Actors, Active Objects, Components for Massive Distributed and Parallel Multi-Core Applications

Denis.Caromel@inria.fr

1. Background: INRIA, OASIS, ActiveEon
2. Parallel Programming:
   a) Asynchronous Active Objects and Futures
   b) Groups and SPMD
   c) Parallel Components
   d) Skeleton
   e) Deployment (that scales)
3. Small Video
1. Background and Team at INRIA
OASIS Team & INRIA

A joint team between: INRIA, Nice Univ. CNRS
• Participation to EU projects:
  – CoreGrid, EchoGrid, Bionets, SOA4ALL
  – GridCOMP (Scientific Coordinator)

• ProActive Parallel Suite
  Distributed and Parallel:
  From Multi-cores to Enterprise+Science GRIDs

Computer Science and Control
• 8 Centers all over France
• Workforce: 3 800
• Strong in standardization committees:
  – IETF, W3C, ETSI, …
• Strong Industrial Partnerships
• Foster company foundation:
  90 startups so far
  - Ilog (Nasdaq, Euronext)
  - …
  - ActiveEon
Startup Company Born of INRIA
Co-developing, Providing support for Open Source ProActive Parallel Suite
Worldwide Customers (EU, Boston USA, etc.)
2. Programming Models
Distributed and Parallel
Active Objects
**ProActive** : Active objects

- `A ag = newActive ("A", [...], VirtualNode)`
- `V v1 = ag.foo (param);`
- `V v2 = ag.bar (param);`
- `...`
- `v1.bar(); //Wait-By-Necessity`

Wait-By-Necessity is a Dataflow Synchronization
First-Class Futures

Update
Wait-By-Necessity: First Class Futures

Futures are Global Single-Assignment Variables

\[ V = \text{b.bar}() \]

\[ \text{c.gee}(V) \]
Wait-By-Necessity: Eager Forward Based

AO forwarding a future: will have to forward its value

\[
V = b.\text{bar}() \\
c.\text{gee}(V)
\]
Wait-By-Necessity: Eager Message Based

AO receiving a future: send a message

\[ V = b.\text{bar}() \]
\[ c.\text{gee}(V) \]
Standard system at Runtime:
No Sharing

NoC: Network On Chip

Proofs of Determinism
\[
\begin{align*}
(a, \sigma) & \rightarrow_\mathcal{S} (a', \sigma') \\
\alpha[a; \sigma; i; F; R; f] \parallel P & \rightarrow \alpha[a'; \sigma'; i; F; R; f] \parallel P \\
\end{align*}
\]

\text{LOCAL}

\[
\begin{align*}
\gamma \text{ fresh activity} & \quad i' \not\in \text{dom}(\sigma) & \sigma' = \{i' \mapsto AO(\gamma)\} :: \sigma \\
\sigma_\gamma = \text{copy}(\iota'', \sigma) & \quad \text{Service} = \langle \text{if } m_j = \emptyset \text{ then FifoService else } i''.m_j() \rangle \\
\alpha[\mathcal{R}[\text{Active}(\iota'', m_j)]; \sigma; i; F; R; f] \parallel P & \rightarrow \alpha[\mathcal{R}[\iota']; \sigma'; i; F; R; f] \parallel \gamma[\text{Service}; \sigma_\gamma; \iota''; \emptyset; \emptyset; \emptyset] \parallel P \\
\end{align*}
\]

\text{NEWACT}

\[
\begin{align*}
\sigma_\alpha(i) = AO(\beta) & \quad i'' \not\in \text{dom}(\sigma_\beta) & f_i^{\alpha \rightarrow \beta} \text{ new future} & \quad i_f \not\in \text{dom}(\sigma_\alpha) \\
\sigma_\beta' = \text{Copy&Merge}(\sigma_\alpha, i'; \sigma_\beta, i'') & \quad \sigma'_\alpha = \{i_f \mapsto \text{fut}(f_i^{\alpha \rightarrow \beta})\} :: \sigma_\alpha \\
\alpha[\mathcal{R}[i.m_j(i')]; \sigma_\alpha; i'; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; i_\beta; F_\beta; R_\beta; f_\beta] \parallel P & \rightarrow \alpha[\mathcal{R}[i_f; \sigma'_\alpha; i'; F_\alpha; R_\alpha; f_\alpha]] \parallel \beta[a_\beta; \sigma'_\beta; i_\beta; F_\beta; R_\beta :: [m_j; i''; f_i^{\alpha \rightarrow \beta}; f_\beta]] \parallel P \\
R = R' :: [m_j; i'; f'] :: R'' & \quad m_j \in M & \quad \forall m \in M, m \notin R' \\
\alpha[\mathcal{R}[\text{Serve}(M)]; \sigma; i; F; R; f] \parallel P & \rightarrow \alpha[i.m_j(i') \uparrow f, \mathcal{R}[]]; \sigma; i; F; R' :: R''; f'] \parallel P \\
\end{align*}
\]

\text{REQUEST}

\[
\begin{align*}
\iota' \not\in \text{dom}(\sigma) & \quad F'' = F :: \{f \mapsto i'\} & \sigma' = \text{Copy&Merge}(\sigma, i' ; \sigma, i') \\
\alpha[i \uparrow (f', a); \sigma; i; F; R; f'] \parallel P & \rightarrow \alpha[a; \sigma'; i; F'; R; f'] \parallel P \\
\end{align*}
\]

\text{ENDSERVICE}

\[
\begin{align*}
\sigma_\alpha(i) = \text{fut}(f_i^{\gamma \rightarrow \beta}) & \quad F_\beta(f_i^{\gamma \rightarrow \beta}) = i_f & \quad \sigma'_\alpha = \text{Copy&Merge}(\sigma_\beta, i_f ; \sigma_\alpha, i) \\
\alpha[a_\alpha; \sigma_\alpha; i_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; i_\beta; F_\beta; R_\beta; f_\beta] \parallel P & \rightarrow \alpha[a_\alpha; \sigma'_\alpha; i_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; i_\beta; F_\beta; R_\beta; f_\beta] \parallel P \\
\end{align*}
\]

\text{REPLY}
TYPED

ASYNCHRONOUS

GROUPS
Creating AO and Groups

- A ag = newActiveGroup ("A", [...], VirtualNode)
- V v = ag.foo(param);
- ...

v.bar(); //Wait-by-necessity

Typed Group Java or Active Object
Group, Type, and Asynchrony are crucial for Composition
Broadcast and Scatter

Broadcast is the default behavior
Use a group as parameter, Scattered depends on rankings

```java
ag.bar(cg);   // broadcast cg
ProActive.setScatterGroup(cg);
ag.bar(cg);   // scatter cg
```
Object-Oriented
SPMD
OO SPMD

- A ag = newSPMDGroup ("A", [...], VirtualNode)
  // In each member
- myGroup.barrier ("2D"); // Global Barrier
- myGroup.barrier ("vertical"); // Any Barrier
- myGroup.barrier ("north","south","east","west");

Still, not based on raw messages, but Typed Method Calls ==> Components
MPI Communication primitives

For some (historical) reasons, MPI has many communication primitives:

- MPI_Send: Std
- MPI_Ssend: Synchronous
- MPI_Bsend: Buffer
- MPI_Rsend: Ready
- MPI_Isend: Immediate, async/future
- MPI_Ibsend, ...

I’d rather put the burden on the implementation, not the Programmers!

How to do adaptive implementation in that context?

Not talking about:
- the combinatory that occurs between send and receive
- the semantic problems that occur in distributed implementations
Application Semantics rather than Low Level Optimization

MPI:
- MPI_Send
- MPI_Recv
- MPI_Ssend
- MPI_Irecv
- MPI_Bsend
- MPI_Rsend
- MPI_Isend
- MPI_Ibsend

What we propose:
High-level Information from Application Programmer
(Experimented on 3D ElectroMagnetism, and Nasa Benchmarks)

Example:
ro.foo (ForgetOnSend (params))

⇒ Both Distributed &
⇒ Multi-Core Optimizations
Skeleton
Algorithmic Skeletons for Parallelism

High Level Programming Model [Cole89]
Hides the complexity of parallel/distributed programming
Exploits nestable parallelism patterns
Algorithmic Skeletons for Parallelism

public boolean condition(BlastParams param) {  
    File file = param.dbFile;
    return file.length() > param.maxDBSize;
}

Skeleton Program

Task
- farm
- while
- pipe
- if
- for

Data
- divide & conquer
- fork
- map

divide & conquer ($f_b$, $f_d$, $f_c$)
- pipe
- fork
- seq($f_3$)
- seq($f_1$)
- seq($f_2$)
Parallel Components
GridCOMP Partners
Objects to Distributed Components

IoC: Inversion Of Control (set in XML)

Example of component instance

Typed Group

Java or Active Object

JVM

Truly Distributed Components
Scopes and Objectives:

Grid Codes that Compose and Deploy

No programming, No Scripting, No Pain

Innovation:

Abstract Deployment

Composite Components

Multicast and GatherCast
Optimizing MxN Operations

2+ composites can be involved in the Gather-multicast
Deployment
Deploy on Various Kinds of Infrastructures

The Cloud

Utility Computing

What

End Users See
P2P: Programming Models on Overlay Networks
Current Open Source Tools:

**PROGRAMMING**
- Java Parallel Frameworks for HPC, Multi-Cores, Distribution, Enterprise Grids and Clouds.

**OPTIMIZING**
- Eclipse GUI (IC2D) for Developing, Debugging, Optimizing your parallel applications.

**SCHEDULING**

Acceleration Toolkit: Concurrency + Parallelism + Distributed
Pies for Analysis and Optimization
Video 1: IC2D Optimizing Monitoring, Debugging, Optimizing
Summary
Perspectives for TERACOMP

• A need for several, coherent, Programming Models for different applications:
  – Actors (Functional Parallelism) + Active Objects
  – NEW SPMD: away from low-level optimizations
  – Parallel Component: Codes and Synchronizations:
    • Parallel and Multi-Core Components
    • MultiCast & GatherCast: Capturing // Behavior at Interfaces!
  – Adaptive Parallel Skeletons
  – Event Processing (Reactive Systems)

• Efficient Implementations are needed to prove Ideas!
• Proofs of Programming Model Properties are Needed for Scalability!

Our Current Record: 4 000 Cores all over the world at once
Our TERACOMP Target:
10 000 to 100 000 Cores POC Applications with
“Declarative, High-level” programming
Previous Questions

People ready to learn new environment?

⇒ My answer: YES and ONLY if they have a ROI !!!
   Performance and/or Productivity!