Perspectives in parallel programming

MARCO DANELUTTO
PHD COURSE
MAY 2015
Non functional concerns

Performance
○ Latency, throughput

Security
○ Data, code

Fault tolerance

Power management
Management of NF concern

Business logic code

NF concern policies

Parallel patterns

Parallel application code
User defined Service level agreement

Sample
- Maximum throughput
- Minimal latency
- Throughput = 25 frames/second
- Cypher all data
- Secret code
- Tolerate processor faults
Multiple NF concern management

Power and performance

Power
  ◦ Frequency throttling
  ◦ Coprocessors

Performance
  ◦ Parallelism degree
  ◦ Frequency throttling
  ◦ Coprocessors
Performance (throughput)

IF (not enough) AND (may increase par degree) THEN increase par degree

IF (too much) THEN decrease par degree

Sample: farm (OK), pipeline (OK?), seq (???)
Mechanisms: performance

Parallelism degree

Coprocessors
- Data parallel (GPU), data flow (FPGA), GP (many core)
- Impact on scheduling
- Resource recruitment
- Contention vs. pre-allocation
Power

IF (too much) THEN reduce freq
IF (too much) THEN move comp to accelerator
IF (too much) THEN reduce par degree
IF (low) THEN ???
Mechanisms: power management

Frequency throttling
  ◦ Power drained prop to square of freq
  ◦ #instructions/sec prop to frequency
  ◦ DVFS (Dynamic Voltage Frequency Scaling)

Parallelism degree (=> #instructions)

Resource switch off/on
  ◦ power PC per core on/off
Security

Application data
- Encrypt/decrypt via standard algorithms
- Adds time (overhead)
- Adds space (comm time, storage)

Application code
- Same as data
- Plus code decryption at untrusted sites
- Obfuscation, encrypted execution

Abstract

We present secret execution in which an encrypted program is evaluated without decryption, to give an encrypted result whose decryption yields the original result.

1 Introduction

Can we execute an encrypted program without decryption? In the cloud computing era, more people want to execute their programs in cloud servers. The biggest challenge for delegating program execution is security—how to keep the programs confidential? One possible solution is protecting the programs via obfuscation (e.g., virtual machine-based obfuscation) [4, 15, 3, 14]. Unfortunately, however, this protection is not perfect; one can decode the obfuscation—it's only a matter of time. Fortunately, however, cryptologist has already researched similar problem for decades and recently proposed a solution: homomorphic encryption—basis of our work. Homomorphic encryption is an encryption scheme that preserves certain operations on encrypted data. A homomorphic encryption $E$ is said to preserve an operation $op$ if it provides $op$, an encrypted version of $op$, such that for a plain text $m$,

$E(op(m)) = E(op(E(m)))$
```c
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
define f (e-g-h)
define j (e*e-g)
define k (j-h)
define (x) tab2[x]/h
#define m(n,a) ((n&\(a\))==\(a\))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L }; int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 }; main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s; if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); } else switch(m1-=h){
    case f:
        a=(b=(c=(d=g)<<g)<<g)<<g; return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
    case h:
        for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
    case g:
        if(n<h)return(g); if(n<j){n-=g;c='D';o[f]=h;o[g]=f;} else{c=''-'';n-=j-g;o[f]=o[g]=g;} if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-g]+c; return(o[b-g]%n+k-h);
    default:
        if(m1==e) main(m1-g+e+h,s+g); else *(s+g)=f;
        for(*s=a=f,a<e; *s=(*s<<e)|main(h+a++,(char *)m1);
    }
} main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); } else switch(m1-=h){
        case f:
            a=(b=(c=(d=g)<<g)<<g)<<g; return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
        case h:
            for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
        case g:
            if(n<h)return(g); if(n<j){n-=g;c='D';o[f]=h;o[g]=f;} else{c=''-'';n-=j-g;o[f]=o[g]=g;} if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-g]+c; return(o[b-g]%n+k-h);
        default:
            if(m1==e) main(m1-g+e+h,s+g); else *(s+g)=f;
            for(*s=a=f,a<e; *s=(*s<<e)|main(h+a++,(char *)m1);
            }
```
Fault tolerance

Classical approach (seq)

- Periodic checkpointing
- Rollback to last recovered state
- Requires observer
- Adds overhead
- Adds memory (disk) requirements
Parallel code checkpointing
Parallel code checkpointing (2)
Fault tolerance (proactive)

Upon fault detection
- Keep computing
- Re-directing the computation to a correct state
- Possibly through uncorrect state traversal
1. What is "fault tolerance"?

The phrase "fault tolerance" means many things to many people. Typical definitions range from user processes dumping vital state to disk periodically to checkpoint/restart of running processes to elaborate recreate-process-state-from-incremental-pieces schemes to ... (you get the idea).

In the scope of Open MPI, we typically define "fault tolerance" to mean the ability to recover from one or more component failures in a well defined manner with either a transparent or application-directed mechanism. Component failures may exhibit themselves as a corrupted transmission over a faulty network interface or the failure of one or more serial or parallel processes due to a processor or node failure. Open MPI strives to provide the application with a consistent system view while still providing a production quality, high performance implementation.

Yes, that's pretty much as all-inclusive as possible -- intentionally so! Remember that in addition to being a production-quality MPI implementation, Open MPI is also a vehicle for research. So while some forms of "fault tolerance" are more widely accepted and used, others are certainly of valid academic interest.

2. What fault tolerance techniques does Open MPI plan on supporting?

Open MPI plans on supporting the following fault tolerance techniques:

- Coordinated and uncoordinated process checkpoint and restart. Similar to those implemented in LAM/MPI and MPICH-V, respectively.
- Message logging techniques. Similar to those implemented in MPICH-V
- Data Reliability and network fault tolerance. Similar to those implemented in LA-MPI
- User directed, and communicator driven fault tolerance. Similar to those implemented in FT-MPI.

The Open MPI team will not limit their fault tolerance techniques to those mentioned above, but intend on extending beyond them in the future.

3. Does Open MPI support checkpoint and restart of parallel jobs (similar to LAM/MPI)?

Yes. The v1.3 series was the first release series of Open MPI to include support for the transparent, coordinated checkpointing and restarting of MPI processes (similar to LAM/MPI).

Open MPI supports both the BLCR checkpoint/restart system and a "self" checkpointer that allows applications to perform their own checkpoint/restart functionality while taking advantage of the Open MPI checkpoint/restart infrastructure. For both of these, Open MPI provides a coordinated checkpoint/restart protocol and integration with a variety of network interconnects including shared memory, Ethernet, InfiniBand, and Myrinet.

The implementation introduces a series of new frameworks and components designed to support a variety of checkpoint and restart techniques. This allows us to support the methods described above (application-directed, BLCR, etc.) as well as other kinds of checkpoint/restart systems (e.g., Condor, libckpt) and protocols (e.g., uncoordinated, message induced).
Behavioural skeleton

Behavioural skeleton (BS) concept

- Co-design of
  - parallelism exploitation pattern (skeleton)
    - known way of exploiting parallel machines
  - autonomic management of non-functional features
    - control loop based management, ensuring user defined QoS contract

- BS shields application programmer from the environment
CoreGRID

This web site is discontinued and the information is being maintained for archive/historical purposes. It will not be updated.

The CoreGrid project ended in October 2008

For information about the ERCIM Working Group "Grids, P2P and Services"
(frederic.desprez@inria.fr)

The European Research Network on Foundations, Software Infrastructures and Applications

The CoreGRID Network of Excellence (NoE) aims at strengthening and advancing scientific and technological research. To achieve this objective, the Network brings together a critical mass of well-established researchers who have constructed an ambitious joint programme of activities. This joint programme of activities is based on the basis of their strategic importance, their research challenges and the recognition of European context.

- knowledge & data management;
- programming models;
- architectural issues: scalability, dependability, adaptability;
- Grid information, resource and workflow monitoring services;
- resource management and scheduling;
- Grid systems, tools and environments.

Grid Programming with Components

An Advanced Component Platform for an Effective Invisible Grid

End results

Extended Data Record Processing

WingDesign

Biometric identification system

The Days Sales Outstanding

Non-Functional Component Features

Vision

More about Convergence between Grid Computing and SOA

Record Processing

Virtual Services

The Days Sales Outstanding
BS: autonomic management

- Control loop based
  - monitor: perceive system current behaviour
  - analyse: compare current behaviour with the expected one
    - needs abstract model of well behaving system
  - plan: devise a strategy to handle malfunctioning (if any)
  - execute: actuate the chosen plan → sequence of actions

- Rule based
  - pre-condition → action rules
  - analyse ⇒ lookup of fireable rules in the DB (those with pre-cond true)
  - plan, execute ⇒ apply action rules
Behavioral skeleton concept

- Complete separation of concerns
  - System programmers
    - parallelism exploitation pattern
    - autonomic management
  - Application programmers
    - “best” BS
    - business application logic
    - QoS contract (rules) → BS AM

Diagram:
- Behavioral skeleton
  - Instantiation
  - Compile
  - Working application
  - Algorithmic skeleton
  - Autonomic manager
  - Business logic params
Behavioural skeleton (history)

- Introduced in GCM
  - The Grid Component Model by CoreGRID:
    1. Hierarchical components,
    2. With collective, stream and event ports
    3. With autonomic managers taking care of non-functional features
  - First implementation within GridCOMP
    ProActive/GCM reference implementation (http://gridcomp.ercim.org)
    → Task farm & data parallel BS
    → With autonomic management of performance
  - Validate through GridCOMP industrial use cases
    - Biometric identification systems
    - Electronic record processing
    - Wing design code
Hierarchical management

- Computations as BS compositions
  - e.g. BS task farm worker in a BS pipeline

- Autonomic managers
  - react to local triggers
  - using local rules
  - lack of "global view" (and decisions!)

- Need for hierarchical management
  - provide "global framework" for non-local optimized decisions
  - at different levels → each root node provides a global decision framework for its leaves nodes
  - alternatives:
    → decisions taken at the root node or
    → "driven" by root node at leaf nodes
Hierarchical management: new issues

1. **Contract propagation**

2. Local violation of contract with no local alternatives

3. Phases (different rôles w.r.t. management)
   - **User contract**
     - directed to root manager
   - **Actual need**
     - one contract per manager
     - contract “propagation” through the manager tree
       - contract may or may not change while propagating
       - kind of propagation depends on the type of the node propagating the contract
Hierarchical management: new issues

1. Contract propagation
2. Local violation of contract with no local alternatives
3. Phases (different rôles w.r.t. management)
   - Local violation of contract
     - with no local alternatives
     - concept of violation raised to upper level manager
   - Moving responsibilities to the higher level
     - with global view
   - While raising violation
     - keep maintaining the current state
     - and keep raising violation to upper level
Hierarchical management: new issues

1. Contract propagation
2. Local violation of contract with no local alternatives
3. Phases (different rôles w.r.t. management)
   - **AM phases**
     - *different rôles w.r.t. management*
     - *active*
       - taking local decisions
     - *passive*
       - local “standby”
       - waiting for new strategies (new rules to be used in the AM)
### BS component implementation

- **Composite component → component → black box**
- **Functional interface**
  - Set worker
    - component actually computing a single task
    - multiple instances organized within the BS
  - Compute tasks
    - stream or RPC port
- **Non-Functional interface**
  - Set contract
    - determine the intended behaviour w.r.t. a given non-functional concern
  - Report behaviour
    - query current and actual status of the computation
BS component implementation

- **Composite component** → **component** → **black box**

- **Functional interface**
  - Set worker
    - *component actually computing a single task*
    - *multiple instances organized within the BS*
  - Compute tasks
    - *stream or RPC port*

- **Non-Functional interface**
  - Set contract
    - *determine the intended behaviour w.r.t. a given non-functional concern*
  - Report behaviour
    - *query current and actual status of the computation*
AM implementation

- AM implemented on top of JBoss Drools
  - beans implementing behaviour
    - monitoring
    - actuator actions

- Sample rules
  - performance management

```
rule "CheckRateLow"
  when
    $departureBean : DepartureRateBean( value < ManagersConstants.FARM_LOW_PERF_LEVEL )
    $arrivalBean : ArrivalRateBean( value >= ManagersConstants.FARM_LOW_PERF_LEVEL )
    $parDegree: NumWorkerBean(value <= ManagersConstants.FARM_MAX_NUM_WORKERS)
  then
    $departureBean.setData(ManagersConstants.FARM_ADD_WORKERS);
    $departureBean.fireOperation(ManagerOperation.ADD_EXECUTOR);
    $departureBean.fireOperation(ManagerOperation.BALANCE_LOAD);
end
```
Hierarchical management with rules

- **Raising violations**
  - `$actuatorBean.fireOperation(ManagerOperation.RAISE_VIOLATION);`

- **Contract propagation**
  - `$actuatorBean.fireOperation(ManagerOperation.PUSH_LIST);`
    - reconsult(file) or change bean values (constants) at the target AM

- **Active / passive phases**
  - **active:**
    - *one or more rules with pre-condition holding true and non-violation actions*
  - **passive:**
    - *rule(s) with pre-condition holding true and violation raising actions*
Experimental results

- **Sample application in GCM**
  - three stage pipeline with parallel second stage
  - rendering of mammography series to evidence possible cancer sites

- **First stage: Sequential code**
  - produce image stream (may vary image rate, if requested)

- **Second stage: Task farm**
  - process images (may vary parallelism degree (self-decision) or raise violations (no locally effective rule fireable)

- **Pipeline**
  - deals with violations raise from task farm
  - monitors pipeline stage balancing
Multiple NF concern management

- Independent development of managers for NF concern $NFC_a$ and $NFC_b$
- Coordination needed when both concerns are taken into account
  - to avoid conflicting decisions
  - to stabilize configuration
    (avoid loops back and forth from the same configurations)
- Classical solutions in the past:
  - linear combination of changes required by different managers
    - only with “continuous” concerns
Super manager
● Coordinated AMs
Possible strategy:

- each decision
- agreed among AMs before committing
- then
  - committed (possibly modified)
  - or aborted

1. analyze current status & plan some corrective action
2. broadcast decision to all AMs
3. collect answers
4. execute action (all ACK) or abort it (1 NACK) or modify plan (provide PropX)
5. send ACK/NACK/provide(PropX)
6. evaluate request
• Minimum common ground to coordinate manager activity
AM actions

- **non interfering**
  - action a planned by $AM_X$ does not interfere with policies of any other manager $AM_Y$

- **interfering**
  - action a planned by $AM_X$ has consequences on policies managed by at least one other manager $AM_Y$

- **corrective action**
  - action $a'$ (instead of an equivalent effect) by $AM_X$ taking into account $AM_Y$ policies
Consensus building

- AM(X) proposes decision $d$
- AM(Y) answers $a_y \in \{\text{ACK}, \text{needProperty}(P), \text{NACK}\}$
  - $\textbf{NACK} \in \{a_p, a_s, a_E, \ldots\}$
    - $d$ is aborted, AM(X) priority of $d$ is lowered
  - $\text{needProperty}(P) \in \{a_s, a_E, \ldots\}$
    - plan for $d$ modified according to $P$ (if possible)
  - $\forall a_x \in \{a_s, a_E, \ldots\}$ is an ACK
    - $d$ plan applied as prospected
Priority of concerns

- user defined SLAs ($\text{SLA}_X$ directed to $\text{AM}(X)$)
- order defines priority
  - first SLA determines the init AM (hierarchy)
  - the one in charge of defining the initial process graph
- e.g.
  - $\text{SLA}_E, \text{SLA}_P \Rightarrow \text{few slow processors}$
  - $\text{SLA}_P, \text{SLA}_E \Rightarrow \text{many fast processors}$
Sample case study

- **BS**: Farm
- **NFC**: Performance and Security
- **AMP** actions
  - *add worker* (to hopefully increase performance)
  - *remove worker* (if not needed due to external constraints)
- **AMS** actions
  - ask to *secure new worker* on non secure node
### Performance AM rules

<table>
<thead>
<tr>
<th>Name</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Farm\textsubscript{inc} | \( \text{priority}(x), \) \( \text{instanceof(farm)} \) \& \( T_{arr} > QoS \) \& \( \text{Throughput} < QoS \)  
\quad \rightarrow \text{findNewResource, allocateNewWorker, connectNewWorker} |
| Farm\textsubscript{dec} | \( \text{priority}(x), \) \( \text{instanceof(farm)} \) \& \( \text{Throughput} >> QoS \)  
\quad \rightarrow \text{removeWorker}                                                                                     |
# Consensus rules

<table>
<thead>
<tr>
<th>Name</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Farm\_inc\_PH1 | priority(x),  
instanceof(farm) \& \ T\_arr > QoS \& Throughput < QoS  
→ findNewResource, askConsensus(G', R') |
| Farm\_inc\_PH2 | priority(x),  
ackFromAll → allocateNewWorker, connectWorker |
| Farm\_inc\_PH2 | priority(x),  
ackFromAll\&needProperty(security)  
→ allocateNewWorker, connectSSLWorker |
| Farm\_inc\_PH2 | priority(x),  
nackConsensus → lowerPriority(Farm\_inc) |
| Farm\_dec   | priority(x),  
instanceof(farm) \& Throughput >> QoS  
→ removeWorker |
Security AM rules (consensus rules)

<table>
<thead>
<tr>
<th>Name</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Node\textsubscript{new} | \texttt{priority}(y),
|            | \texttt{consensusAsked}(G') \& \texttt{diff}(G\textsubscript{current}, G') = N
|            | \& \texttt{nonSecure}(N) \rightarrow \texttt{answer}(\texttt{needProperty}(security)) |
| Node\textsubscript{new} | \texttt{priority}(y),
|            | \texttt{consensusAsked}(G') \& \texttt{diff}(G\textsubscript{current}, G') = N
|            | \& \texttt{secure}(N) \rightarrow \texttt{answer}(ACK) |
Sample run
LIBERO: a framework for autonomic management of multiple non-functional concerns

M. Aldinucci°, M. Danelutto*, P. Kilpatrick†, V. Xhagija*

° University of Torino – * University of Pisa – † Queen’s University Belfast

Abstract. We describe a lightweight prototype framework (LIBERO) designed for experimentation with behavioural skeletons—components implementing a well-known parallelism exploitation pattern and a rule-based autonomic manager taking care of some non-functional feature related to pattern computation. LIBERO supports multiple autonomic managers within the same behavioural skeleton, each taking care of a different non-functional concern. We introduce LIBERO—built on plain Java and JBoss—and discuss how multiple managers may be coordinated to achieve a common goal using a two-phase coordination protocol developed in earlier work. We present experimental results that demonstrate how the prototype may be used to investigate autonomic management of multiple, independent concerns.