Programming and Parallelization with Algorithmic Skeletons

An Introduction

Example: 1D smoothening in C

```c
float filter(float a, b, c) { return wa*a + wb*b + wc*c; }
```

```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

```c
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?

Example: 1D smoothening in C + Skeletons

```c
float filter(float a, b, c) ( return wa*a + wb*b + wc*c; )
```

```c
void main(int argc, char *argv[]) {
    DistFloatArray *array = new_DistrFloatArray(n+2);
    DistFloatArray *tmp = 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);
    }
}
```

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
**Data parallelism**

Given:

One or several data containers \( x \) with \( n \) elements, e.g. array(s) \( x = (x_1, \ldots, x_n) \), \( z = (z_1, \ldots, z_n) \), …

An operation \( f \) on individual elements of \( x, z, \ldots \)

(e.g. incr, sqrt, mult, …)

Compute: \( y = f(x) = (f(x_1), \ldots, f(x_n)) \)

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: \( y_i = f(x_{i-k}, \ldots, x_i+k), \ i = 0, \ldots, n-1 \)

**Data-parallel Reduction**

Given:

A data container \( x \) with \( n \) elements, e.g. array \( x = (x_1, \ldots, x_n) \)

A binary, associative operation \( \text{op} \) on individual elements of \( x \)

(e.g. add, max, bitwise-or, …)

Compute: \( y = \text{OP}_{i=1}^{i=n} x = x_1 \text{op} x_2 \text{op} \ldots \text{op} x_n \)

Parallelizability: Exploit associativity of \( \text{op} \)

Notation with higher-order function:

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

**Task farming**

Independent computations \( f_1, f_2, \ldots, f_m \)

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:

\( (y_1, \ldots, y_3) = \text{farm} \ (f_1, \ldots, f_m \ (x_1, \ldots, x_n) \)

**Paralleles Divide-and-Conquer**

(Sequential) Divide-and-conquer:

Divide: Decompose problem instance \( P \) in one or several smaller

independent instances of the same problem, \( P_1, \ldots, P_k \)

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:

\( \text{solution} = \text{DC} \ (\text{divide}, \text{combine}, \text{istivial}, \text{solvedirectly}, \ n, \ P) \)

**Example: Parallel Divide-and-Conquer**

Example: Parallel Sum over integer-array \( x \)

Exploit associativity:

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

Divide: trivial, split array \( x \) in place

Combine is just an addition.

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

Data parallel reductions are an important special case of DC.

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

Example: Parallel QuickSort over a float-array \( x \)

Divide: Partition the array \((\text{elements} \leq \text{pivot}, \text{elements} > \text{pivot})\)

Combine: trivial, concatenate sorted sub-arrays

\( \text{sorted} = \text{DC} \ \text{partition} \ \text{concatenate} \ \text{nisSmall} \ \text{qsort} \ n \ x \)
**Pipelining**

applies a sequence of dependent computations \((f_1, f_2, \ldots, f_k)\)

elementwise to data sequence \(x = (x_1, \ldots, x_n)\)

For fixed \(x_j\), compute \(f_i(x_j)\) before \(f_{i+1}(x_j)\)

Computations of \(f_i\) on different \(x_j\) are independent.

**Parallelizability:**

Overlap execution of all \(f_i\) for \(k\) subsequent \(x_j\)

\[
\begin{align*}
\text{time}=1: & \quad f_1(x_1) \\
\text{time}=2: & \quad f_1(x_2) \quad \text{and} \quad f_2(x_1) \\
\text{time}=3: & \quad f_1(x_3) \quad \text{and} \quad f_2(x_2) \quad \text{and} \quad f_3(x_1) \\
\cdots & \\
\text{Total time:} & \quad O((n+k)\max\{\text{time}(f_j)\}) \quad \text{with} \quad k \quad \text{processors}
\end{align*}
\]

**Notation** with higher-order function:

\[(y_1, \ldots, y_n) = \text{pipe}\ (f_1, \ldots, f_k, x_1, \ldots, x_n)\]

---

**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

\begin{itemize}
\item e.g. squaresum(x) \((\text{tmp} = \text{map}(\text{sqr}, x)); \text{return reduce}(\text{add}, \text{tmp});)\)
\item or by function composition: \((f \circ g)x := f(g(x))\)
\item e.g. squaresum = \text{reduce}(\text{add}) \circ \text{map}(\text{sqr})
\end{itemize}

---

**Example revisited:** 1D smoothening in C + Skeletons

```c
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);
    }
    ...
}
```

---

**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:

\[y = Ax, \quad y_i = \sum_{j=1..n} (A_{ij} \cdot x_j)\]

(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 (intrinsics / compiler-known functions)
- Generative programming, Static metaprogramming (Macros / templates)

Many research prototypes, e.g.:

- P3L - C + skeletons
- SCL, Eden, HDC - functional
- SKIE / FAN - graphic editor + rule based transformation system for P3L
- eelk - C + MPI
- Lithium - Java + RMI
- BlockLib - C + macros (generative) + DMA for Cell BE
- Muskel, ASSIST - C++, grid computing
- MusELI, QUAFF - C++, based, MPI
- Domain-specific Skeleton Systems, e.g.
  - Muskel (combinatorial optimization: BB, DP, GA, …)
  - MapReduce (distributed data mining, Google)

---
Example: Skeletons in P3L

Data-parallel skeletons:

- Reduce Skeleton
- Map Skeleton
- Compose Skeleton

Control-parallel skeletons:

- Sequential Skeleton
- Preadex Skeleton
- Future Skeleton
- Loop Skeleton

Visual Editor for P3L (SkIE)

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

Example: Task Parallelism

```
#include "Skeleton.h"

static int niter = 1;
static int nprocs = 2;

int main(int argc, char *argv[])
{
    int status;
    MPI_Init(&argc, &argv);
    if (argc < 2) { return 0; }
    niter = atoi(argv[1]);
    //... create a process topology (using C++ constructions)
    ParallelEnv pe(nprocs);
    Process p(nprocs);
    for (int i = 0; i < niter; i++)
    {
        p[i] = new AtomicValue<int> (i + 1);
        p[i + nprocs] = new ParallelInput<int> (p[i], p[i + nprocs]);
    }
    p[0] = new ParallelOutput<int> (0);
    for (int i = 0; i < niter; i++)
    {
        p[i + nprocs] = new ParallelOutPut<int> (p[i + nprocs], p[i]);
    }
    //... start the system of processes
    p.start();
    p.end();
    MPI_Finalize();
}
```

Randomized ODE solver application using BlockLib skeletons

<table>
<thead>
<tr>
<th>Speedup (n=20000)</th>
<th>Speedup (n=10000)</th>
<th>Speedup (n=50000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4</td>
<td>3.2</td>
<td>3.9</td>
</tr>
</tbody>
</table>

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


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

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)