Distributed Maple

Lessons Learned on Parallel Computer Algebra in Distributed Environments

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria

Currently: Department of Engineering for Computer-based Learning (CBL)
Upper Austrian University of Applied Sciences, Hagenberg, Austria

Contents

- Introduction
- Architecture
- Fault Tolerance
- Application Programming
- Conclusions

Introduction

Distributed Maple

- System for parallel computer algebra [Euro-CM-Par 1999].
  - Distributed coordination environment in Java.
  - High-level parallel programming model in Maple.
  - Execution in numerous parallel and distributed environments.
    - Origin 3800 MPP, Sun E3000 SMP, Beowulf clusters, heterogeneous networks.
- Development of various parallel applications and algorithms.
  - Reliable plotting of algebraic curves in 2D [EuroPar 2000, CASC 2000]
  - Reliable plotting of algebraic surfaces in 3D [ASCM 2000]
  - Neighborhood analysis of algebraic curves [DapSys 2000, PACT 2001]
- Fault tolerance support [EuroPar 2001, DapSys 2002].

Comprehensive summary in [JSC 2003].

History

- 1998: Parallel CASA algorithms.
  - Computer Algebra Software for Algebraic Geometry.
  - Franz Winkler et al, 1990–
- 1999: First stable version, first experiments.
  - Basic parallel computer algebra algorithms.
- 1999–2000: Diploma thesis (Christian Mittermaier).
  - `pacPlot`, `ssiPlot`, `neighbGraph`.
- 2000–2001: Refinements and extensions.
  - More/revised parallel algorithms; performance evaluation; Distributed Mathematica.
- 2001–2003: PhD thesis (Karoly Bosa).
  - Fault tolerance for Distributed Maple.

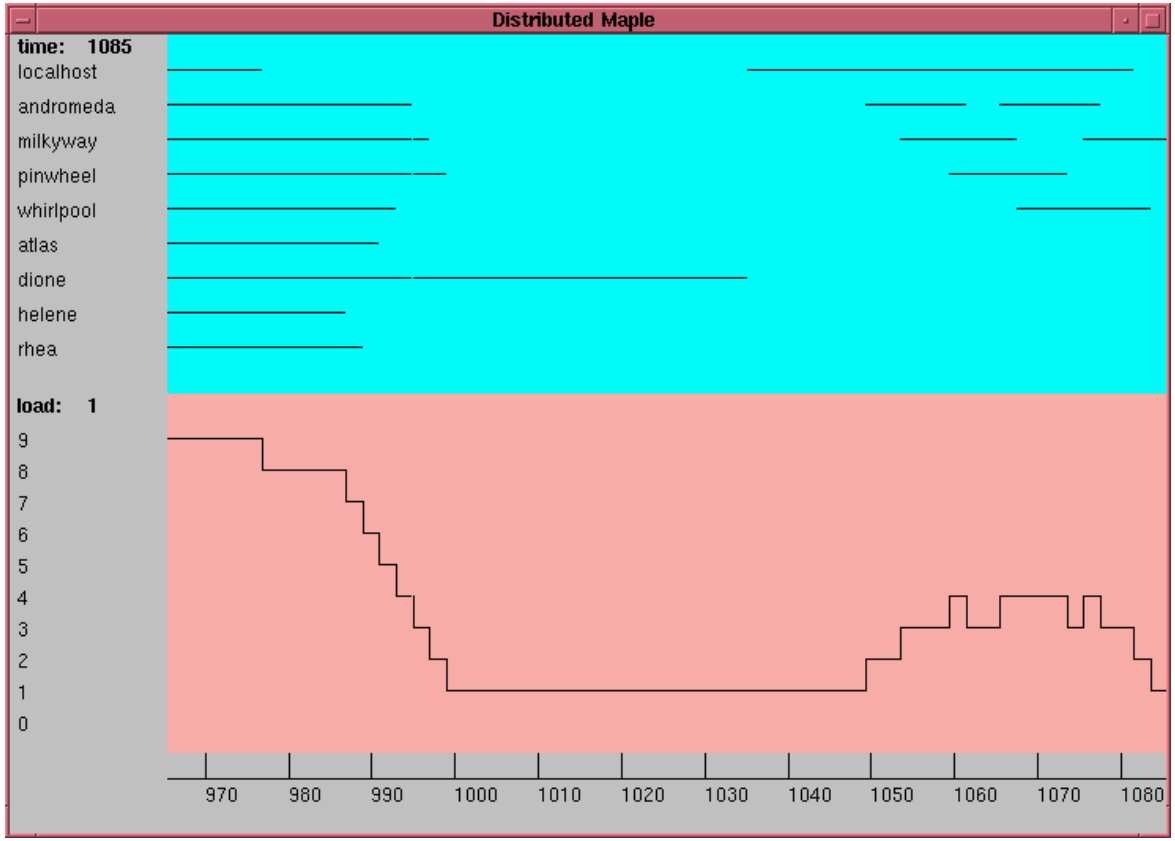
Related Work

- Own:
  - PACLIB kernel (1992): multithreaded SMP kernel for CA library SACLIB.
  - pD compiler (1994): para-functional environment on top of PACLIB.
- Maple in parallel context:
  - Sugarbush (1990, Char): Maple and Linda.
  - ||MAPLE|| (1993, Siegl): Maple and Strand.
  - Maple on Paragon (1997, Bernardin): Maple and message passing.
  - PVMaple (2000, Petcu): Maple and PVM (inspired by Distributed Maple).
  - GPH-Maple (2000, Loidl and Schreiner): Maple and Parallel Haskell.
- Distributed computer algebra based on various C-libraries:
  - PAC++ (Gautier et al, 1994), Sturm (Hong and al, 1994), DTS (Küchlin and al, 1995), FoxBox (Diaz and al, 1998), ...
  - PVM or MPI.

User Interface

```
    |\^/|     Maple V Release 5.1 (Universitaet Linz)
._|\|   |/|_. Copyright (c) 1981-1998 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read `dist.maple`;
Distributed Maple V1.1.10 (c) 1998-2001 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[deneb,solaris], [iris,irix]]);
connecting deneb...
connecting iris...
                                        okay
> t1 := dist[start](int, x^n, x):
> t2 := dist[start](int, x^n, n):
> dist[wait](t1) + dist[wait](t2);
                                 (n + 1)     n
                                x           x
                                -------- + -----
                                 n + 1     ln(x)
```

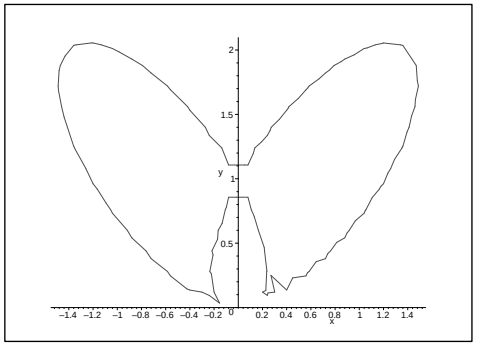
On the Fly Visualization



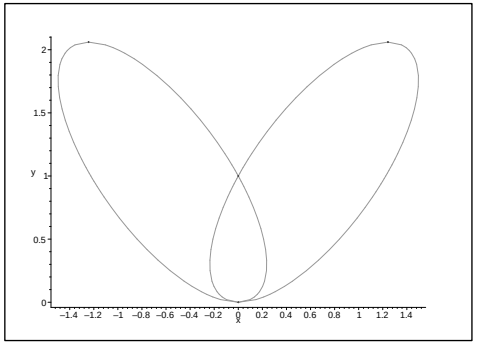
Application: Reliable Plotting of Plain Algebraic Curves

- CASA `pacPlot`
  - Tran-Quoc Nam, 1995.
- Hybrid symbolic-numerical method
  - Symbolic computation of critical points.
  - Numerical plotting of branches between points.
- Distributed Maple `pacPlot`
  - Christian Mittermaier, 2000.
  - Parallel resultants, real root isolation, solution check, and solution refinement.

Speedup of 16 on heterogeneous cluster.



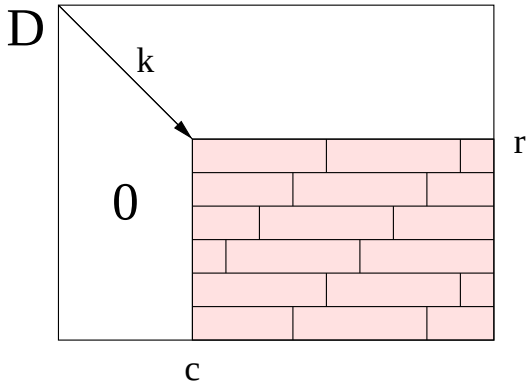
Maple `implicitPlot`



CASA `pacPlot`

Application: Reliable Plotting of Algebraic Space Curves

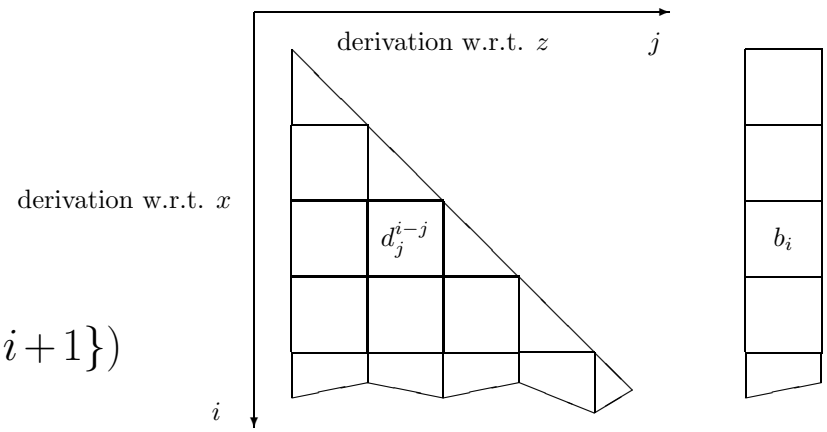
- CASA `ssiPlot`
  - Tran-Quoc Nam, 1995.
- Hybrid symbolic-numerical method
  - Basic idea similar to `pacPlot`.
  - Details different.
- Core: Dixon resultant.
  - Determinant of Dixon matrix.
- Distributed Maple `ssiPlot`
  - Parallel Gaussian elimination.



Speedup of 12 on heterogeneous cluster.

Application: Resolution of Singularities

- CASA `neighbGraph`
  - Tran-Quoc Nam, 1995
- Core: iterative construction
  - $b_0(x) := \mathrm{gsfd}(p(x,0))$
  - $b_{i+1}(x) := \gcd(b_i(x), \{d_v^u(x,0) : u+v = i+1\})$
  - $n := \min\{i : \deg b_i = 0\}$
- Distributed Maple `neighbGraph`
  - Christian Mittermaier 2000, Schreiner 2001.
  - Parallel computation of triangular matrix of partial derivatives d_u^v



Speedup of 30 on Linux cluster (superlinear).

Architecture

System Requirements

- Availability.
  - To everyone, everywhere, today and tomorrow.
  - Widely available industrially supported base technologies.
  - Open source or COTS components, no special source licenses (Maple).
- Portability and manageability.
  - Keep machine dependence at a minimum.
  - Focus on heterogeneous environments.
- Usability.
  - Reuse existing Maple libraries.
  - Focus on mathematical programmers with little systems experience.
- Performance.
  - Speedup over sequential Maple implementation.

Design Decisions

- Maple as compute engine without intrusion.
  - Programming language and algorithm libraries.
  - Only use of standard interfaces.
- Distributed execution framework implemented in pure Java.
  - Networking code executed by Java Virtual Machine.
  - Binary compatibility across machines.
- Para-functional programming model, automated task scheduling.
  - Expressed in Maple $\Rightarrow$ single language for mathematical programmer.
  - High level abstractions than message passing models.
  - Based on previous PACLIB/pD experience.

Add functionality (only) when parallel algorithm would benefit.

Constraints

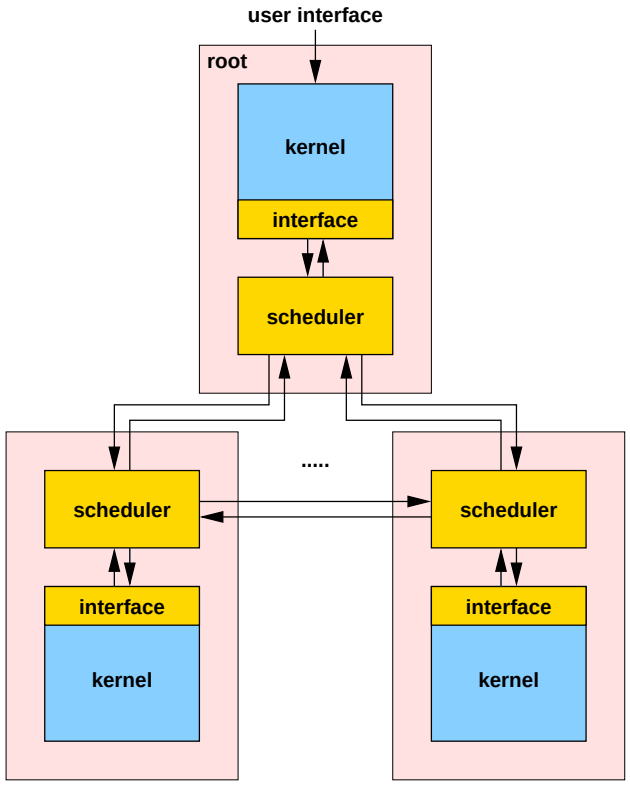
- Maple is purely sequential
  - No multi-threading.
  - No asynchronous event handling.
- Maple is closed.
  - No access to data structures in kernel.
  - (Maple 6 introduces dynamic linking interface).
- Only external interface: input/output.
  - Linearization of internal data structures.
  - Reading/writing from/to files or pipes.

System architecture has to cope with constraints.

System Model

Wrapped compute engines.

- Computing Layer.
  - Maple kernels as computational engines.
- Coordination Layer.
  - Distributed interaction and task scheduling.
  - Java program *dist.Scheduler*.
- Interface Layer.
  - Connecting Maple kernel to scheduler.
  - Maple package *dist.maple*.



System not bound to Maple (→ Distributed Mathematica).

Programming Interface

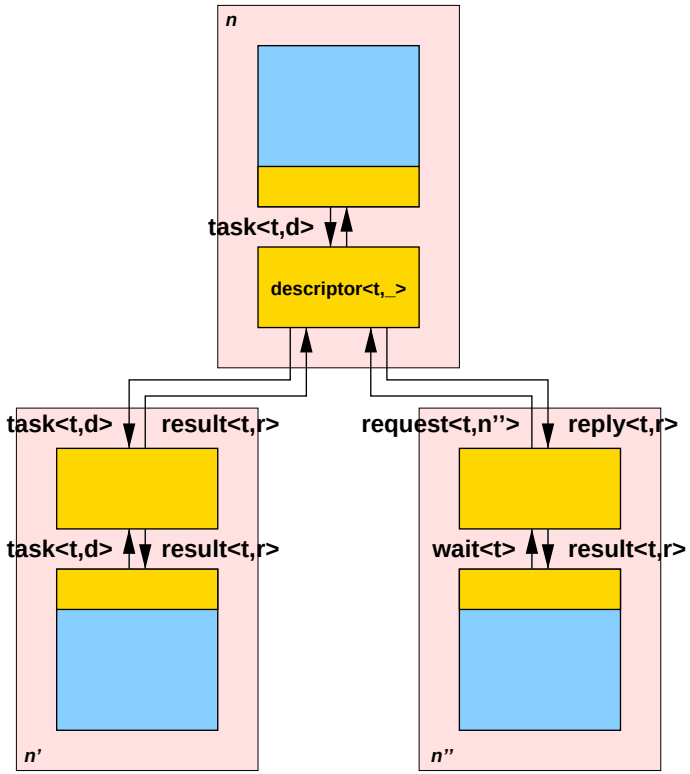
- Functional Parallelism
  - Task Creation: `t = dist[start](f, ...)`.
  - Synchronization: `r = dist[wait](t)`.
- Non-Determinism
  - Selection: `(i, r) = dist[select](tlist)`.
  - Abortion: `dist[delete](t)`.
- Shared Data Objects
  - Create empty object: `d = dist[data]()`.
  - Write data: `dist[put](d, v)`.
  - Wait for data: `v = dist[get](d)`.

Derived from functional/logic/dataflow models.

Execution Model

1. $\mathrm{k}_n \xrightarrow{\text{task}\langle t,d\rangle} \mathrm{s}_n \xrightarrow{\text{task}\langle t,d\rangle} \mathrm{s}_{n'} \xrightarrow{\text{task}\langle t,d\rangle} \mathrm{k}_{n'}$
   $\xrightarrow{\text{result}\langle t,r\rangle} \mathrm{s}_{n'} \xrightarrow{\text{result}\langle t,r\rangle} \mathrm{s}_n$
2. $\mathrm{k}_{n''} \xrightarrow{\text{wait}\langle t\rangle} \mathrm{s}_{n''} \xrightarrow{\text{request}\langle t,n''\rangle} \mathrm{s}_n \xrightarrow{\text{reply}\langle t,r\rangle} \mathrm{s}_{n''}$
   $\xrightarrow{\text{result}\langle t,r\rangle} \mathrm{k}_{n''}$

- n ... node creating t and storing its result.
- n' ... node executing t.
- n'' ... node requesting result of t.



n is synchronization point between n' and n''

Execution Model: Task scheduling

Two levels of task pools.

- Task pool in root node and in every remote node.
- All new tasks are delegated to root node.
- Root node forwards tasks to remote pools.
  - Root nodes lets remote task pool hold up to max tasks.
  - Remote node tells root when it has less than min tasks in pool.
  - $max = 0$: wait until remote kernel reports idleness.
  - $max = 1$: submit additional task to already executing task.
  - $max > 1$: load imbalances may occur (no task stealing yet).

Hide latency of task submission to keep remote kernel busy.

Execution Model: Kernel Management

How many Maple kernel instances per node?

- Initially: one kernel per node.
  - Tasks created by `dist[start]` are executed by fixed number of kernels.
  - Each kernel has stack of tasks of which only top is active.
  - When task waits for result, new task can be submitted to kernel.
  - When task returns result, previous task can resume execution.
- Later extension: multiple kernels per node.
  - Task created by `dist[process]` to receive kernel on its own.
  - Each node has pool of kernels which is extended on demand.
  - When node receives task, it assigns unused kernel to task.

Avoid deadlock of tasks that communicate by shared data.

Fault Tolerance

Session Failures

An awfully lot of things may go wrong.

- A session may fail because of various faults:
  - Machine/OS (reboot).
  - Network (broken connection).
  - Software (bugs in JVM or Maple).
- Computation sizes limited by meantime between failures.
  - $\ll$ 1 day.
  - Not possible to run "real world" applications.

We need to cope with faults.

Failure Model

We focus on the following situations.

- Node may fail at any time by stopping.
  - Does not send messages any more.
- Connection may fail at any time by stopping.
  - Does not transmit messages any more.
- Failures may be temporary.
  - Node may be rebooted.
  - Connection may become operational again.
  - Failure detection may have been wrong.

Cannot reliably recognize/distinguish these situations.

System Assumption

Subset of nodes has access to common network file system.

- Assumptions on file system:
  - Independent of any session node.
  - Persistently available across failures.
  - Holds data reliably.

Used to implement the concept of a "stable storage".

Fault Tolerance

Two basic strategies.

1. Recover from session failures caused by faults.
   - Time already spent in computation is not wasted.
   - Technique: log task results on stable storage.
2. Avoid session failures by tolerating faults.
   - Computation is not interrupted.
   - Technique: reschedule tasks from failed nodes.

Distributed Maple applies both strategies.

Terminology

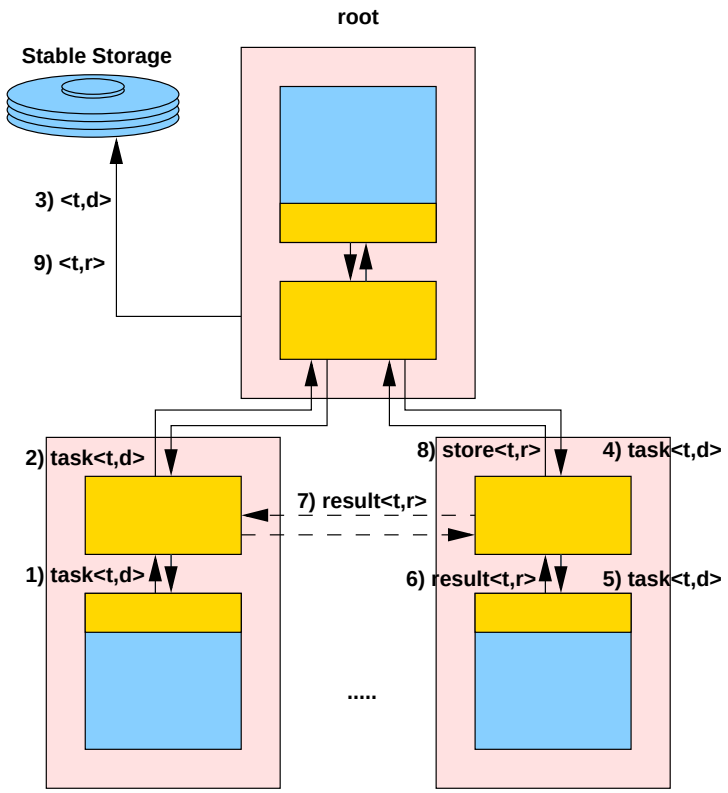
- At any time, one node is designated as the root.
  - Must have access to stable storage.
  - Initially the node to which the user interface is attached.
  - If root fails, another eligible node may become the root.
- The root receives all newly created tasks.
  - Schedules task for execution on any node of the session.

The root plays a special role in all fault tolerance mechanisms.

Result Logging

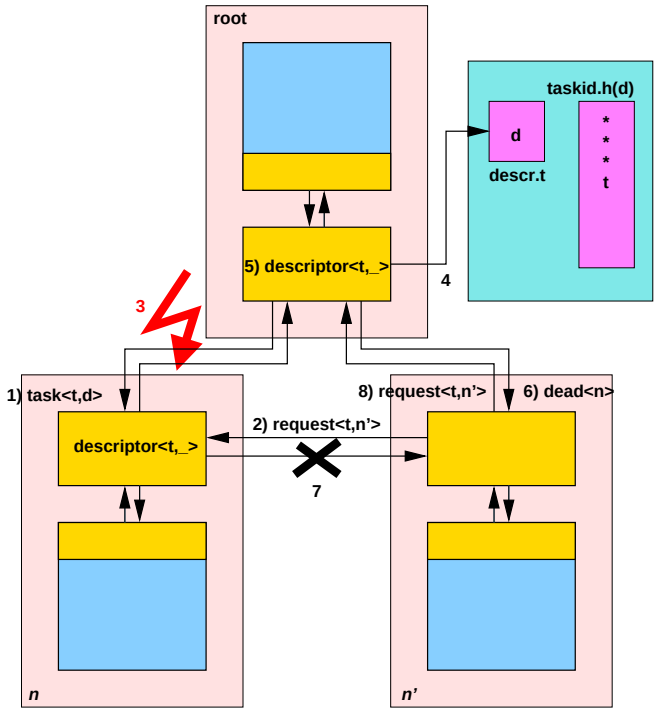
- Root logs task descriptions.
  - If the node fails that executes the task, the task can be rescheduled to another node.
- Root logs task results.
  - If the node fails that holds the task result, the result can be restored.

If a failed session is restarted, it reads the stored results rather than recomputing them.



Tolerating Non-Root Failures

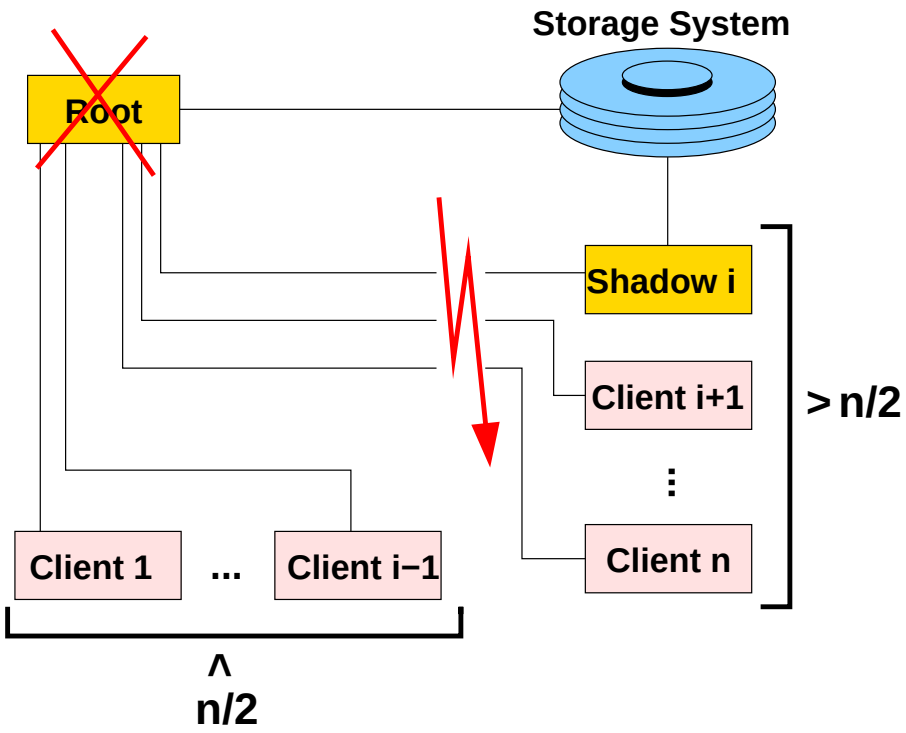
- Root considers node n as dead.
  - Cannot contact n any more.
  - Is informed by other node that cannot contact n any more.
- Announces death of n.
  - All nodes maintain the same system view.
- Restores info from storage.
  - Reschedules tasks currently executed on n.
  - Manages result descriptors stored on n.



Lost tasks are rescheduled, lost task results are restored.

Tolerating Root Failures

- Node considers root dead.
  - Asks another potential root (the shadow) to become new root.
  - Shadow itself checks contact to root.
- Shadow agrees to offer.
  - Shadow starts election among all nodes it can contact.
  - If it gathers more than $n/2$ votes, it becomes the new root.



If network is partitioned, the larger partition continues execution.

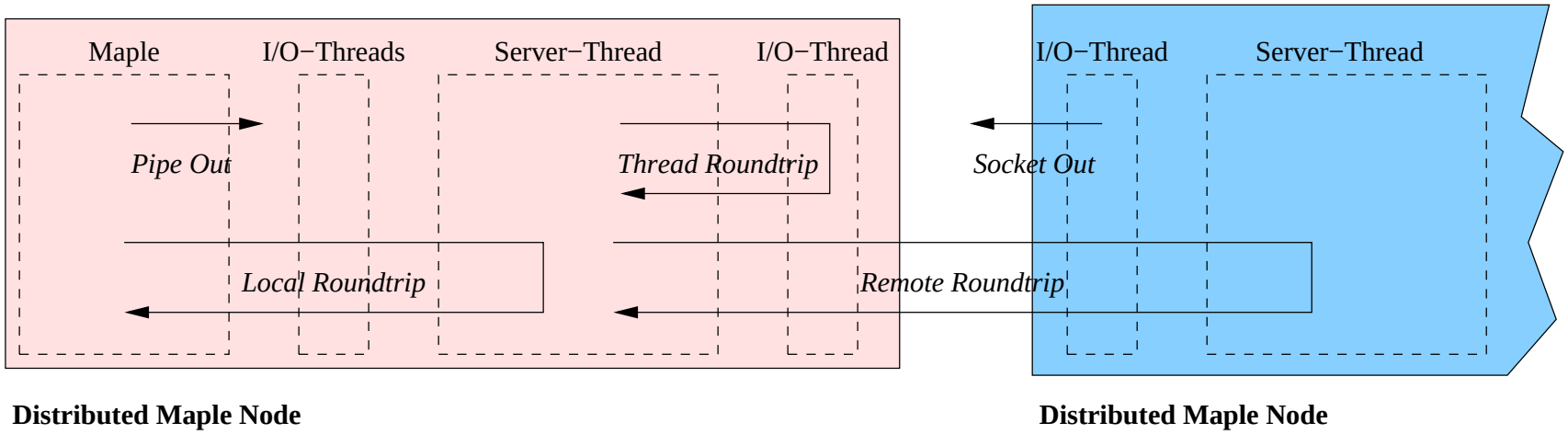
Fault Tolerance Achievements

- If a session fails or deadlocks:
  - The results computed so far may be reused after restart.
- A session does not fail except for:
  - Neither the root or any shadow can keep contact to at least $n/2$ nodes.
    - The root and all shadows have failed.
    - The network is partitioned such that no part has more than $n/2$ nodes.
- A session does not deadlock except for:
  - A kernel (Maple) process fails.

We plan to deal with the last issue.

Application Programming

Performance Model



Multiple communication layers.

Performance Measurements

Measurement	Time (ms)
Thread Roundtrip	<0.1
Local Roundtrip	0.8
Remote Roundtrip	31.0
Pipe Out	0.5
Socket Out	<0.1
Start Overhead	0.2
Wait Overhead	0.1
Latency	0.5
Bandwidth pipe:	1030 KB/s
Bandwidth socket:	3938 KB/s

Linux PC

Measurement	Time (ms)
Thread Roundtrip	<0.1
Local Roundtrip	1.2–2
Remote Roundtrip	150–300
Pipe Out	0.7
Socket Out	0.4
Start Overhead	0.4
Wait Overhead	0.3
Latency	10–12
Bandwidth pipe:	853 KB/s
Bandwidth socket:	4100 KB/s

SGI Octane

Pipe bandwidth and remote roundtrip time represent bottlenecks.

Consequences

How can we take these factors into account?

- Avoid the waiting for remote requests.
  - Store task results on creator node, most likely it will need results.
  - Have always a task in remote pool, i.e., use $max = 1$ for task scheduling.
  - Send data spontaneously (without being asked, e.g. for monitoring).
- Transfer only minimal data beyond Maple boundaries.
  - Compact binary encoding of Maple data (`.m` format).
  - Programming style reducing superfluous communication (see later).
- Keep task granularity above the $\approx$ 1–2s limit.

Consequences on system design and on application programming.

Application Programming

Experience base: 13+2 parallel algorithms.

- Various applications combine multiple parallel algorithms.
  - pacPlot: 4 parallel algorithms.
  - ssiPlot: 5 parallel algorithms (nested).
  - Berlekamp: 3 parallel algorithms.
  - resultant, gcd: 2 parallel algorithms.
- 2 parallel versions newly developed after review.
  - Original performance insufficient.

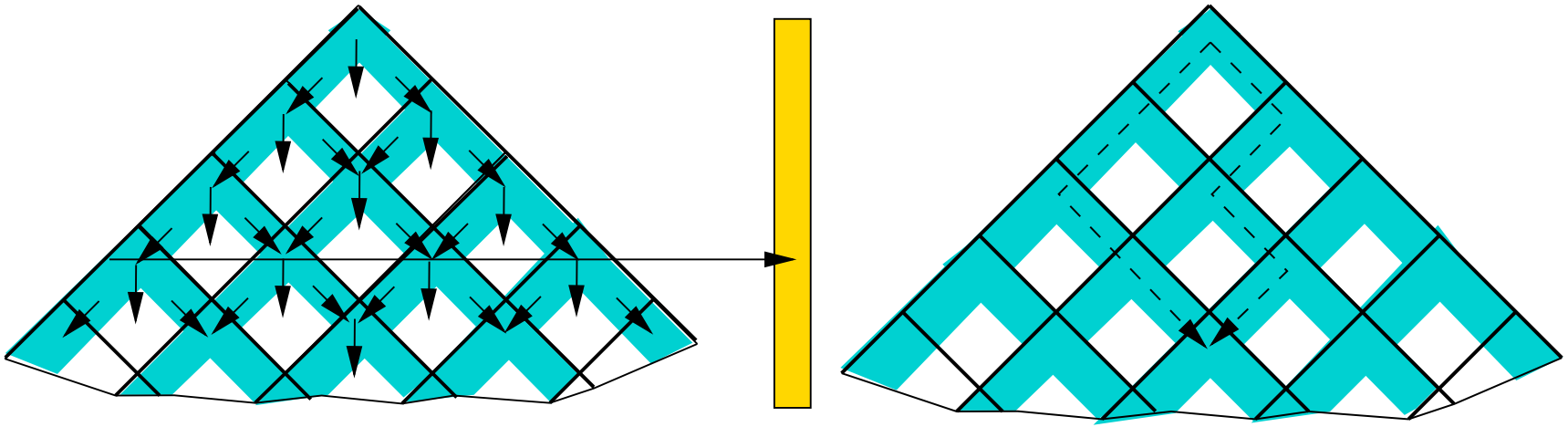
Selection from algebraic geometry and computer algebra.

Programming Patterns

- Most dominant: simple task or data parallelism (6).
  - Start independent tasks, wait for their results.
- Dataflow synchronization (3).
  - Express more complicated dependency patterns.
- Non-deterministic synchronization (4).
  - Wait for bag of results in any order.
- Numerous variations.
  - Nested parallelism, recursion, speculation, shared data.
  - Also persistent tasks with explicit placement.

First two categories (only) fit typical functional skeletons.

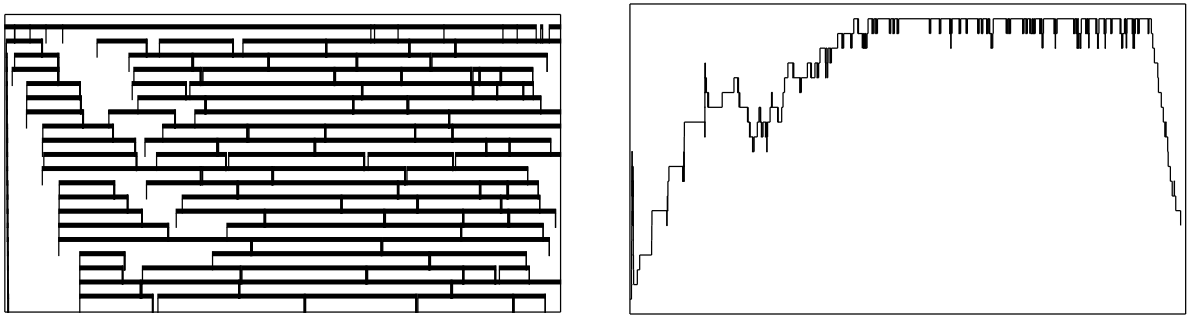
Dataflow Synchronization



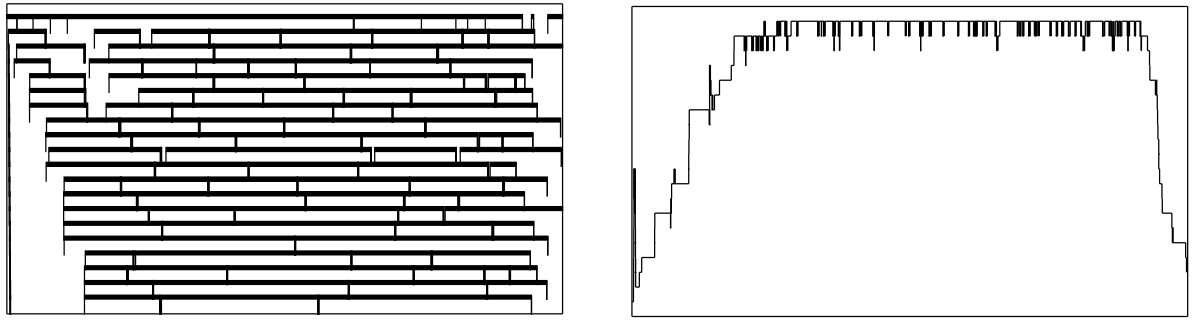
Alternative execution paths possible.

Non-Determinism

Det:



Non:



Non-determinism yields higher utilization.

Optimization Techniques

- Block work to tasks of appropriate granularity.
  - Combine asymptotic runtime analysis with experiments to measure constant factors.
  - Determine configuration parameters to let tasks run a couple of seconds (minimum).
  - Originally: $[f(i) : 0 \leq i < n]$.
  - Transformed: $[\texttt{dist[start]}(F, l(i), u(i))) : 0 \leq i < t(n)]$.
- Transmit descriptions rather than contents.
  - Originally: $f(g(x))$
  - `dist[start](`fg`, `x`)` rather than `dist[start](`f`, `$g(x)$`)`.
- Use shared data for common task arguments.
  - Originally: $[f(i, x) : 0 \leq i < n]$
  - Transformed: $\texttt{dist[put]}(d, x); [\texttt{dist[start]}(f, i, d) : 0 \leq i < n]$
  - Drastically influences scheduling patterns!

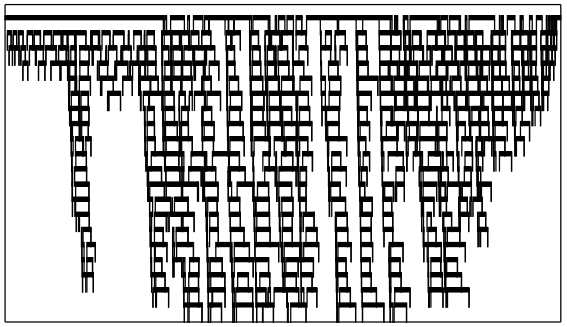
Optimization Techniques

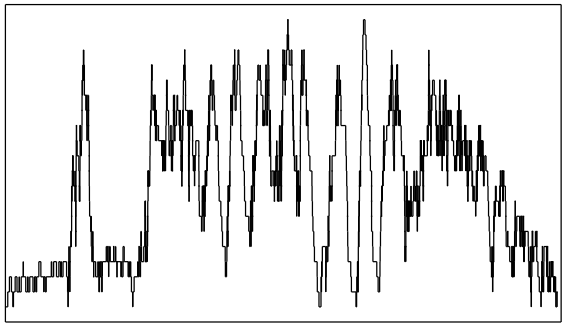
- Other uses of shared data:
  - As synchronization points (express complex dependence patterns).
  - For decomposition of task arguments (overcome Maple limitations).
- Avoid application-level scheduling.
  - High latency of manager/worker communication.
  - Let tasks be automatically scheduled by task/dataflow dependencies.
  - May lead to increased memory consumption!

Consideration of communication and scheduling issues still crucial for good performance.

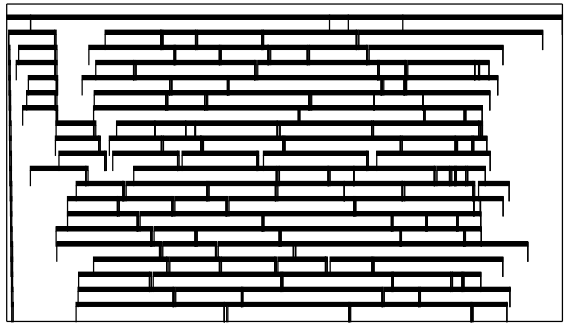
Avoiding Application-Level Scheduling

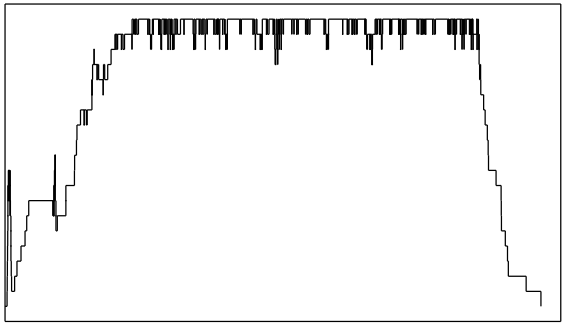
M/W:





DF:





Dataflow scheduling with shared data for synchronization.

Conclusions

Conclusions

Basic design decisions.

- Maple as a compute engine without intrusion.
- Distributed execution framework implemented in pure Java.
- Para-functional model with automated task scheduling.

How to assess these in retrospective?

Conclusions

Decision: Maple as a compute engine without intrusion.

+ Various generations of Maple encompassed.
  - Maple V.3, Maple V.4, Maple V.5, Maple 6–8.
+ Reuse for different compute engines.
  - Implemented interfaces: Maple, Mathematica; discussed: Matlab?
+ Publically available.
  - Installed at a handful of sites.
+ All flexibility by distributed execution framework.
  - No multi-threading, no asynchronous event handling required from compute engine.
– Significant bandwidth loss at interface layer.
  - Since Maple 6: dynamic linking capabilities.

Conclusions

Decision: Distributed execution framework implemented in pure Java.

+ Portability from the beginning with minimum effort.
  - Multithreading and networking.
  - Numerous parallel and distributed architectures.
+ System management and code distribution easy.
  - One installation in every NFS domain.
– Roundtrip latency high.
  - Possible alternative: MPI library via JNI.

Conclusions

Decision: para-functional model with automated task scheduling.

+ Considerable simplification of programming.
  - Simple parallel patterns are simple to express.
  - Simplification of additional system requirements (fault tolerance).
+ Self-synchronized shared data very valuable.
  - Formulation of more complex dependence patterns.
  - Minimization of data transfer between compute layers.
+ Delegating scheduling/synchronization to distributed framework.
  - Manual handling within compute layer much too inefficient.
– Good efficiency requires understanding of system operation.
  - Essential to have a basic mental performance model and to apply optimization techniques.
  - Manual memory control for shared data and task results required (in our system).

Summary

- If you have the CA application of your life, then ...

  ... write it from scratch in C/Fortran and use MPI/PVM for parallelization.

- However, if you enjoy the features of a system like Maple, then ...

  ... Distributed Maple is a reasonable basis for getting good performance with limited efforts.

- To optimize Distributed Maple for performance only, I would ...

  ... still stick to the overall architecture, programming model, and system protocols,

  ... use C+MPI for the distributed computing layer (sacrificing ease of portability and operation),

  ... use dynamic the new linking capabilities of Maple for combining the two system layers (sacrificing independence from the computing engine).

`http://www.risc.uni-linz.ac.at/software/distmaple`