Parallel Systems Software, short overview → MosiX

Hermann Härtig

SS 2010

TECHNISCHE
UNIVERSITÄT
DRESDEN

Linux, Small kernels, and Linux

SMP (Linux,K42, ...)	MPP (Jaguar, Blue Gene, ...)	Clusters (MosiX, ...)
Shared Memory SMP	Distributed Memory	COTS networks
Linux syscall interface	Message Passing Interface	Distribute Linux
Balance Load, Optimise locality and concurrency	Partition	Balance Load dynamically

SMP: Shared Memory / Symmetric MP

- Characteristics of SMP Systems:
  - Highly optimised interconnect networks
  - Shared memory (with several levels of caches)
  - Sizes: 2 .. ~1024 CPUs

- Successful Applications:
  - Large Linux (Windows) machines / servers
  - Transaction-management systems

- Not usually used for:
  - CPU intensive computation, massively parallel Applications

MPP: Massively Parallel Multiprocessors

- Characteristics of MPP Systems:
  - Highly optimised interconnect networks
  - Distributed memory
  - Size today: up to few 100000 CPUs (cores)

- Successful Applications:
  - CPU intensive computation, massively parallel Applications, small execution/communication ratios

- Not optimal for:
  - Transaction-management systems
  - Unix-Workstation + Servers

"Clusters"

- Characteristics of Cluster Systems:
  - Use COTS (common off the shelf) PCs/Servers and networks
  - Size: No principle limits
- Successful Applications:
  - CPU intensive computation, massively parallel Applications, larger execution/communication ratios
  - Data Centers, google apps
- Not optimal for:
  - Transaction-management systems
  - Unix-Workstation + Servers

Parallel Programming Models

- Organisation of Work
  - Independent, unstructured processes (normally executing different programs) independently on nodes (make and compilers, ...), “pile of work”
  - SPMD: single program on multiple data
    asynchronous handling of partitioned data
    “map/reduce” (google)
- Communication
  - Shared Memory, shared file system
  - Message Passing:
    Process cooperation through explicit message passing

Usage- and Programming Model

- SPMD

```
while (true) {
  work
  exchange data (barrier)
}
```

- Common for many MPP:
  All participating CPUs: active / inactive

- Techniques:
  - Partitioning (HW)
  - Gang Scheduling
  - Load Balancing

MPI, very brief overview

- Library for message-oriented parallel programming.
- Programming-model:
  - MPI program is started on all processors
  - Static allocation of processes to CPUs .
  - Processes have “Rank”: 0 ... N-1
  - Each process can obtain its Rank (MPI_Comm_rank).
- Typed messages
- Communicator: collection of processes that can communicate, e.g., MPI_COMM_WORLD
- MPI_Spawn (MPI – 2)
  - Dynamically create and spread processes

MPI - Operation

- Init / Finalize
- MPI-Comm-Rank delivers “rank” of calling process, for example

  MPI_Comm_Rank(MPI_COMM_WORLD, &my-rank)

  if (my_rank != 0)
  …
  else ….

- MPI_barrier(comm) blocks until all processes called it
- MPI_Comm_Size how many processes in comm

MPI – Operations Send, RCV

- MPI_Send (

```
void* message,
int count,
MPI-Datatype,
int dest,            /*rank of destination process, in */
int tag,
MPI_Comm comm) /* communicator*/
```

- MPI_RCV(

```
void* message,
int count,
MPI-Datatype,
int src,              /* rank of source process, in */
                     /* can be MPI_ANY-SRC */
int tag,             / can be MPI_ANY_TAG */
MPI_Comm comm, /* communicator*/
MPI_Status*   status); /* source, tag, error*/
```

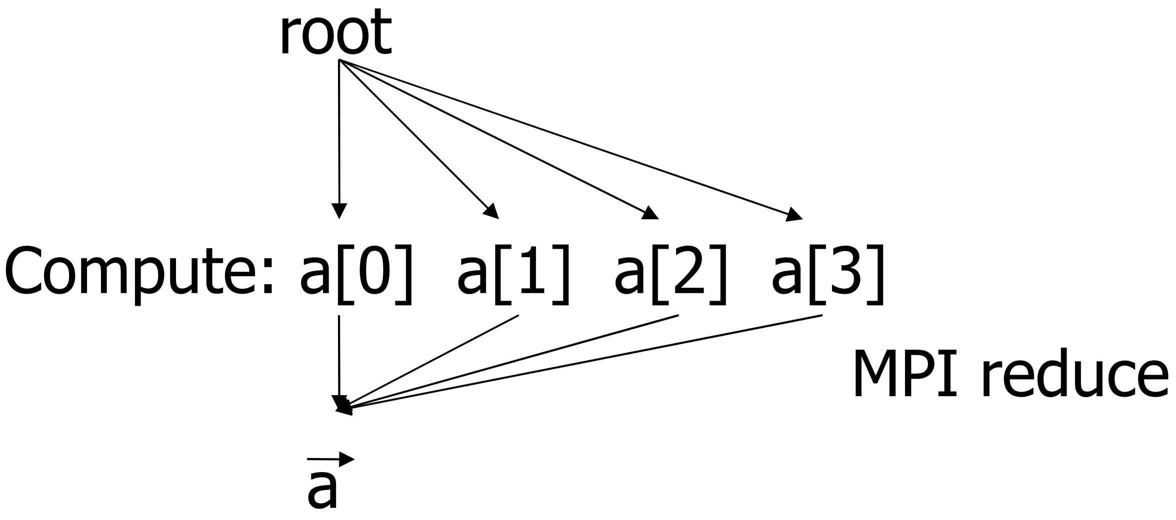
MPI – Operations Broadcast

- MPI_BCAST(
    void * message,
    int count,
    MPI-Datatype,
    int root,
    MPI_Comm comm)
- process with rank == root sends,
  all others receive message
- implementation optimised for particular interconnect

MPI – Operations

- Aggregation:
  - MPI_Reduce
    - Each process holds partial value,
    - All processes reduce partial values to final result
    - Store result in RcvAddress field of <u>Root</u> process
  - MPI_Scan
    - Combine partial results into n final results and store them in RcvAddress of <u>all</u> n processes

MPI - Operations

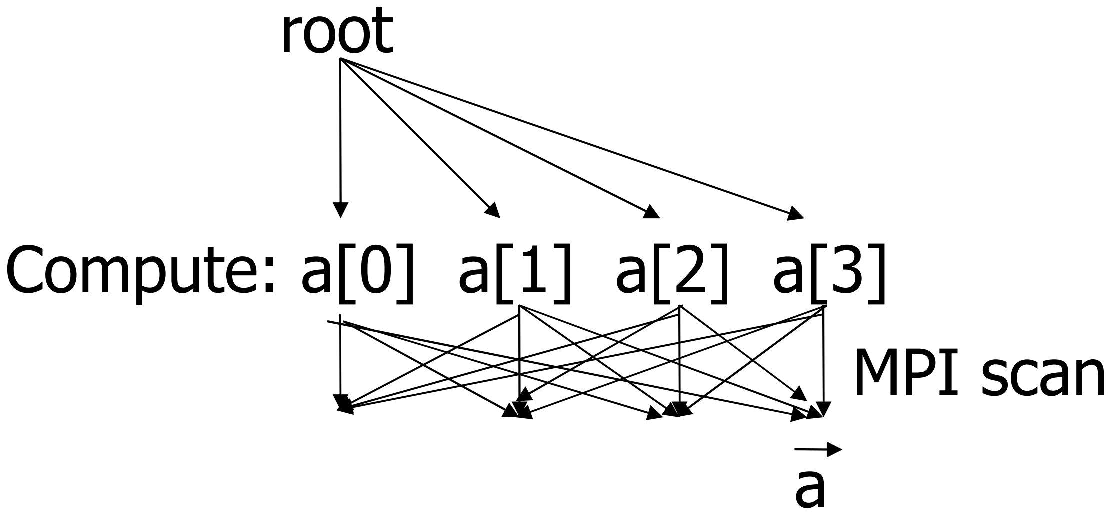


root

Compute: a[0] a[1] a[2] a[3]

MPI reduce

$\vec{a}$



root

Compute: a[0] a[1] a[2] a[3]

MPI scan

$\vec{a}$

MPI – Operations

-
  ```
  MPI_Reduce(
          void* operand,    /* in*/
          void * result,    /* out*/
          int count,        /* in */
          MP_Datatype datatype,
          MPI_Op operator,
          int root,
          MPI_Comm comm)
  ```

  predefined MPI_OPs:
  sum, product, minimum, maximum,
  logical ops, ...

Common MPP Operating-System-Model (for example Blue Gene)

- PE: compute intensive part of application
  - Micro-Kernel
  - Start + Synchronisation of Application
  - elementary Memory Management (no demand paging)

- all other OS functionality on separate Servers or dedicated nodes
- strict space sharing:
  only one application active per partition at a time

“Space” Allocation in MPP

- Assign partition from field of PEs
  - Applications are pair wise isolated
  - Applications self responsible for PEs
  - shared segments for processes within partition (Cray)
- Problems:
  - debugging (relatively long stop-times)
  - Long-running jobs block shorter jobs
- Isolation of application with respect to:
  - Security
  - Efficiency
- Buzzword: “eliminate the OS from the critical path"

“Space” Allocation in MPP

- Hardware-Supported assignment of nodes to applications
- Partitions
  - static at configuration
    Installed by operator for longer period of time
  - Variable(Blue Gene/L):
    Selections and setup on start of Job
    established by “scheduler”
  - Very flexible (not in any MPP I know):
    - increase and shrink during operation
    - Applications need to deal with varying CPU numbers

Alternative: Distribution of Load

- Static
    - Place processes at startup, don’t reassign
    - Requires a priori knowledge
- Dynamic Balancing
    - Process-Migration
    - Adapts dynamically to changing loads
- Problems
    - Determination of current load
    - Distribution algorithm
    - Oscillation possible
- successful in SMPs and clusters, not (yet ?) used in MPPs
- Most advanced dynamic load balancing: MosiX

The Limitation of CC

Example: a numerical application that computes what happens during car crash.

Such simulations typically compute one time step, require some communications about the boundaries and some global variables, and then next time step and so on. If you compute bus crash, the problem is fairly big, so each time step takes a lot of time - e.g. 1 minute. Even if you use 600 computers efficiently, you'll have 0.1 second per time step, which is a usually enough in terms of communications. So this is coarse-grain.

Same simulations, you check what happens when a hammer left in space impacts a space ship shield. This time the problem is very small, but the velocities and the materials are higher, so the time step is smaller (the physical time) and you need 1,000,000 time steps. Each time step may take 10msec. Impossible to parallelize efficient even on a 100 nodes CC since comm. cost is large. This is fine grain, and you'll have to wait A LOT until it finishes.

Challenges for Cluster Management

View provided for users/programming model

How to distribute load,

- The mechanism to migrate load
- The mechanisms to use remote resources
- Optimal placement (an NP-Hard problem)
- …

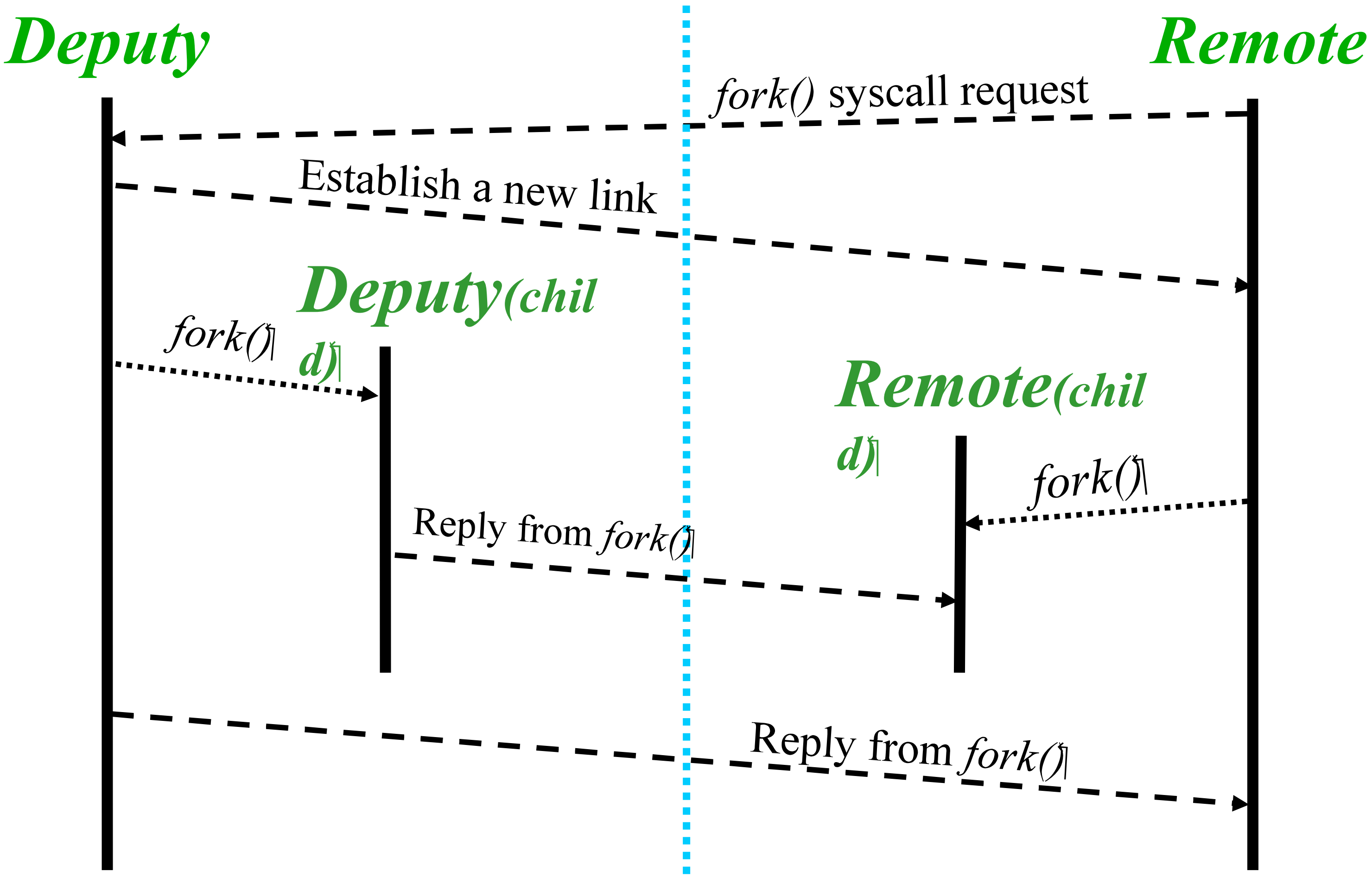
Information distribution, acting on partial knowledge

Cope with addition of nodes, subclusters, …

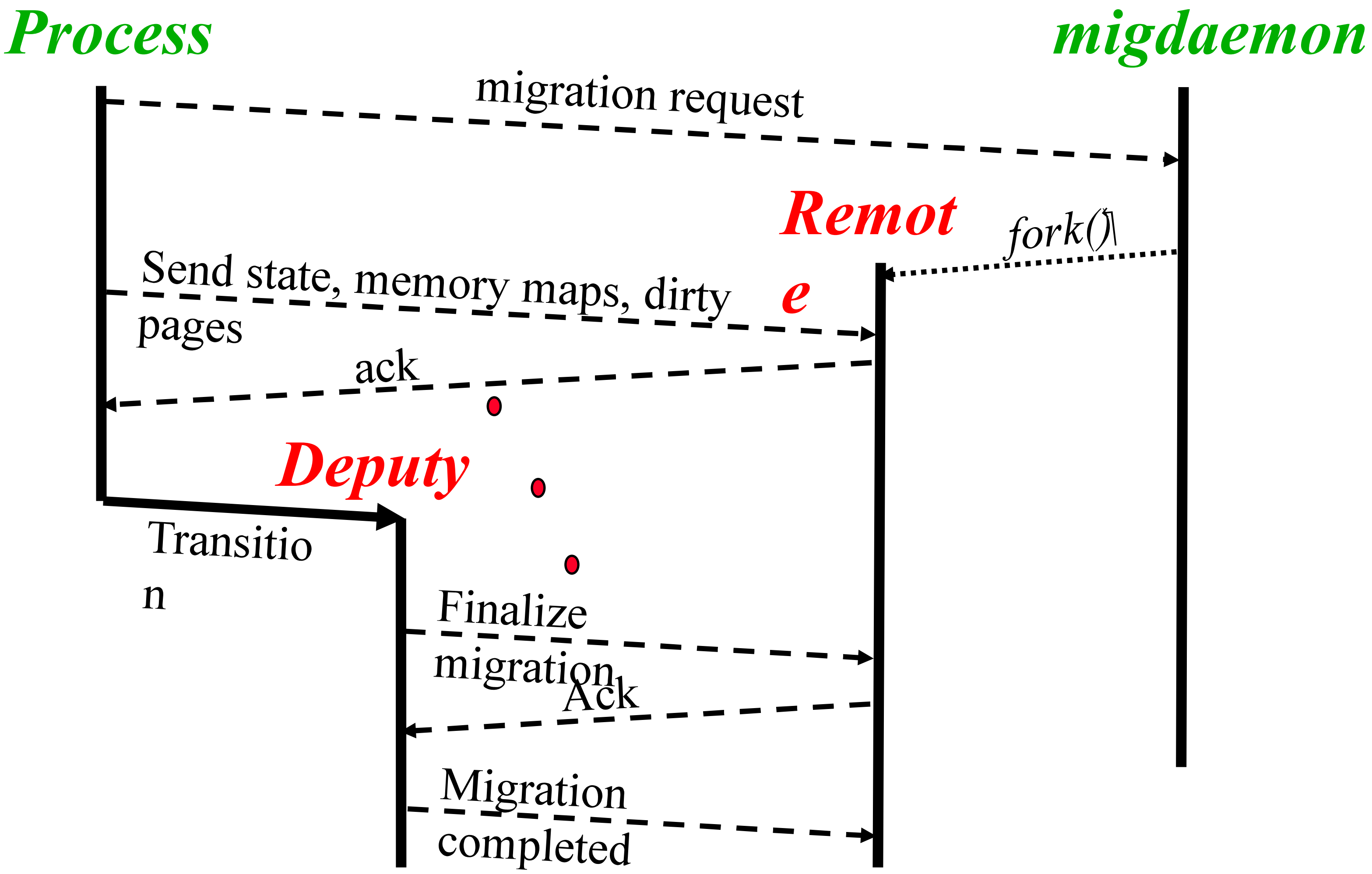
Administration

Which are the practical details

Special Case: fork()



Process Migration



Process

migdaemon

migration request

Remot
e

fork()

Send state, memory maps, dirty pages

ack

Deputy

Transitio
n

Finalize migration

Ack

Migration completed

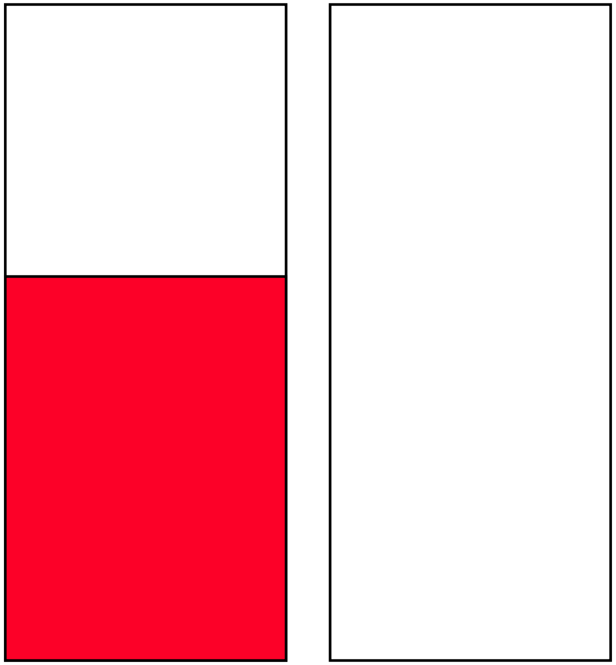
Oren Laadan/Hermann Härtig

Ping Pong and Flooding

prevent

- flooding (all processes jump to one new empty node):
  decide immediately before migration commitment (extra communication, piggy packed)
- ping pong:
  if thresholds are very close, processes moved back and forth
  => tell a little higher load than real

The Ping Pong Problem



One process two nodes

scenario:
compare load on nodes 1 and 2

node 1 moves process to equal. loads

...

<u>solutions:</u>

- add one + little bit to load
- average over time

solves short peaks problem as well
(short cron processes)

The Flooding Problem

scenario 1: new node comes in

scenario 2: node becomes unloaded suddenly

=> "everybody joins the party"

Solution:

- use expected load (committed load) instead of run queue length
- check again before committing

IPC

- IPC and load are contradictive optimum: NP hard
- apply heuristics: exchange locally