Advanced parallel programming

Dottorato di Ricerca
Dip. Informatica di Pisa

M. Danelutto
Giugno-Luglio 2007
Program

• Introduction

• Classical programming models

• Structured programming models

• **Skeleton parallel programming environments**

• Open problems

• Project
Algorithmical skeletons
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 (1988)
Fixed degree DC, Iterative combination, Cluster Task queue

Darlington (1992)
Pipeline, Farm, RaMP, DMPA

P3L (1991)
Pipeline, Farm, Map, Reduce

SCL
Fortran S

BMF ('80)
map fold reduce prefix + algebra

Skillicorn (mid '90)

Muesli (2002)
Pipeline, Farm, Parallel array + collectives

Kuchen Skil (1998)

MALLBA ('00)
Combinatorial optimisation

Serot (1999)
Skipper (→MDF)

Gorlatch (late '90)
HOC (early '00)

Skillicorn (mid '90)

eSkel (2002)
Parametric skeletons + Give/Take
Cole PhD Thesis

• Fixed degree D&C, Iterative Combination (2 “best” items in the set combined, iterated), Cluster Skeleton (abstract 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 f1 dl)
  where
  C = SeqArray ((1:.SIZE(A,1)), (1:.SIZE(A,2)))
  f1 = [((row.block p),id), ((row.block p),id), ((row.block p),id)]
  dl = [A, B, C]
```
Kuchen : Muesli

- Clearly separates data and control parallelism exploitation

- Builds on top of MPI

- Inherits two tier model from P3L:
  - Arbitrary control parallel nestings
  - With data parallel or sequential leaves
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;}
}
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
  - Caramel, Cole, Danelutto

```java
public interface Worker {
    public double[] compute(double[] input);
}

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

```java
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)
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

P3L (the Pisa Parallel Programming Language 1991)

SkIE
(Skeleton Integrated Environment 1997)

OcamlP3L
(1998)

Macro Data Flow RunTime (1999)

SKElib (2000)

Lithium (2000)

ASSIST
(A Software development System based on Integrated Skeleton Technology 2001)

muskel
(μskeleton lib 2003)
The Pisa picture

P3L (the Pisa Parallel Programming Language 1991)

SkIE
(Skeleton Integrated Environment 1997)

OcamlP3L
(1998)

Muesli
(two tier model)

Macro Data Flow RunTime (1999)

SKElib (SKEleton LIBrary 2000)

eSkel
(library concept)

Lithium (2000)

Skipper
(MDF impl)

HOC
(interpreter)

muskel
(μskeleton lib 2003)

ASSIST
(A Software development System based on Integrated Skeleton Technology 2001)
The Pisa picture: alive projects

P3L (the Pisa Parallel Programming Language 1991)

SkIE
(Skeleton Integrated Environment 1997)

OcamlP3L
(1998)

Macro Data Flow RunTime (1999)

SKElib (SKEleton LiBrary 2000)

Lithium (2000)

muskel
(µskeleton lib 2003)

ASSIST
(A Software development System based on Integrated Skeleton Technology 2001)

Vanneschi

Danelutto

Pelagatti
The Pisa picture: side effects ...

**ORC based support methodology**

- **ASSIST**
  (A Software development System based on Integrated Skeleton Technology 2001)

- **dynamic ASSIST**

- **ProActive Skel Components**

- **Macro Data Flow RunTime (1999)**

- **Lithium (2000)**

- **muskel (μskeleton lib 2003)**

- **JXTA skel**

- **JJPF**
Languages vs. libraries

Languages
- P3L (the Pisa Parallel Programming Language 1991)
- SkIE (Skeleton Integrated Environment 1997)
- ASSIST (A Software development System based on Integrated Skeleton Technology 2001)

Libraries
- OcamlP3L (1998)
- SKElib (SKEleton LIBrary 2000)
- Macro Data Flow RunTime (1999)
- Lithium (2000)
- muskel (μskeleton lib 2003)
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**

```plaintext
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**

```java
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
- P3L (the Pisa Parallel Programming Language 1991)
- SkIE (Skeleton Integrated Environment 1997)
- OcamlP3L (1998)
- ASSIST (A Software development System based on Integrated Skeleton Technology 2001)

Macro data flow
- SKElib (SKEleton LiBrary 2000)
- Lithium (2000)
- muskel (µskeleton lib 2003)

Template vs. macro data flow

M. Danelutto - Tecniche di programmazione avanzata - Corso di dottorato - Pisa - Giu-Lug-07
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

Step 3: farm it
Fixed skeleton set vs. parametric

Fixed set

P3L (the Pisa Parallel Programming Language 1991)
SkIE (Skeleton Integrated Environment 1997)
OcamlP3L (1998)
Macro Data Flow RunTime (1999)
SKElib (SKEleton LIBrary 2000)
Lithium (2000)

Parametric

ASSIST (A Software development System based on Integrated Skeleton Technology 2001)
muskel (μskeleton lib 2003)
Parametric skeletons: the ASSIST way

- Set of virtually parallel activities (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

```c
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] P;
    attribute long B[N][N] scatter B[*ib][*jb] onto P{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 {
        elabl (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 $\rightarrow$ normal form $\rightarrow$ 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
Cole’s manifesto

1. Propagate the concept with minimal disruption
   - No chance to introduce yet another parallel programming language

2. Integrate ad hoc parallelism
   - Specialized, ad hoc solutions must be hosted

3. Accommodate diversity
   - Slightly different skeletons should be derivable

4. Show the payback
   - Advertising: demonstrate that moving to skeletons is worthwhile
Our additions (PARCO 2005)

6. **Support code reuse**
   - Huge amount of (dusty deck?) code
   - Large amounts of (open source) “libraries”

6. **Handle heterogeneity**
   - Cluster/networks/grids *are* heterogeneous
   - Upgrades of clusters (with different release procs and
different amounts/speed of main store)

7. **Handle dynamicity**
   - Non dedicated computing nodes (varying load)
   - Different nodes, different power
Our claim

- Skeleton systems (either skeleton based languages or skeleton libraries) can be classified as “mature, second-generation” if they satisfy the 7 requirements:

1. Minimal disruption
2. Ad hoc parallelism
3. Accommodate diversity
4. Show payback
5. Reuse code
6. Handle heterogeneity
7. Handle dynamicity

M. Danelutto - Tecniche di programmazione avanzata - Corso di dottorato - Pisa - Giu-Lug-07
The muskel way

<table>
<thead>
<tr>
<th></th>
<th>Propagate the concept with minimal disruption</th>
<th>Plain Java library</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Integrate <em>ad hoc</em> parallelism</td>
<td>User access to streams &amp; MDF level</td>
</tr>
<tr>
<td>3</td>
<td>Accommodate diversity</td>
<td>User access to streams &amp; MDF level</td>
</tr>
<tr>
<td>4</td>
<td>Show the pay back</td>
<td>OO + rapid prototyping + efficiency</td>
</tr>
<tr>
<td>5</td>
<td>Support code reuse</td>
<td>Java only</td>
</tr>
<tr>
<td>6</td>
<td>Handle heterogeneity</td>
<td>By Java</td>
</tr>
<tr>
<td>7</td>
<td>Handle dynamicity</td>
<td>Application manager</td>
</tr>
</tbody>
</table>
The ASSIST way

<table>
<thead>
<tr>
<th></th>
<th>Propagate the concept with minimal disruption</th>
<th>... definitely NO!</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Integrate <em>ad hoc</em> parallelism</td>
<td>Parametric parmmod</td>
</tr>
<tr>
<td>3</td>
<td>Accommodate diversity</td>
<td>Parametric parmmod</td>
</tr>
<tr>
<td>4</td>
<td>Show the pay back</td>
<td>Fairly fast application development + high efficiency</td>
</tr>
<tr>
<td>5</td>
<td>Support code reuse</td>
<td>C, C++, Fortran77</td>
</tr>
<tr>
<td>6</td>
<td>Handle heterogeneity</td>
<td>Compiler + run time</td>
</tr>
<tr>
<td>7</td>
<td>Handle dynamicity</td>
<td>Module &amp; application manager</td>
</tr>
</tbody>
</table>
More in general

<table>
<thead>
<tr>
<th></th>
<th>muskel</th>
<th>ASSIST</th>
<th>eSkel</th>
<th>muesli</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propagate the concept with minimal disruption</td>
<td>Plain Java library</td>
<td>...</td>
<td>Plain MPI</td>
<td>Plain C++ &amp; MPI library</td>
</tr>
<tr>
<td>Integrate ad hoc parallelism</td>
<td>User access to streams &amp; MDF level</td>
<td>Parametric parmod</td>
<td>Protected MPI communicators within skeletons</td>
<td>Variety of combinations of (data parallel) skeletons</td>
</tr>
<tr>
<td>Accommodate diversity</td>
<td>User access to streams &amp; MDF level</td>
<td>Parametric parmod</td>
<td>Parametric skeleton calls</td>
<td></td>
</tr>
<tr>
<td>Show the pay back</td>
<td>OO + rapid prototyping + efficiency</td>
<td>Fairly fast application development + high efficiency</td>
<td>Fast application development</td>
<td>OO library expressive power + fast development</td>
</tr>
<tr>
<td>Support code reuse</td>
<td>Java only</td>
<td>C C++ Fortran77</td>
<td>C C++</td>
<td>C C++</td>
</tr>
<tr>
<td>Handle heterogeneity</td>
<td>By Java</td>
<td>Compiler + run time</td>
<td>Guaranteed by MPI</td>
<td>Guaranteed by MPI</td>
</tr>
<tr>
<td>Handle dynamicity</td>
<td>Application manager</td>
<td>Module &amp; application manager</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
III. Advanced features

- Heterogeneity
- Dynamicity/adaptation
- Fault tolerance
- Interoperability
- Hierarchical composition
- Optimizations
- Deployment>Loading
- Security
- Performance
Heterogeneity

P3L (the Pisa Parallel Programming Language 1991)

SkIE
(Skeleton Integrated Environment 1997)

OcamlP3L
(1998)

Macro Data Flow RunTime (1999)

SKELib (SKEleton LIBrary 2000)

ASSIST
(A Software development System based on Integrated Skeleton Technology 2001)

Lithium (2000)

muskel
(μskeleton lib 2003)
Canonical handling

• Supported by virtual machine
  • Java Data + code portability through JVM
  • Web services XML/SOAP/HTML

• Easier to achieve, less performant (but JIT)

• Supported by compiler tools
  • Cross compiling code for different architectures
  • Dynamic linking to libraries (hopefully)
  • XDR needed somewhere (DVSM ?)
  • Loading time decisions about code to load

• Harder to implement, much better performance
Support via Virtual machine

- Sample code: task farm
- Code for master and worker written in Java
- Bytecode dynamically loaded on remote processors
- Data exchange via serialization (internal XDR!)
  - Performance advancement vs. XML based formats
Support via virtual machine (exp)

- Tasks per PE
- Heterogeneous
- Production WS !
- WSx YYY (BogoMIPS)
- Different runs

- muskel (2004)
Support via compiler

• Compile time, step 1:
  • Generation of suitable (high level) intermediate code
    • Data interchange code (marshall/unmarshall)
    • Platform independent comm and sync primitives
  • Generation of cross-makefiles
• Load time:
  • Target node kind
• Compile time, step 2:
  • JIT? ( right after the loading policies eval)
  • Cross compile vs. ssh compile
• Load time, step 2:
  • Stage needed libraries
Support via compiler (2)

Linux

gcc

TCP/IP

different padding schema

gcc

MacOSX

BSD

0x00010000

0x00000001

0x000000001
Support via compiler (exp results)
Support via compiler (exp res 2)

Scalability of the shared memory library

Aggregate bandwidth (MB/s)

Number of servers

homogeneous cluster
heterogeneous cluster
Dynamicity/adaptation

P3L (the Pisa Parallel Programming Language 1991)

SkIE
(Skeleton Integrated Environment 1997)

OcamlP3L
(1998)

Macro Data Flow RunTime (1999)

SKElib (SKEleton LIBrary 2000)

Lithium (2000)

ASSIST
(A Software development System based on Integrated Skeleton Technology 2001)

muskel
(μskeleton lib 2003)
Dynamicity sources

- System dependent
  - Load of single, shared machines
  - Heterogeneous machine collections
    - OS react differently to events
  - Network load (connectivity, bandwidth)
  - (node & link) faults
    - (handled separately fault tolerance)
- Application dependent
  - Hot spots (e.g. stream of “heavy” tasks)
  - Bad, (“system aware”) application coding
Dynamicity effects

- Resource under-utilization
- Resource over-utilization

- Impaired load balancing policies
- Degraded performance
- Degraded efficiency
Dynamicity handling: adaptation

- Several phases
  - Recognize problem
  - Devise a solution
  - Plan execution
  - Commit & exec

- Each with its own implications
  - Each with its own problems/solutions

Diagram:
- Adapt
- Decide
  - Trigger
  - Policy
- Commit
  - Plan
  - Execute
    - Timing
    - Mechanisms

Generic adaptivity aspect
Domain specific
Adaptivity: decide phase

• Triggering application
  • Monitoring
    • Planned at compile time
    • Automatically handled vs. user driven
    • Target architecture proper mechanisms
  • Policy
    • Intervene on the parallelism degree
    • Restructure data distribution
    • ...
    • Possibly supported by performance models
Adaptivity: commit phase

• Make a plan
  • Current computation involved as less as possible
  • Exploit structured parallel programming model features to intervene in the right points
  • Plan a complete set of actions
• Commit
  • Decide which mechanism
  • Apply them according to the plan
Adaptivity: case study (decide)

- Data parallel skeleton
- Initial partition of data across PEs
  - Load balancing achieved
- Several PEs overloaded (different user burst)
- Option 1): look for other PEs, recruit and redistributed
- Option 2): redistribute

- Performance contract driven (!)
Performance contract

• User supplied (top level skeleton/application)

• Several kind of:
  • Service time
  • Completion time
  • Deadline
  • QoS

• Sub-contracts derived for application sub-components
  • Relies on performance models
### Adaptivity: case study (commit)

<table>
<thead>
<tr>
<th>Action 1</th>
<th>Action 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embarrassingly parallel data parallel skeleton</td>
<td>Data parallel skeleton with stencil</td>
</tr>
<tr>
<td>Stop a process with excess load</td>
<td>Wait for barrier</td>
</tr>
<tr>
<td>Restart with smaller data partition item</td>
<td>Stop all process involved</td>
</tr>
<tr>
<td>Stop process with low load</td>
<td>Exchange data</td>
</tr>
<tr>
<td>Move data &amp; restart</td>
<td>Restart processes</td>
</tr>
</tbody>
</table>
Adaptivity: experiments (ASSIST)


**Graph:**
- Y-axis: Time (secs)
- Bar graph showing time for different numbers of PEs (Processor Elements).
- X-axis: Stream items computed (computation unfolding).
- Legend:
  - Current Service Time
  - Average Service Time

- Key points:
  - Performance contract
  - Max. Service Time

M. Danelutto - Tecniche di programmazione avanzata - Corso di dottorato - Pisa - Giu-Lug-07
Adaptivity: experiments (ASSIST)

PEs overload caused by the activation of other applications. The AM reacts by re-distributing data and computation onto available PEs.

Performance contract max. Service Time

Current Service Time

Average Service Time

+2 PEs

2 PEs

4 PEs
(no more PEs are available)
Adaptivity: experiments (muskel)

M. Danelutto, *QoS in parallel programming through application managers*
Euromicro Conference on Parallel, Distributed and Network-based processing, Lugano, Feb. 2005
Adaptivity: ASSIST

- Each parmod has its own *manager*
  - Takes care of optimizing performance
- Parmods in a generic graph
  - Graph *manager* optimizes overall computation
Implementation

Aldinucci et al. *Dynamic reconfiguration of grid-aware applications in ASSIST*, Europar 2005
Adaptivity: recent results (ASSIST)

Aldinucci et al. *Dynamic reconfiguration of grid-aware applications in ASSIST*, Europar 2005
Adaptivity: recent results (JJPF)

Dazzi et al. *A Java/Jini framework supporting stream parallel computations*, PARCO 2005
Adaptivity: recent results (JJPF)

Dazzi et al. *A Java/Jini framework supporting stream parallel computations*, PARCO 2005
Fault tolerance

• Key issue
  • Long running jobs
    • What if power elapses just before getting the result?
  • Grid target architectures
    • What if a network link or a processing element “dies”?

• Several approaches possible
  • Don’t care
  • Checkpoint & restart
  • Exploit structured model features
Fault tolerance: don’t care

- Former approach in structured parallel programming
  - Functionality and performance as the goals
- Large computations “batch style”
  - Users can tolerate some missing results (looks like …)
- Useless approach in serious computer science
Fault tolerance: checkpoints

- Programmer inserts checkpoints vs. automatic checkpoint instrumentation
- Local checkpointing vs. remote
- Restarting mechanisms
  - Synchronization among the parallel components
- Definitely needed for long running applications
  - Weeks to months lasting applications
- Checkpointing must take into account messages
  - Already sent, already received, just sent
Fault tolerance: macro data flow

- Macro data flow from skeleton programs
- Fireable instructions
  - Checkpoint units
  - Scheduled to a node for execution
    - Monitored
  - In case of failure, rescheduled
- Grain of computation \(\Rightarrow\) fault tolerance efficacy
Fault tolerance: muskel
Interoperability

P3L (the Pisa Parallel Programming Language 1991)

SkIE
(Skeleton Integrated Environment 1997)

OcamlP3L
(1998)

Macro Data Flow RunTime (1999)

SKElib (SKEleton LIBrary 2000)

Lithium (2000)

ASSIST
(A Software development System based on Integrated Skeleton Technology 2001)

muskel
(μskeleton lib 2003)
Interoperability

• Further aspect of code reusage
  • plain code reusage \(\rightarrow\) kernel/application reusage

• Usually to/from a framework
  • E.g. interoperability with Web Services
    • Export application as a Web Service
      (WSDL, Tomcat & co)
    • Use Web Services in your applications (import)

• Also to support structured parallel programming
  • HOC
    • WS used to provide code parameters to skeletons
Interoperability: the wrapper way

- Compile structured application
- Generate wrapping code
  - Outer interface: the one of the target framework
  - Inner interface: the one of the application
- Compile and deploy wrapping code
- Wrap external framework code in proper application framework invocation
Sample wrapping

• Java code (application framework)
• Generate WSDL descriptor
• Link to Axis
• Set up Tomcat server

• In a single process
  • Link a SOAP library
  • Issue a call to the remote service
  • Unmarshall answer back to Java
CORBA-ASSIST interoperability

Wrapping of ASSIST code in a CCM component

Bertolli, Coppola et al. *Integration of ASSIST parallel programs within the CCM Framework* 2005
Support to structured parallel

- Skeleton based parallel programming
- Components as skeleton parameters
- Components wrapped in Web Services

- SOAP firewall safe
- Java virtual machine heterogeneity handling
- WS repository (outer interoperability)
Support to structured parallel

J. Duennweber et al. *Behaviour Customization of Parallel Components For Grid Application Programming*, submitted 2005
Hierarchical composition

• Debated item since the very beginning of skeletons/design patterns
• Included in some frameworks (a few)
  • P3L was the first one
  • Muesli/muskel/ASSIST are the more recent examples
• Recently re-newed interest from other contexts
  • Components: a composition of components is a component?
Pros

• Basic idea
  • Provide in a (skeleton, pattern, component) composition the same interface of the composite

• Sample
  • Y = farm(seq2)
  • X = pipe(Seq1, Y, Seq3)

• *Ex pluribus unum*
  • Abstraction
  • Expressive power
  • Code reusage / interoperability
Cons

- Implementation level consequences
  - Need fairly well structured implementation design
  - Optimized mechanism for composition
  - Effect on run time optimizations
    - Pipeline manager requests should overwrite single stage manager requests
- Analytical models implications
  - Fairly simple analytical model possible for single patterns
  - Rather complicate (intractable) to take into account composition
• Assume
  pipe(farm(F), farm(G))
• Farm model
  match input task rate with service task rate
• Bottom up
  farm(F) ⇒ pardegree(3) T_s=10msec
  farm(G) ⇒ pardegree(5) T_s=5msec
  pipe(…) T_s=10msec
• Top down
  As above + re-ask farm(G) for 10msec
  farm(G) ⇒ pardegree(2) T_s=10msec
  pipe(…) T_s=10msec with less PEs
Optimizations

• Always claimed as the (one of the) main reason(s) to use structured parallel programming
  • Exposing parallel structure ⇒ 1) clear places where and 2) clear ways how to intervene

• Most relevant difference with unstructured parallel programming models:
  • Analysis phase needed, then synthesis of optimization
  • Non trivial analysis phase, usually trace or monitoring driven
Sample optimizations

- Compile time optimizations
  - Template selection
  - Process template reduction
  - Source2source (skeleton normal form)

- Run time optimizations
  - Communication clustering
  - Shared, read only, data caching
Template selection (muskel)
Template selection (ASSIST)
Process template reduction

- Service/wrappers processes per template
  - E.g. input/output managers in ASSIST
  - Master process in master/slave
- Usually composition can eliminate process redundancy
Process template reduction

- Service/wrappers processes per template
  - E.g. input/output managers in ASSIST
  - Master process in master/slave
- Usually composition can eliminate process redundancy

![Diagram of process template reduction]
Source to source transforms

- Normal form
  (stream parallel skeleton composition)
  - Take seq frontier and farm it (equivalent, better $T_s$)

- Backus (‘78! Turing award lecture)
  - $(\alpha f) \circ (\alpha g) \equiv \alpha (f \circ g)$
  - Means:
**Unfolding Rules**

1. seq \( f : \tau ^{f} \xrightarrow{\ell} \) seq \( f : \tau ^{f} \)
2. farm \( \Delta : \tau ^{\ell} \xrightarrow{\ell} \) farm \( \Delta : \tau ^{\ell} \)
3. pipe \( \Delta \Delta : \tau ^{\ell} \xrightarrow{\ell} \) pipe \( \Delta \Delta : \tau ^{\ell} \)
4. comp \( \Delta \Delta : \tau ^{\ell} \xrightarrow{\ell} \) comp \( \Delta \Delta : \tau ^{\ell} \)
5. map \( p^{-1} \Delta : \tau ^{\ell} \xrightarrow{\ell} \) map \( p^{-1} \Delta : \tau ^{\ell} \)
6. d&c \( p^{-1} \Delta : \tau ^{\ell} \xrightarrow{\ell} \) d&c \( p^{-1} \Delta : \tau ^{\ell} \)
7. while \( t \Delta : \tau ^{\ell} \xrightarrow{\ell} \) while \( t \Delta : \tau ^{\ell} \)

** Exec Rules**

1. seq \( f : \tau ^{f} \xrightarrow{\ell} f : \tau ^{f} \)
2. farm \( \Delta : \tau ^{\ell} \xrightarrow{\ell} \Delta : \tau ^{\ell} \)
3. pipe \( \Delta \Delta : \tau ^{\ell} \xrightarrow{\ell} \Delta \Delta : \tau ^{\ell} \)
4. comp \( \Delta \Delta : \tau ^{\ell} \xrightarrow{\ell} \Delta \Delta : \tau ^{\ell} \)
5. map \( p^{-1} \Delta : \tau ^{\ell} \xrightarrow{\ell} p^{-1} : \Delta \)
6. d&c \( p^{-1} \Delta : \tau ^{\ell} \xrightarrow{\ell} d&c \( p^{-1} \Delta : \tau ^{\ell} \)
7. while \( t \Delta : \tau ^{\ell} \xrightarrow{\ell} \) while \( t \Delta : \tau ^{\ell} \)

Can be used to formally derive

\[ \text{farm} \Delta \equiv \Delta' \]

as well as to evaluate different parallel executions.
Communication clustering

- Sample case:
  - Data parallel computation
  - Logically high number of parallel activities
  - Multiple parallel activities on the same PE
  - Need to communicate shared data after each iteration
- Distribution analysis
  - Infer partitions to be communicated
- Virtual process code
  - Issues a single send/receive
- Support
  - Groups all the comms towards the same node (latency improvement)
Communication with futures

- Original Lithium pipe$(f,g)(x)$ computation (left)
- Futures introduced when communicating with the remote workers (right)
- Problem: tradeoff between communication opt and task prefetch overhead (run time + endoff phase)
Communication with futures (2)

Overloading the most powerful machine leads to higher utilization of the other CPUs

Number of threads on the overloaded machine
Communication with futures (3)

Load balancing OK
Larger PE#  simpler policy OK

Aldinucci, Duennweber et al. *Optimization Techniques for Implementing Parallel Skeletons in Distributed Environments*, CoreGRID TR-0001, 2005
Caching

- Matrix multiplication code
  - Cartesian of rows A and columns B
  - Each item to IP
  - Collect results
- Each item to IP (remote worker):
  - Send “IP(ref(A_i),ref(B_j))”
    - cache miss then ask for A_i
    - cache hit (don’t ask for B_j)
    - compute and send back C_{ij}
Deploying/loading facilities

- Interpreter based environments
  - Interpreter deployed once and forever
    - Needs authentication (VO)
  - Tasks/duties deployed from time to time
    - Online (on the fly, in demand)
    - Offline (prefetch, lookahead)
- Compiler based environments
  - Code deployed from time to time (staging)
  - Authentication needed (VO)
  - Co-allocation needed
  - Manage needed libraries/machine configurations
Muskel deploying/loading

- RMI object placed on machines once and for all
  - By hand/script/ssh
  - Under rmid control
- Ssh + authentication needed
- Then tasks via RMI
  - Functional (task computation)
  - Non functional (monitoring, statistics, management)
- RMI subsystem ports needed
  - ProActive RMI ssh tunnelling
ASSIST deployment/loading

- Object code on the start machine
- Configuration files
- Libraries
- Staging of the needed software via GEA (ex ASAP)
  - AAR files (JAR with ASSIST specific components)
  - adHOC stream implementation framework incldued
- Remote commanding via SSH/Globus
- Port range open needed for adHOC streams
- Large amount of libraries moved (every time)
  - Portability at the price of startup time …
Deployment/loading time

- Is it part of completion time ($T_c$)?
  - Yes
    - Lower speedups
    - Better chance to interpreter based approaches
  - No
    - Higher speedups
    - Better chance to optimize code at compile time

- Still an open question
- We always take away init time from $T_c$
Security

• Big issue
  • Can user X ask for my machine cycles?
    Should I guarantee access to my data to user X?
  • Is my data/code staged elsewhere secure?
  • Can I be sure no interfering communications happens, modifying my program/data?
• Authentication, confidentiality, non repudiation …
• Well known approaches to handle them, but …
But

- E.g. session key in PKI
- Very good to exchange spot data
  - Time to break too long with respect to data validity
- What about long running, distributed computations?
  - Mechanisms needed to take care of time (WEP vs WPA analogy)
  - Dedicated PKI infrastructures
    - Access peculiar, not usage
    - Globus installation ...
Security: implications

- Implementation
  - sXX protocols, at minimum
  - Crypted communications cost in performance terms!
    (try an scp -r vs a tar file transfer)

- Design
  - Design proper, non excessive security levels
  - Implement proper, non intrusive security policies
Security: implications

Baraglia et al. *HPC APPLICATION EXECUTION ON GRIDS*, Dagstuhl NGG’05, Springer Verlag
Security: exploit structuring

• Interpreter based approaches
  • Use Secure Socket Layer API
    e.g. secure RMI  secure data/code staging
• Authentication based interpreter interaction
  • Authenticated code load
• Sandbox / access security policy
  • Rely on abstract machine implementation
Performance models

- Needed to support
  - Compile time optimizations
  - Run time policies
- Precision $\propto$ effectiveness
- Quite simple for simple parallel patterns
- Quite complex for complex, highly dynamic and composite patterns
- Several approaches followed in the past:
  - Analytical
  - Operational research
  - Queue theory
Analytical models

- $T_s(\text{pipe}) = \max_i \{T_s(\text{stage}_i)\}$
- $T_s(\text{farm}) = \max \{T_s(\text{in}), T_s(\text{out}), T_s(\text{worker})\}$
- $NW_{\text{farm}} = \left\lfloor \frac{T_s(\text{worker})}{\max \{T_s(\text{in}), T_s(\text{out})\}} \right\rfloor$

- Working for the simple cases
- Subject to run time refinement
- Difficult to calculate both compute intensive and communication service times
Operational research models

- Given constrains
  - Service time of one stage non bigger than the one of next pipeline stage
  - Worker string service time bigger/equal to the interarrival time
  - Number of resources used less than those available

- Minimise resources used
- Minimize service time

- Requires integer programming techniques
Notable application

• P3L: optimal resource usage
• Traverse tree root to bottom giving the best number of resources
• Iterate:
  • Take away 1 resource from one node
  • Propagate balancing in the three
  • Bottom up

• Then
  • Integer programming system
  • Gave (slightly) better solutions
    • Same service time with less nodes
    • Better service time with the same amount of node
Queue theory

- Template process network studied as a queue system
- Non trivial solution for fairly simple systems
  - Non composite (not too much!)
- Trivial solutions for complex systems
- [Zoc05] PhD thesis showed better solutions:
  - System of differential equations modeling
    - Steady state
    - Dynamic behaviour
Performance

• Performance *ab initio*
• Deep implications on heterogeneity, adaptivity
• Alternative choices affect and are affected by:
  • virtual machine vs compile time heterogeneity handling
  • fault tolerance with automatic checkpoint vs. user handled
  • XML based XDR vs. binary ones
• Fundamental: integrated approach
  • Muesli vs. muskel
  • C++/MPI vs. Java/RMI
• No adaptivity vs. contracts + fault tolerance
• Performance comparable in heterogeneous (!) environments
How to take perf. into account (2)

- Tradeoff among
  - Compile time
  - Load time
  - Run time
- Compile time
  - Once & for all, cannot be changed at run time (!)
- Load time
  - Target machine dependent decisions, migration (?)
- Run time
  - Better adaptivity, depends on reaction times and stability of the system