Structured parallel programming

M. Danelutto

Dept. Computer Science - Univ. Pisa & CoreGRID Programming model Institute
Contents

• Structured parallel programming (vs. unstructured/traditional)

• Algorithmic skeletons (parallel design patterns)

• Advanced features heterogeneous & dynamic architecture targeting, adaptation, autonomic behaviour

• Perspective & open problems
Structured parallel programming
vs. non structured/traditional
Thesis

Parallel/Distributed Programming

Functional concerns

System engineer Responsibilities

Non functional concerns

Programmer Responsibilities

Efficiency, portability, adaptation, autonomic control, ...
Non functional concerns

- design parallel activities (threads? processes?)
- map & schedule parallel activities (resource discovery? mapping strategies? scheduling policies?)
- communications & synchronization ((a)synch? (a)symmetric? data sharing? consistency? collectives?)
- load balancing (heterogeneous hw/sw? (non)dedicated nodes?)
- fault tolerance ((dis)appearing resources? connectivity? )
Functional concerns

• What has to be computed
  • the function
  • the algorithm
  • the data
  • types of input and output data
E.g. task farm

- Task farm, embarrassingly parallel computation, master/worker, ...

- Probably the more frequent, recognized pattern in parallel and grid computing

  - Parallel activities: how many workers?

  - Mapping: were can I place workers?

  - Scheduling: round robin? auto scheduling?
Task farm (2)

- Communication: TCP/IP? RMI/RPC? secure?
- Load balancing: job stealing? auto scheduling? static vs. dynamic?
- Fault tolerance: checkpointing? task buffering?
- ...
Task farm (2)

• Communication: TCP/IP? RMI/RPC? secure?
• Load balancing: job stealing? auto scheduling? static vs. dynamic?
• Fault tolerance: checkpointing? task buffering?
• ...

how much time did we spend just to mention the problems to be solved ???
Separation of concerns

- Task farm: functional concerns
  - function computed by workers
  - input & output data handling
- Task farm: non functional concerns
  - all the rest ...
Task farm as pattern

```java
public interface FarmWorker<Task, Result> {
  public Result compute(Task task);
}

FarmWorker<Image, Image> myWorker = new ImageBlur(...);
TaskFarm<Image, Image> myFarm = new TaskFarm(myworker);

myFarm.process(fileIn, fileOut);
```
Algorithmic skeletons
The concept

- 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)
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
User perspective

• Design application as composition of parallel building blocks

• Provide code parameters (functional) to customize building blocks

• Ask the system to compute the program
Bits of history

Cole PhD (1988)
Fixed degree DC, Iterative combination, Cluster Task queue

Darlington (1992)
Pipeline, Farm, RaMP, DMPA

P3L (1991)
Pipeline, Farm, Map, Reduce

BMF (‘80)
map fold reduce prefix + algebra

Skillicorn (mid ‘90)

Kuchen Skil (1998)

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

SCL
Fortran S

ASSIST
Lithium
OcamlP3L

eSkel (2002)
Parametric skeletons + Give/Take

Gorlatch (late ‘90)
HOC (early ‘00)

Serot (1999)
Skipper (→MDF)

MALLBA (‘00)
Combinatorial optimisation

HOC (early ‘00)

Gorlatch (late ‘90)

BMF (‘80)
map fold reduce prefix + algebra

Muesli (2002)
Pipeline, Farm, Parallel array + collectives
... in Pisa

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)
Layered approach
Layered approach

- Applications
- Programming language / model
- Compiler tools
- Intermediate code + run time
- OS + Middleware
- Target hw
Template based
Macro Data Flow

Pipeline main (...) stage1(...) stage2(...) stage3(...) end pipeline
seq stage1(...)
{ ... }
farm stage2(...) seq2(...)
end farm
farm stage3(...) seq3(...)
end farm

Source code

Simbolic tree

Instantiation of graph with input tasks

Data flow graph

Data flow instruction pool

Deployment

Target architecture
Muesli

• Kuchen, Univ. of Muenster
• C++ MPI library
• Nestable (two tier, à la P3L) stream parallel and data parallel skeletons
• Targets MPI clusters and networks
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 (2)

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;};
}
Muskel

• Full Java (RMI based) skeleton library

• Stream parallel skeletons only (Lithium has data parallel skeletons too)

• Introduces the concept of Manager

  • takes care of performance contracts and fault tolerance

• Macro Data flow implementation

• Introduces skeleton set expandability
Muskel
public static void main (String [] args) {
    ...
    Skeleton s1 = new PreProcessInput(...);
    Skeleton w  = new ParameterSweep(...);
    Skeleton s2 = new Farm(w);
    Skeleton s3 = new PostProcessResult(...);
    Skeleton p  = new Pipe(s1,new Pipe(s2,s3));

    Manager mgr = new Manager();
    mgr.setContract(new ParDegree(Integer.parseInt(args[0])));
    InputStreamManager ism =
                            new InputStreamManager("in.dat");
    OutputStreamManager osm =
                            new OutputStreamManager("out.dat");
    mgr.compute(ism,osm);

    return;
}
ASSIST

- ASSIST program = generic graph of modules:
  - sequential (C, C++, F77) or parallel (parmod)

- Data flow streams among module + non deterministic stream control (input)

- parmod: parametric parallel module
  - collection of virtual processors
  - arranged in a topology
  - possibly synchronized and/or sharing data
  - processing items from input streams
  - delivering data to output streams
ASSIST (2)

- targeting TCP/IP networks/clusters/grids as well as Globus grids
- components
- autonomic managers for notable composite components (farm, data parallel, pipeline)
- designed within GRID.it
  several groups working on SAR image processing, computational chemistry, image rendering, bioinformatics, ...
ASSIST framework
Sample code (2)

```c++
/* definition of the application module graph */

generic main()
{
    stream double[N][N] A;
    stream double[N][N] res;

    firstStage (output_stream A);
    secondStage (input_stream A output_stream res);
    endStage (input_stream res);
}

/* definition of sequential modules */

firstStage (output_stream double A[N][N]) {
    read_from_disk (output_stream A);
}

proc proc_first (output_stream double A[N][N])
inc "fstream", "iostream", "string"
{
    // C++ code reading tmpA from disk here ...
    assist_out(A, tmpA); // then output tmpA on stream
}

parmod secondStage (input_stream double A[N][N]
    output_stream double risultato[N][N]) {

    topology array [i:N] P;
    attribute double S[N][N] scatter S[*i0][] onto P[i0];
    attribute bool diff replicated;
    stream double ris[N];

    do input_section {
        guard1: on , , A {
            distribution A[*k0][] scatter to P[k0];
        } while (true)
    }

    virtual_processors {
    compute_secondStage (in guard1) {
        VP i=0 {
            init(in A[i1][] out S[i1][]);
            sync;
            do {
                // do nothing
            } while (reduce (diff, ||) == true);
            assist_out (ris, S[i1][]);
        }
        VP i=N-1 {
            init(in A[i1][] out S[i1][]);
            sync;
            do {
                // do nothing
            } while (reduce (diff, ||) == true);
            assist_out (ris, S[i1][]);
        }
        VP i=1..N-2 {
            init(in A[i1][] out S[i1][]);
            sync;
            do {
                computeStencil (in S[i1][], S[i-1][], S[i+1][]
                    out S[i1][], diff);
            } while (reduce (diff, ||) == true);
            assist_out (ris, S[i1][]);
        }
    }
}

output_section {
    collects ris from ALL P[i1] {
        static double risult[P[i1][N][N];
        const double *el;
        AST_FOR_EACH(el) {
            for(int j=0; j<N; ++j) risult[i1][j]=el[j];
            assist_out(res, risult);
        }<>
    }
}
```
The skeleton advantage
Scalability

- Muskel
- Computational grain is the key factor
- $G = \frac{T_w}{T_c}$
  - $T_w$ compute time
  - $T_c$ comm time (in&out)
- With medium to coarse grain efficiency is near 99%
Heterogeneous hw

- Muskel
- Differently configured workstations (WSname Bogomips)
- Execute different number of tasks
  - adaptation...
Fault tolerance

- Muskel
- replace faulty nodes (rebooted, network unreachable, ...)
- reschedule aborted tasks
- recruit new PEs
  - adaptation ...
Template selection

- Muskel
- Different macro data flow scheduling policies affect performance
- Policy chosen w.r.t. input task set features
- Adaptation ...
Template selection (2)

- ASSIST
- parmod template: static (template) vs. macro data flow (dynamic)
- MPI implementation compared to plain TCP/IP

- on non homogeneous tasks (5% penalty MDF vs. static template in case of homogeneous tasks)
Security handling

- Muskel
- Set of processing nodes on secure network + set on “public” network
- Code & data confidentiality must be guaranteed
- Deployment: system responsibility
- adaptation ...
Optimizations

• Normal form (stream parallel skeleton composition)

• Take seq frontier and farm it (equivalent, better Ts)

• Backus ('78! Turing award lecture)

• \((\alpha f) \circ (\alpha g) \equiv \alpha(f \circ g)\)

• Means:
Advanced features

Adaptation

Autonomic management
Adaptation

• Already experimented/seen
  • heterogeneous hw targeting, load balancing, security management

• in all those cases
  • ad hoc policies
  • on top of existing mechanisms
E.g. fault tolerance (Muskel)

```java
while (taskPool.anyMore()) {
    Task task = (Task) taskPool.getItem();
    if (task != null) {
        nTasks++;
        Object result = null;
        long elapsed = 0L;
        try {
            result = worker.execute(task);
        } catch (RemoteException e1) {
            taskPool.addItem(task);
            // then terminate the thread
            workerManager.died();
            return; // thread terminates
        }
    }
}
```
while (taskPool.anyMore()) {
    Task task = (Task) taskPool.getItem();
    if (task != null) {
        nTasks++;
        Object result = null;
        long elapsed = 0L;
        try {
            result = worker.execute(task);
        }
        catch (RemoteException e1) {
            taskPool.addItem(task);
            // then terminate the thread
            workerManager.died();
            return; // thread terminates
        }
    }
}
Autonomic computing

1) An autonomic computing system needs to "know itself" - its components must also possess a system identity. Since a "system" can exist at many levels, an autonomic system will need detailed knowledge of its components, current status, ultimate capacity, and all connections to other systems to govern itself. It will need to know the extent of its "owned" resources, those it can borrow or lend, and those that can be shared or should be isolated.

2) An autonomic computing system must configure and reconfigure itself under varying (and in the future, even unpredictable) conditions. System configuration or "setup" must occur automatically, as well as dynamic adjustments to that configuration to best handle changing environments.

3) An autonomic computing system never settles for the status quo - it always looks for ways to optimize its workings. It will monitor its constituent parts and fine-tune workflow to achieve predetermined system goals.

4) An autonomic computing system must perform something akin to healing - it must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction. It must be able to discover problems or potential problems, then find an alternate way of using resources or reconfiguring the system to keep functioning smoothly.

5) A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection. It must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.

6) An autonomic computing system must know its environment and the context surrounding its activity, and act accordingly. It will find and generate rules for how best to interact with neighboring systems. It will tap available resources, even negotiate the use by other systems of its underutilized elements, changing both itself and its environment in the process -- in a word, adapting.

7) An autonomic computing system cannot exist in a hermetic environment. While independent in its ability to manage itself, it must function in a heterogeneous world and implement open standards -- in other words, an autonomic computing system cannot, by definition, be a proprietary solution.

8) An autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden. It must marshal I/T resources to shrink the gap between the business or personal goals of the user, and the I/T implementation necessary to achieve those goals -- without involving the user in that implementation.
Autonomic computing

1- KNOW ITSELF
2- SELF-CONFIGURE
3- SELF-MANAGEMENT (CONTINUOUS IMPROVEMENT)
4- SELF_HEALING
5- SELF-PROTECTION
6- CONTEXT AWARENESS
7- OPERATE IN HETEROGENEOUS ENVIRONMENT
8- ANTICIPATE NEEDS WITHOUT INTRODUCING COMPLEXITY
Adaptation (revisited)
Skeleton pros

- **monitoring**: system concern, deployed on need
- **triggering**: exploits template structure
- **analyze and plan**: exploit complete knowledge on the parallel structured of the application
- **execute**: exploit complete knowledge about implementation structure + template mechanisms
ASSIST results

- 2 PEs
- 3 PEs
- 4 PEs
- 5 PEs
- 6 PEs
- 7 PEs (no more PEs are needed)

Time (secs)

100 110 120 130 140 150 160 170 180 190 200 210 220 230

Current Service Time
Average Service Time

Stream items computed (computation unfolding)

performance contract max. Service Time

+1 PE
ASSIST results (2)

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

Stream items computed (computation unfolding)
CoreGRID

- Programming model Institute activities
- GCM : Grid Component Model
  - Hierarchical composition
    Collective+data+stream interaction patterns
    Autonomic management of notable composite components
    XML based ADL
- Reference implementation on ProActive within GridCOMP
Behavourial skeletons

• **Key idea:**
  Combine functional and non functional concerns in a skeleton

• **Behavourial skeleton:**
  algorithmic skeleton (class) + autonomic manager
  • designed and implemented by system programmers
  • instantiated (possibly specialized) by application programmers
Sample BS

- Component (GCM)
- Autonomic Controller
  - passive part
    (monitor, mechanism)
- Autonomic Manager
  - active part
    (analyse, plan)
- Functional skeleton
  - farm, map, MISD, ...
    (depending on S and C)
AM cycle

1. AM asks AC measures
2. AM triggers a policy and designs a plan
3. AM commands AC to execute the plan
4. then loops
User perspective

- Plain component
- Functional port (provides/server):
  0) submit a task to be computed
- Non functional ports (provides/server):
  1) set performance contract
  2) activate autonomic management

Submit(Task x)
Pros

• Autonomic management hides ALL non functional aspects

• User (application programmers) concentrate on functional concerns

• Possibly

  • non functional ports to supply new AM policies
“Scientist” perspective

• implement a suitable Task data type
• provide a suitable Contract
  • then loop:
    • fetch data (e.g. from disk) into a Task
    • add callback to store results
  • submit it
“Propagate the concept with minimal disruption” (Cole’05)

- SCA / Tuscany implementation of a Task Farm BS with autonomic manager
- AM exploits JBoss rules
  - precondition from AC bean
  - actions with AC bean mechanisms
- Completely transparent service
- Java client works out of the box (with WSDL)
AM policies

- Rules sent to the AM as TEXT through non functional interfaces

- rule "AdaptUsageFactor"
  when $workerBean:
    WorkpoolBean(serviceTime > 0.25)
    then
    $workerBean.addWorkerToNode(""");
  end
Experiments

- Measured (coarse grain)
- Ideal (coarse grain)
- Measured (fine grain)
- Ideal (fine grain)

The graph shows the completion time (seconds) for different numbers of workers. The x-axis represents the number of workers, and the y-axis represents the completion time in seconds.
Experiments (2)

Double worker no as a consequence of service time increase (after 1/2 tasks computed)
Perspectives &
open problems
Hierarchic AMs

• Structured applications require structuring of AM decisions
  • to avoid conflicts and “local” minima
• Inner AMs passive with respect to outer ones
  • but can be turned to active in case of “best effort” requirements
The problems

- “split” (user) contract into inner component subcontracts
- precise contracts vs. “best effort” ones
- contract violations handling local vs. non local
The problems

- “split” (user) contract into inner component subcontracts
- precise contracts vs. “best effort” ones
- contract violations handling local vs. non local
The problems

• “split” (user) contract into inner component subcontracts
• precise contracts vs. “best effort” ones
• contract violations handling local vs. non local
The problems

- “split” (user) contract into inner component subcontracts
- precise contracts vs. “best effort” ones
- contract violations handling local vs. non local
The problems

- “split” (user) contract into inner component subcontracts
- precise contracts vs. “best effort” ones
- contract violations handling local vs. non local
The problems

• “split” (user) contract into inner component subcontracts

• precise contracts vs. “best effort” ones

• contract violations handling local vs. non local
The problems

- “split” (user) contract into inner component subcontracts
- precise contracts vs. “best effort” ones
- contract violations handling local vs. non local
The problems

- “split” (user) contract into inner component subcontracts
- precise contracts vs. “best effort” ones
- contract violations handling local vs. non local
Contracts

• general formalism needed
  • XML based proposal from IBM
• capable of expressing
  • performance contracts, (boolean) quality of service contracts, resource contracts, ...
• which logic to handle contracts?
  • first order? temporal? ...
Conclusions
Conclusions

- Structuring
- “raises the level of abstraction” presented to programmers
  - sensibly shortens design+implementation times
  - improves efficiency of tools
- gives the opportunity to implement effective autonomic management policies
Perspective
Perspective

- in the near future there will be no "non autonomic/adaptive" program ...

- more or less as today you NEVER configure a printer when visiting a CS Department ...
Thank you for your attention

Any questions?

slides @ www.di.unipi.it/~marcod/belfast08
email: marcod@di.unipi.it
Any questions?

slides @ www.di.unipi.it/~marcod/belfast08
email: marcod@di.unipi.it