ASSIST
High-performance Programming Environment: Application Experiences and Grid Evolution

Marco Vanneschi
Department of Computer Science, University of Pisa

EuroPVM/MPI 2003, Venice
ASSIST

(A Software development System based on Integrated Skeleton Technology)

Department of Computer Science, University of Pisa

ASSIST

A Programming Environment for High-performance Portable Applications on Clusters, Large-scale Platforms and Grids

Projects:

• ASI-PQE2000
• CNR Agenzia 2000
• MIUR-CNR Strategic Programme L449/97, 1999 and 2000
• MIUR-FIRB Grid.it

Implementations:

• Cluster/Beowulf (on top of ACE)
• First Grid version – AssistConf (on top of Globus)
• On-going: High-performance Component ASSIST

ASSIST as a research vehicle

- Significant improvements for cluster architectures
- Feasible and successful approach for:
  - Computational Chemistry, and other scientific codes,
  - Image & Signal Processing,
  - Earth Observation Systems,
  - Video Compression,
  - Knowledge Discovery and Data Mining, User Profiling,
  - Search Processing on Structured / Unstructured Data,
  - Query Language Interpreters, …

Can it be a feasible approach for Large-scale and Grid platforms too?
Outline

1. Structured Parallel Programming: ASSIST as an improvement \textit{wrt}
   “classical” skeletons

2. Flexible implementation model

3. Towards Grid programming
Structured Parallel Programming

ASSIST as an improvement *wrt* “classical” skeletons
Structured Parallel Programming

Parallel program

Pipeline main
farm stage1
farm stage2
End pipe

pipeline
forall
farm
scan
Structured Parallel Programming

- High-level constructs for task parallelism (e.g. PIPELINE, FARM), data parallelism (e.g. MAP, SCAN, STENCILS), mixed task+data parallelism (D&C, PARMOD), and their compositions (GENERIC or STRUCTURED GRAPHS)

- Semantic model and associated performance model
  - constraints on the parallel paradigm adopted to compose (sequential / parallel) modules into complex applications

- Many potentialities for intensive optimizations and restructuring of applications
Structured Parallel Programming

- Approaches to Structured Parallel Programming:
  - Parallel Skeletons model
  - Parallel Design Patterns
  - …

- Overcoming the difficulties of traditional data parallel languages (HPF) and their evolutions

- Our past experience (Univ. Pisa): skeletons-based coordination languages
  - **P3L** (1991), C-based, fixed skeleton set: pipe, map …
  - **SkIE** (1997), C/C++/F77/Java
  - **Lithium** (2001), Java-based, macro data-flow, pipe, farm, map, D&C
  - Several variants of them
Structured parallel programming and performance models

Example: Farm / Master-Slave / Parameter Sweeping / …

Load-balanced execution of Tasks belonging to a Stream

Efficient and parametric implementation templates for platform- and application-dependent optimizations

Input Stream

Output Stream

Emitter: Task Scheduling

Collector: Set of functionally identical Workers

Optimal number of workers and other performance parameters (e.g. throughput, efficiency) can be expressed as functions of processing times, communication times, and utilization factors

ASSIST - Marco Vanneschi - EuroPVM/MPI 2003, Venice
Several **pros**: easy programmability, rapid prototyping, sequential software reuse, efficiency
- mainly for regular applications and/or regular compositions

**Cons**: for complex compositions, and for some irregular and dynamic applications
- Lack of expressiveness / inefficiency
- Lack of flexibility
  - Any modification led to extensive changes within compiler & run-time support

**Optimizations**:
- not so intensive at compile time as it was expected,
- very significant at the **run-time** support level,
  - also for **dynamic** approaches to the run-time design
ASSIST: general program structures

• **Classical skeletons:** often, *fixed-patterns* program structures are too simple for complex applications

• **ASSIST:** parallel programs represented as *generic graphs*
  • whose nodes are structured
  • and can share objects
Simple composition of fixed-patterns (stream parallel: pipeline, farm skeletons)

Example: a simple Ray Tracer

Streams of scenes
Parallelism among scenes

Stream of input scenes

Stream of output scenes

Stage 1

Stage 2 (farm)
Rendering algorithm

Stage 3
Composition of stream + data parallelism

Example: a more powerful Ray Tracer

Parallelism among scenes and inside every single scene

Stream of input scenes

Stream of output scenes

stage 1

stage 2 (farm + map)

rendering algorithm

stage 3
Sequential modules
- written in several host languages (C, C++, Fortran, Java)

Arbitrary Composition  
- stream-oriented
- both data-flow and nondeterministic with internal state

Not only fixed-pattern Parallel Skeletons ...
ASSIST graphs and Shared Objects

- Global variables
- Shared memory
- Files and I/O
- Libraries
- CORBA, DCOM, ...
- ASSIST modules
- ...

Composition by
- Typed streams
- External objects

Parallel (or sequential) module

Output stream

Composition by
- Typed streams
- External objects
Generic graphs: data-flow + nondeterminisms

Acyclic precedence graph (DAG) of components with data-flow behaviour

Stream-based, possibly cyclic graph of components: data-flow and/or nondeterministic behaviour

Stream-based computations are more general and possess interesting features of complex applications (e.g. data management, servers)

Nondeterminism + state is a powerful feature \textit{wrt} purely functional (e.g. data-flow) behaviour
Parallel Module (parmod): a paradigm for structured parallelism

- Instead of specific skeletons:
  - a GENERIC SKELETON
    - i.e. a structure that can be effectively specialized at every utilization
- The parmod construct includes the classical (stream-parallel and data-parallel) skeletons as special cases …,
- … but it aims to achieve much more expressive power.
- In addition, parmod expresses parallel computations with state, nondeterminism, and access to external shared objects.
The \texttt{parmod} construct

Multiple input and output typed data streams

Set of Virtual Processors (VP) executing user code

VPs have assigned topology for naming (one, none, arrays)
The \texttt{parmod} construct

independent distribution and collection strategies
(e.g. broadcast, multicast, scatter, \texttt{on-demand})

input and output sections can also host arbitrary user code
The *parmod* construct

VPs host several user functions, activation can be data-driven
(CSP-like nondeterministic execution, guarded channels)

VPs share data structures (run-time provides consistency)

Partitioning rules, replication
Efficient run-time support of \texttt{parmod}

- One of the main advantages of structured parallel programming is the opportunity for efficient run-time implementation of “specific” skeletons.

- ASSIST has proved that this is true for “generic” skeletons too: \texttt{parmod} performance is
  - comparable to that of the same programs written in MPI,
  - comparable to, or better than, that of the same programs expressed by “specific” skeletons
  - More difficult implementation
Parallel Partitioned Apriori (Data Mining)

- Mainly stream-parallel
- Computation intensive, well balanced
- dataset > 160 Mb
- regular I/O pattern

- 8 x Pentium 4, Gbit Eth

Performance Benchmarks of parmod (efficient as MPI or classical skeletons)
Apriori algorithm (data mining) as a pipeline of parmod-farms ("none" topology)

1. Database reading, generation of stream of partitions
2. Apriori algorithm in parallel (load balanced farm)
3. Combination of partial results: collapsing hash-tree data structures
4. Database scan, generation of a new stream of partitions of appropriate size
5. Computation of "support" of the candidate solution (farm with broadcast)
Performance Benchmarks of \texttt{parmod} (efficient as MPI, better than classical skeletons)

Data-Parallel Benchmark

Variable Stencil – single \texttt{parmod}

- 2-D matrix 400x400
- partitioned row-wise
- communication stencil varies at each step
  \[
  \text{for } h \ldots \\
  \text{forall } i, j \ldots
  \]

- 8 x Pentium 4, Gbit Eth

\begin{figure}
\centering
\includegraphics[width=\textwidth]{chart}
\caption{Data parallel speed-up}
\end{figure}
An irregular-dynamic benchmark
(much better than classical skeletons)

N-body, Burnes-Hut

- Parmod implementing a “specialized farm”, with shared memory objects
- Plummer Model, very irregular data-set
- 8 x Pentium 4, Gbit Eth

N-body speed-up, size 1000K

N. of Processors

Ideal  Measured
Complex parallel programs in ASSIST

- Complex applications, frameworks and/or critical cases for compositions

- Intimate mix of task + data parallelism:
  - Systolic computations (single parmod, task + data parallelism)
  - Classification, clustering algorithms (graph of parmods)
  - User profiling by data mining (graph of parmods)
  - Language interpreters – Knowledge discovery in semi-structured datasets (graph of parmods)

- Parallel external objects
  - Data repositories
  - Web caching
  - Interfaces for legacy SW
Example: data-mining C4.5 as a parallel D&C

\{ training set TS, decision tree \( \Gamma \) \}

For load balancing:
- during some phases: *Divide* works in a data-parallel manner,
- in other phases: in a farm-like manner,
- in other phases ...

Shared Tree objects exploited efficiently

ASSIST - Marco Vanneschi - EuroPVM/MPI 2003, Venice
A user-profiling framework

Control and tuning
Knowledge Repository (XML)
Interface CRM-DB -> DR

Visualize

Data Repository (parallel file system)

Clust Classif Assoc Selection Feedback

Interface CRM-DB -> DR
Interface DR -> CRM-DB

CRM-DB (Oracle)

SAIB project: MIUR L46
SEMA Schlumberger, Univ.Pisa, Poly.Turin
External objects: a necessary feature for flexibility

- Interactive applications
- Objects reuse with primitive APIs
- Devices, files, Parallel File System
- Data repositories
- Shared memory objects
- ASSIST programs themselves

• Composition by streams only is not sufficient
• Towards Component ASSIST
Example: data-mining C4.5 as a parallel D&C

\{ training set TS, decision tree $\Gamma$ \}

Client → Divide → Conquer → Test

Shared Tree objects exploited efficiently

\begin{itemize}
  \item \textbf{Client}:
  \item \textbf{Conquer}:
  \item \textbf{Test}:
\end{itemize}
Integration with CORBA Code

- N-body simulation
- GUI CORBA server
- Parallel client

**Structure of the ASSIST program**

- Sequential code
- Initial data
- Compute (ParMod)
- Loop control
- Simulation results

**Client side**
- CorBA interface

**Server side**
- CorBA interface

**Grafical interface**

- N-body simulation
- GUI CORBA server
- Parallel client
Part 2

ASSIST
Flexible Implementation Model
ASSIST implementation [EuroPar2003, ParCo2003]

Run-time support for cluster architectures: on top of ACE library and distributed shared memory (DVSA)
Design patterns based

ASSIST program parco.ast

> astcc parco.ast

ASSIST compiler

C++, Makefile

XML conf

ASSIST - Marco Vanneschi - EuroPVM/MPI 2003, Venice
Experimenting with extensions

1. Targeting heterogeneous COWs

2. Integrating parallel MPI libraries

3. AssistConf and ASSIST-G: first ASSIST Grid version on top of Globus
Targeting heterogeneous COWs
Just enrich the code factory
Add parallel MPI libraries
Just enrich the module factory

façade
- front-end factory
- module factory
- config factory
- code factory

ASSIST compiler

MPI builder

Assist program parco.ast

XML conf

C++

Config. builder

Code builder

Parser typecheck

Module builder
MPI integration

parmod

parmod MPI

parmod

MPI wrapper

parmod

parmod

parmod

VP VP VP
VP VP VP
VP VP VP

0 1 2
3 ...

n

VP VP VP
VP VP VP
VP VP VP

0 1 3
2 ...

n

VP VP VP
VP VP VP
VP VP VP

0 1 2
3 ...

n
AssistConf and ASSIST-G: a first Grid implementation on top of Globus

**ASSIST compiler**

- façade
  - front-end factory
  - module factory
  - config factory
  - code factory

**ASSIST compiler**

- XML conf
- C++
- Config. builder
- Code builder

**ASSIST compiler**

- Parser typecheck
- Module builder
- Code factory
- Config factory
- Module factory

**ASSIST compiler**

- parco.ast
- Assist program
- ASSIST parser
XML configuration file

- modules list (parallel activities)
- modules graph
- pathnames, lib-names, code-names
- lib-modules bindings
- machine names
- modules parallel degrees
- modules-machines mapping

static

dynamic
Just enrich the config factory

ASSIST compiler

façade
- front-end factory
- module factory
- config factory
- code factory

GRID conf

Assist program
parco.ast

XML conf

C++

Config. builder

Code builder

Parser typecheck

Module builder
ASSIST-G

ASSIST compiler

XMLconf (static) → ASSISTconf

XMLconf (static)

ASSISTconf

broker

gather & reservation
resources

MDS
GRAM
DUROC

resources requirements

lib staging allocation

CLAM

CLAM

GRIS/GIIS

MDS
Part 3

ASSIST

Towards Grid programming
Grid.it Project
Enabling Platforms for High-performance Computational Grids Oriented to Scalable Virtual Organizations

- MIUR – FIRB and CNR
  - CNR, INFN, ASI, CNIT, Universities
- Basic Research Programme - ICT
  - + infrastructure and demonstrators (25%)
- Timeframe: November 2002 – October 2005
- Total Cost: 11 M€
  - other synergies by MIUR-CNR Projects on Complex Enabling Platforms: 2,5 M€
Software technology of Grid.it

Domain-specific Problem Solving Environments (PSEs)

High-level services
Knowledge services, Data bases, Scientific libraries, Image processing, …

High-performance, Grid-aware component-based programming model and tools

Programming Environment

Resource management, Performance tools, Security, VO, …

Next Generation Middleware

Basic infrastructure - standards (OGSA-compliant)
Critical research issues

- Dealing with heterogeneity
- New compilers, run-time supports, resource management
- Secure and fault tolerant implementations
- Dynamic, adaptive applications
- Implementing requirements for Quality of Service

Focus of this Part

- Principles
- Personal ideas
Notable reference: GrADS Project

- Concept of reconfigurable program
  - High-level formalism
  - High-level information on application requirements
  - Components technology and composition of applications
  - Performance model (“negotiation” at run-time)

- Application manager:
  - set of static and dynamic tools that control all the development-execution cycle of the application (including dynamic restructuring)
Grid.it:
Grids and structured parallel programming

- Applications may contain parallel components
  - in the simplest case, a parallel component is allocated to a single Grid node (cluster, supercomputer),
  - advancement in networking technology: parallelism can be effectively exploited at the large-scale level too.

- More in general, and more important: structured parallelism is a methodology for designing and for managing high-performance Grid-aware application components according to QoS requirements.
Grid.it approach: high-performance, Grid-aware component technology

- Joining component technology and structured parallel programming technology
  - to achieve high-performance, Grid-aware, component-based applications

- The intimate link between Grid programming and structured parallel programming
  - Structured parallel programming as a methodology to enrich the component model with features able to meet QoS requirements
  - Dynamically modifying the allocation, replication / partitioning of the application components, in order to maintain the proper degree of performance, or in order to significantly increase performance when necessary
  - Run-time exploitation of the performance models and implementation templates (fundamental feature of structured parallel programming)
Ideas for Grid-aware components

- **“Contract”** associated to every component (interface), defining possible application requirements:
  - performance, fault tolerance, …
- Every contract is specified by means a structured parallel program
  - using the ASSIST model
- Initial configuration: established at compile-time
- At run-time, the performance model is used to modify the configuration of the composition (in a parametric manner):
  - replication, partitioning, scheduling policy, distribution of data, …
    (all are *programming constructs* in ASSIST)
  - exploiting monitoring, profiling, performance modeling, resource management and information services
Example: an “adaptive pipeline”

- **Gen**
  - Generator of objects stream
  - By default: data-parallel implementation onto a single parallel node.
  - On restructuring: the number of partitions may be varied and allocated onto different nodes.

- **F1**
  - Objects transformation by function F1

- **F2**
  - Objects transformation by function F2
  - By default: sequential implementation.
  - On restructuring: farm implementation,
    - number of workers determined dynamically.

- **F3**
  - Objects transformation by function F3
  - A stream parallel + data parallel composition mapped onto a single parallel node.

- Data intensive, Grid memory hierarchy interface
A snapshot of the evolution of our adaptive application at a certain time.
A possible re-allocation: according to the outcome of the performance model, *some data-parallel partitions* and the *farm collector* can be re-allocated onto different nodes.

Component-structured application
Component-structured application

Reconfiguration of the *farm* component: *more workers* are required to grant the needed degree of performance.
Reconfiguration of the *data-parallel* component: *more partitions* are required to grant the needed degree of performance.
Data-intensive applications

Abstraction of Memory Hierarchy

Abstraction of Shared Objects

Component-structured application

Scheduling and configuration of complex, high-volume data flows through multiple levels of hierarchy
Data-intensive computations in ASSIST

Object (possibly high-bandwidth)

External Object Interface (possibly parallel)

Input Section

Output Section

ASSIST parmod for the high-performance abstraction of Object

Abstraction of high-performance objects can be implemented by ASSIST parmod(s), with proper interface (expressed in ASSIST or another formalism)
Thanks to


ISTI-CNR group, Pisa: Domenico Laforenza, Salvatore Orlando (Univ. of Venice), Raffaele Perego, Nicola Tonellotto, Ranieri Baraglia
Thank you for attention
High-level view of Grid applications

Application

• High-level languages, compositionality, modularity and interoperability
• Compiling Tools
• Run Time Support
• Performance Model (Cost Model) for static and dynamic optimizations
• Development, loading, execution, monitoring,…, reconfiguring tools

Programming Environment

Middleware ⇒ Grid Abstract Machine

Basic HW+SW platform

It is not necessarily the same Middleware “as before”: it should be defined and realized according to the needs of the Programming Environment.