (int) ceil ( (float) n / np );
    local = new_FloatArray( localsize + 2 );
    while ( globalerr > 0.1 ) {
        if (me>0) MPI_Send ( local+1, 1, MPI_FLOAT, left_neighbor, 10, com );
        if (me<np-1) MPI_Send ( last, 1, MPI_FLOAT, right_neighbor, 20, com );
        for (i=1; i<=localsize; i++)
            tmp[i] = filter( local[i-1], local[i], local[i+1] );
        if (me<np-1) MPI_Recv ( tmp, 1, MPI_FLOAT, right_neighbor, 10, com, ... );
        if (me>0) MPI_Recv ( tmp+localsize+1, 1, MPI_FLOAT, left_neighbor, 20, com, ... );
        tmp[1] = filter( local[0], local[1], local[2] );
        tmp[localsize] = filter( local[localsize-1], local[localsize], local[localsize+1] );
        localerr = 0.0;
        for (i=1; i<=localsize; i++) localerr = fmax( localerr, fabs ( local[i]-tmp[i] ) );
        MPI_Allreduce ( &localerr, &globalerr, 1, MPI_FLOAT, MPI_MAX, com );
        for (i=1; i<=localsize; i++)
            local[i] = tmp[i];
    }
}
```



### Complexity of Parallel Algorithms and Programs

- Many different parallel programming models
- Identify parallelism ("tasks")
- Synchronization and Communication?
- Memory structure, -consistency?
- Load balancing, Scheduling?
- Network structure?
- Error prone, hard to debug
- ...

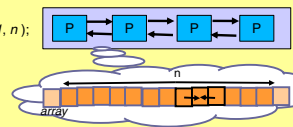
Can we make parallel programming as easy as sequential programming?

### Observation

- Same characteristic form of parallelism, communication, synchronization re-applicable for all occurrences of the same specific structure of computation ((parallel) algorithmic paradigm, building block, pattern, ...)
- Elementwise operations on arrays
- Reductions
- Scan (Prefix-op)
- Divide-and-Conquer
- Farming independent tasks
- Pipelining
- ...
- Most of these have both sequential and parallel implementations

### Example: 1D smoothening in C + Skeletons

```
float filter (float a, b, c) { return wa*a + wb*b + wc*c; }
float elemError ( float a, b ) { return fabs ( a - b ); }
void main ( int argc, char *argv[] ) {
    DistrFloatArray *array = new_DistrFloatArray ( n + 2 );
    DistrFloatArray *tmp = new_DistrFloatArray ( n + 2 );
    DistrFloatArray *err = new_DistrFloatArray ( n + 2 );
    while ( globalerr > 0.1 ) {
        map_with_overlap( filter, 1, tmp, array+1, n );
        map( elemError, err, array+1, tmp, n );
        reduce( fmax, &globalerr, err, n );
        map( copy, array+1, tmp, n );
    }
}
```



## Data parallelism



### Given:

- One or several data containers  $x$  with  $n$  elements, e.g. array(s)  $x=(x_1, \dots, x_n)$ ,  $Z=(Z_1, \dots, Z_n)$ , ...
- An operation  $f$  on individual elements of  $x$ ,  $Z$ , ... (e.g. *incr*, *sqrt*, *mult*, ...)

**Compute:**  $y = f(x) = (f(x_1), \dots, f(x_n))$

### Parallelizability: Each data element defines a task

- Fine grained parallelism
- Portionable, fits very well on all parallel architectures

### Notation with higher-order function:

- $y = \text{map}(f, x)$

**Variant:** map with overlap:  $y_i = f(x_{i+k}, \dots, x_{i+b})$ ,  $i = 0, \dots, n-1$

7

## Data-parallel Reduction

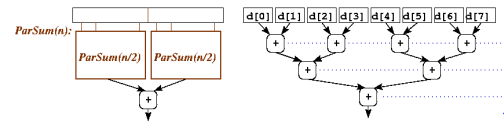


### Given:

- A data container  $x$  with  $n$  elements, e.g. array  $x=(x_1, \dots, x_n)$
- A **binary, associative** operation  $op$  on individual elements of  $x$  (e.g. *add*, *max*, *bitwise-or*, ...)

**Compute:**  $y = OP_{i=1..n} x = x_1 op x_2 op \dots op x_n$

### Parallelizability: Exploit associativity of op



### Notation with higher-order function:

- $y = \text{reduce}(op, x)$

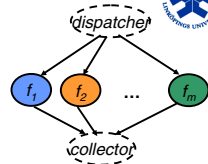
8

## Task farming



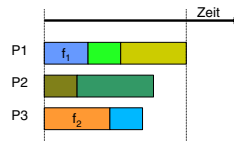
**Independent computations**  $f_1, f_2, \dots, f_m$  could be done in parallel and/or in arbitrary order, e.g.

- independent loop iterations
- independent function calls



### Scheduling problem

- $n$  tasks onto  $p$  processors
- static or dynamic
- Load balancing



### Notation with higher-order function:

- $(y_1, \dots, y_m) = \text{farm}(f_1, \dots, f_m)(x_1, \dots, x_n)$

9

## Parallele Divide-and-Conquer



### (Sequential) Divide-and-conquer:

- Divide:** Decompose problem instance  $P$  in one or several **smaller independent** instances of the same problem,  $P_1, \dots, P_k$
- For all  $i$ : If  $P_i$  *trivial*, solve it *directly*.
- Else, solve  $P_i$  by recursion.
- Combine** the solutions of the  $P_i$  into an overall solution for  $P$

### Parallel Divide-and-Conquer:

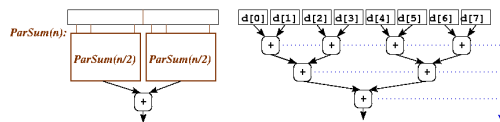
- Recursive calls can be done in parallel.
- Parallelize, if possible, also the divide and combine phase.
- Switch to sequential divide-and-conquer when enough parallel tasks have been created.

### Notation with higher-order function:

- $\text{solution} = \text{DC}(\text{divide}, \text{combine}, \text{istrivial}, \text{solvedirectly}, n, P)$

10

## Example: Parallel Divide-and-Conquer



### Example: Parallel Sum over integer-array $x$

Exploit associativity:

$$\text{Sum}(x_1, \dots, x_n) = \text{Sum}(x_1, \dots, x_{n/2}) + \text{Sum}(x_{n/2+1}, \dots, x_n)$$

Divide: trivial, split array  $x$  in place

Combine is just an addition.

$$y = \text{DC}(\text{split}, \text{add}, \text{nlsSmall}, \text{addFewInSeq}, n, x)$$

Data parallel reductions are an important special case of DC.

11

## Example: Parallel Divide-and-Conquer (2)



### Example: Parallel QuickSort over a float-array $x$

Divide: Partition the array (elements  $\leq$  pivot, elements  $>$  pivot)

Combine: trivial, concatenate sorted sub-arrays

$$\text{sorted} = \text{DC}(\text{partition}, \text{concatenate}, \text{nlsSmall}, \text{qsort}, n, x)$$

12

## Pipelining



applies a sequence of dependent computations ( $f_1, f_2, \dots, f_k$ ) elementwise to data sequence  $x = (x_1, \dots, x_n)$

- For fixed  $x_p$  compute  $f_i(x_p)$  before  $f_{i+1}(x_p)$
- Computations of  $f_i$  on different  $x_j$  are independent.

**Parallelizability:** Overlap execution of all  $f_i$  for  $k$  subsequent  $x_j$

- $time=1$ : compute  $f_1(x_1)$
- $time=2$ : compute  $f_1(x_2)$  and  $f_2(x_1)$
- $time=3$ : compute  $f_1(x_3)$  and  $f_2(x_2)$  and  $f_3(x_1)$
- ...
- Total time:  $O((n+k) \max_i(time(f_i)))$  with  $k$  processors

**Notation** with higher-order function:

- $(y_1, \dots, y_n) = \text{pipe}((f_1, \dots, f_k), (x_1, \dots, x_n))$

## Skeletons



**Skeletons** are reusable, parameterized components with well defined semantics for which efficient parallel implementations may be available.

Inspired by higher-order functions in functional programming

- solid formal basis: Homomorphisms on lists

One or very few skeletons per parallel algorithmic paradigm

- map, farm, DC, reduce, pipe, scan ...

Parameterised in user code

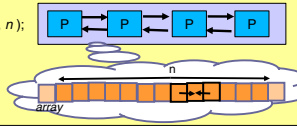
Composition of skeleton instances in program code by sequencing+data flow

- e.g. `squaresum(x) { tmp = map(sqr,x); return reduce( add, tmp ); }`
- or by function composition:  $(f \circ g) x := f(g(x))$
- e.g. `squaresum = reduce add o map sqr`

## Example revisited: 1D smoothing in C + Skeletons



```
float filter(float a, b, c) { return wa*a + wb*b + wc*c; }
float elemError(float a, b) { return fabs(a - b); }
void main(int argc, char* argv[]) {
    ...
    DistrFloatArray *array = new_DistrFloatArray(n+2);
    DistrFloatArray *tmp = new_DistrFloatArray(n+2);
    DistrFloatArray *err = new_DistrFloatArray(n+2);
    ...
    while(globalErr > 0.1) {
        map_with_overlap(filter, 1, tmp, array+1, n);
        map(elemError, err, array+1, tmp, n);
        reduce(fmax, &globalErr, err, n);
        map(copy, array+1, tmp, n);
    }
    ...
}
```



## Skeletons (cont.)



Complex skeletons (**DC**, **pipe**, ...) can be defined by simpler ones (**map**, **farm**) (which yields a default implementation, but more efficient ones may exist)

**DC** divide combine trivial solve P  
= if (trivial P) then (solve P)

else combine (**farm** (**DC** divide combine trivial solve) (divide P

**Ideally:** Skeletons encapsulate completely all coordination of parallelism

Threads/Process creation/termination, communication, synchronization

**Reuse of the coordination code**

Skeletons may also have a sequential implementation

Uniform treatment of sequential and parallel programming

Associate a cost function with each skeleton

Composition of the cost function of a program in same way as for skeletons

## Nesting of Skeletons



Skeletons are (higher-order) functions and may thus parameterize other skeletons...

This creates **nested parallelism**.

There may exist several possibilities for nesting.

Example: Matrix-Vector-Product:  $y = Ax, y_i = \text{Sum}(j=1..n) (A_{ij} * x_j)$

(a) Reduce  $n$  whole  $m$ -vectors:

$y = \text{reduce}(j=1..n) (\text{map}(i=1..m) \text{add}) (\text{map}(i=1..m) \text{mult}) (A_{ij} * x_j)$

(b)  $m$  dot products of length  $n$  in parallel:

$y = \text{map}(i=1..m) (\text{reduce}(j=1..n, \text{map}(j=1..n, \text{mult}, A_{ij}, x_j))$

Selection of best variant e.g. guided by predicted cost

Cost guided transformation of skeleton programs [Gorlatch, Pelagatti '98]

Alternatively: **Sequential composition** = chaining by data flow

## Skeleton Programming Systems



**4 basic approaches for realizing skeletons (esp., parameterisation mechanism):**

- Library of higher-order functions (functional or imperative)
- OO class library (*subclass and define abstract parameter method(s)*)
- New language constructs (*intrinsic / compiler-known functions*)
- Generative programming, Static metaprogramming (*Macros / templates*)

**Many research prototypes, e.g.:**

- P3L* - C + skeletons
- SCL, Eden, HDC* - functional
- SkE / FAN* - graphic editor + rule based transformation system for P3L
- eSkel* - C + MPI
- Lithium* - Java + RMI
- BlockLib* - C + macros (generative) + DMA for Cell BE
- muskel, ASSIST* - C++, grid computing
- MueSLi, QUAFF* - C++ based, MPI

**Domain-specific Skeleton Systems, e.g.**

- MailBa* (combinatorial optimization: BB, DP, GA, ...)
- MapReduce* (distributed data mining, Google)

## Example: Skeletons in P3L

**Data-parallel skeletons:** P3L has its own composition language

Reduce Skeleton

```
reduce R in(int A[n]) out(float Y)
  for sp in(A[*]) out(Y)
end reduce
```

Map Skeleton

```
map M in(int A[n]) out(int B[n])
  W in(A[*]) out(float B[*])
end map
```

Comp Skeleton

```
comp C in(int A[n][m]) out(float B[n][m])
  <List of data parallel skeletons>
end comp
```

**Control-parallel skeletons:**

Sequential Skeleton

```
seq S in(int x) out(float y)
  <User Defined Code>
end seq
```

Pipeline Skeleton

```
pipe P in(int x) out(float y)
  <List of Stages>
end pipe
```

Farm Skeleton

```
farm F in(int x) out(float y)
  <Worker Call>
end farm
```

Loop Skeleton

```
loop L in(int x) out(float y) feedback(x)
  <Body Condition>
  <Body Call>
end loop
```

Image source: S. Gortlach, Tutorial: "Parallel Programming with Skeletons: Theory and Practice", La Laguna, Dec. 1999

## Visual Editor for P3L (SKIE)

Image source: S. Gortlach, Tutorial: "Parallel Programming with Skeletons: Theory and Practice", La Laguna, Dec. 1999

## MueSLi: Skeletons in C++, using templates and other functional features of C++

**Example: Task Parallelism**

```
#include "Skeleton.h"
static int current = 0;
static const int numworkers = 2;
int* init() { if (current++ < 99) return &current; else return NULL; }
int times(int x, int y) { return x * y; }
void fin(int n) { cout << "result: " << n << endl; }
int main(int argc, char **argv) {
  InitSkeletons(argc, argv);
  // step 1: create a process topology (using C++ constructors)
  Initial<int> p1(init);
  Process* p2[numworkers];
  for (int i=0; i<numworkers; i++)
    p2[i] = new Atomic<int,int>(curry(times)(i+1),1);
  Farm<int,int> p3(p2,numworkers);
  Final<int> p4(fin);
  Pipe p5(p1,p3,p4);
  // step 2: start the system of processes
  p5.start();
  TerminateSkeletons();
}
```

Compared to equivalent handwritten message-passing MPI programs, the MueSLi programs have only 30..40% overhead on average.

Image source: H. Kuchen, Univ. Münster, Germany  
Slides of a presentation of MueSLi, 2002.

## BlockLib Skeleton Library for Cell BE

- Generative approach – using C preprocessor macros
- Hides complexity of SPE code
- Dataparallel skeletons
  - map, reduce, map+reduce, map-with-overlap
- Same speedup as hand-written low-level code
- Faster than IBM Cell SDK 3.0 BLAS-1 library for p>2 SPEs

[Ålind, Eriksson, K., IWMSE-2008]

ODE solver application using BlockLib skeletons

## Summary

- Skeleton programming**
  - Algorithmic paradigms
  - Predefined parallel components, parameterized in user code
  - Hiding complexity (parallelism and low-level programming)
- ⊙ Abstraction
- ⊙ Enforces structuring
- ⊙ Parallelization for free
- ⊙ Easier to analyze and transform
- ⊙ Requires complete understanding and rewriting
- ⊙ Available skeleton set does not always fit
- ⊙ May lose some efficiency compared to manual parallelization

- Industry (beyond HPC domain) has discovered skeletons
  - map, reduce, scan in many modern parallel programming APIs
    - e.g., Intel Threading Building Blocks (TBB): par. for, par. reduce, pipe
  - Google MapReduce (for distributed data mining applications)

## Some literature on skeleton programming

- M. Cole: *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press & Pitman, 1989.  
<http://homepages.inf.ed.ac.uk/mic/Pubs/pubs.html>
- S. Pelagatti: *Structured Development of Parallel Programs*. Taylor and Francis, 1998.
- F. Rabhi and S. Gortlach (eds.): *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2003.
- H. Bischof, S. Gortlach, E. Kitzelmann. Cost Optimality and Predictability of Parallel Programming with Skeletons. Proc. Euro-Par 2003, LNCS 2790 p.682-693

See also:  
Workshops on *high-level parallel programming*: HIPS, HLPP, CMPP, ...  
Also: Major parallel processing conferences, e.g. Euro-Par, IPDPS, ...