Scalable data abstractions for distributed parallel computations

Jamie Hanlon

July 10, 2014

Overview

How to build distributed data structures with message passing

Motivation & background

- distributed parallel computer architecture
- opportunities and challenges with embedded computing

Proposal

- a server notation
- implementation issues
- an example program

Distributed memory parallelism

Supercomputers have been doing this for a long time to scale performance

- large amounts of memory, large power budgets
- large problems
- fixed computation and no external interaction
- relatively few supercomputers worldwide (less than a million)

Embedded computing

Robotics, consumer devices, medical systems, automotive, ...

- Robotics: vision, language, artificial intelligence
- Interaction: sensors, actuators and haptics
- Emulation (*real-time* supercomputing)

Embedded computing

Huge numbers of devices worldwide (billions)

Increasing demands on computational performance

But they are subject very different challenges:

- small physical form factor
- limited memory, limited power
- small problem sizes
- external input and output – *real-time response* to events

Increasing use of general purpose processors and high-level languages

Scalable architecture for embedded systems

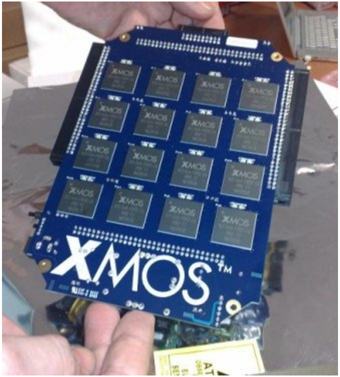
We want to build scalable distributed memory machines with large numbers of processors for embedded systems

Why?

- because parallelism is the primary means of improving computational performance
- and potentially an effective way to reduce power consumption ($P = fcV^2$)
- because distributed memory is scalable (bounded degree nodes, sparse interconnect)
- because a *tiled architecture* based on a replicated processor-memory pair is simpler to design & verify

Example: XMOS XMP-64

- 16 quad-core *general-purpose* chips
- hypercube interconnect
- 8 threads and 64KB RAM per core
- 512 threads @ 50MHz
  or 256 threads @ 100MHz
  = 25.6 GIPS
- 4MB RAM
- 400MB/s bisection bandwidth (in each direction)
- 120 x 120 mm
- 30W



Example: Swallow

- 112 cores
- 615 threads
- 2D mesh interconnect
- 56 GIPS
- 7MB RAM
- 29W

Application requirements

Embedded computing:

- broad range of problems
- diverse program requirements
- systems composed of a number of components

High performance computing:

- narrow class of problems
- often single algorithms
- require particular forms of parallelism

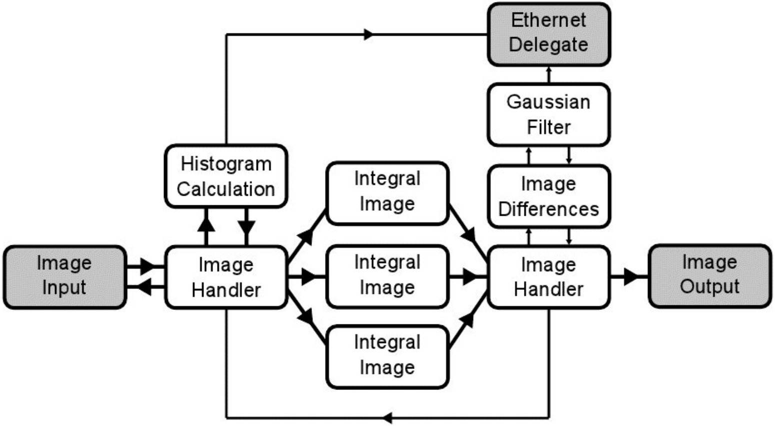
General parallel programming

Want to employ a different styles of parallelism where necessary and in combination

Small set of paradigms:

- Parallel random access machines (PRAMs/BSP)
- Process structures
- Data flow structures
- Task farms
- Event handlers

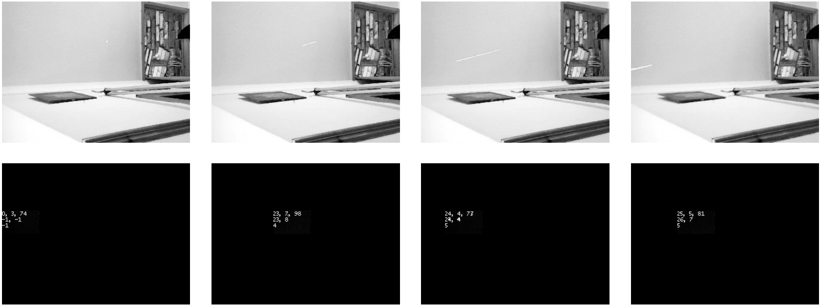
Example: adaptive visual sampling system



Adaptive Sampling for Low Latency Vision Processing, David Gibson, Neill Campbell, David Bull (University of Bristol), Henk Muller (XMOS Ltd.), 2012.

Example: adaptive visual sampling system

Fast response:



...4 frames in 0.16 seconds

Shared memory vs. message passing

Shared memory

- + provides a separation of data
- ▶ dominates architecture and programming in commodity/consumer systems
- - it doesn't capture 'flow of data' or locality

Message passing

- + essential where there is a flow of data
- + simple compilation and efficient execution
- ▶ the standard programming approach in HPC!
- - representation of data is fragmented
- - awkward when data access patterns are not known

The problem with message passing

A key problem with message passing is how to support efficient abstractions of data

- no separation of data from a computation
- awkward for arbitrary access patterns

Data abstraction

A methodology of programming is also bound to include all aspects of data structuring. Programs, after all, are concrete formulations of abstract algorithms based on particular representations and structures of data.

– Nikalus Wirth, 1978, Algorithms+data structures=programs

A fundamental principle in sequential programming

Separation of *data structures* from *algorithms*

- A data structure is a set of basic operations to efficiently manipulate a chosen representation of the data
- An algorithm is a computational procedure

Composition of data structures

Algorithms are *composed* with data structures, typically in sequence

```
Image i;
loadImage(i);
preProcess(i);
findEdges(i);
outputImage(i);
```

Distributed data structures

Data spread over many individual machines

An established concept in large-scale systems

- Internet services, e.g. web search
- Peer-to-peer networking, e.g. distributed hash tables
- Databases

Building distributed data structures

Data representation:

- must specify a *distribution* over a collection of memories
- requires a mapping of each data component to a processor location and memory location

Basic operations:

- access mechanisms (insert, delete, update, iteration, ...)
- load distribution
- replication and combining
- caching

Proposal

To combine shared resources with message passing to support data separation and abstraction

Based on a server component

A server is a compositional tool that

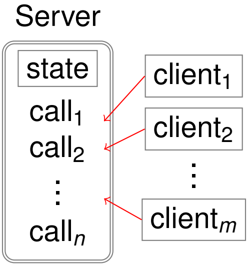
- allows the expression of a global shared state independent of a computation
- gives rise to a subroutining mechanism
- can be compiled in a simple way

Servers

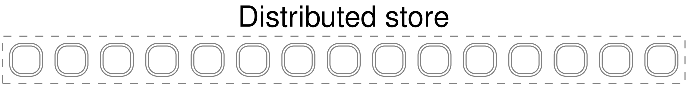
A special kind of process active only in response to *clients*

Provide a set of *calls* that behave in the same way as conventional procedure calls, except the server executes the call

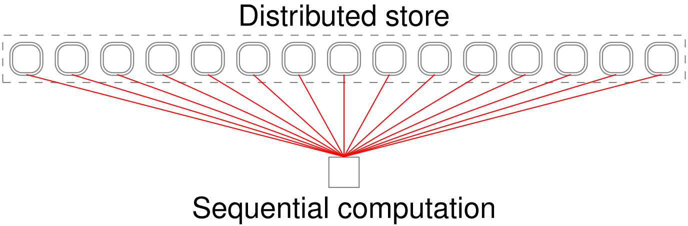
Calls compiled into a sequence of message passing exchanges



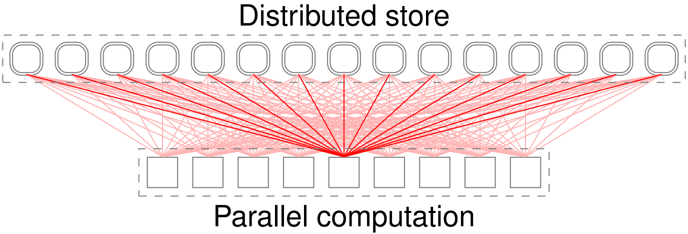
Server array



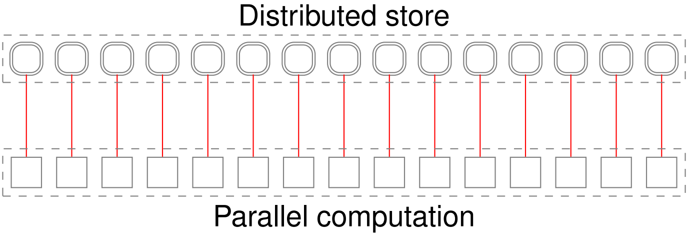
Server array: sequential access



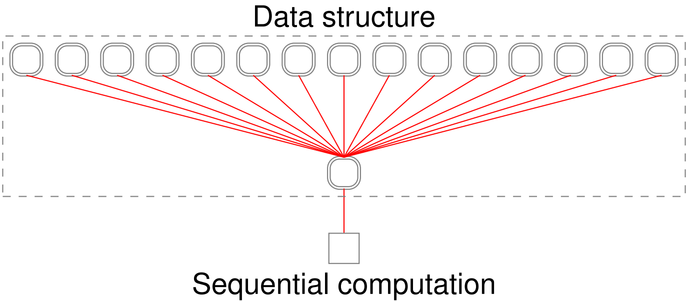
Server array: concurrent access



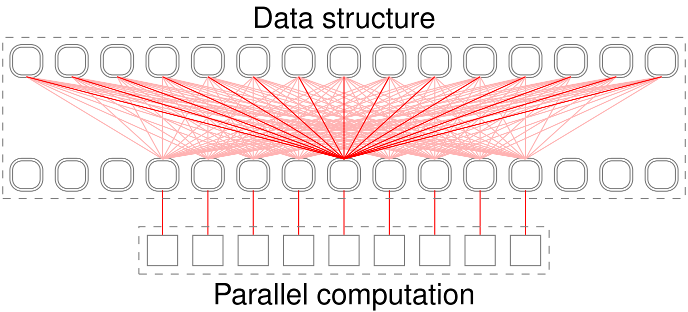
Server array: distributed access



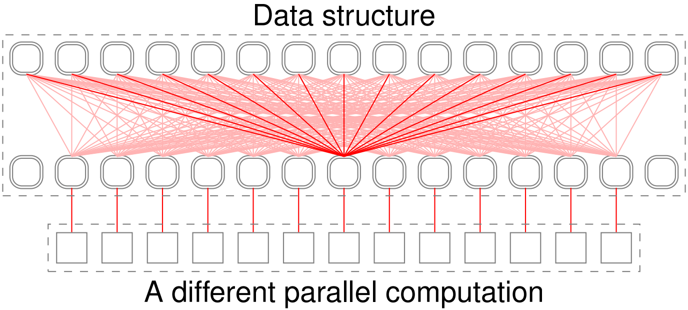
Layered server arrays: sequential access abstraction



Layered server arrays: concurrent access abstraction



Layered server arrays: sequential composition



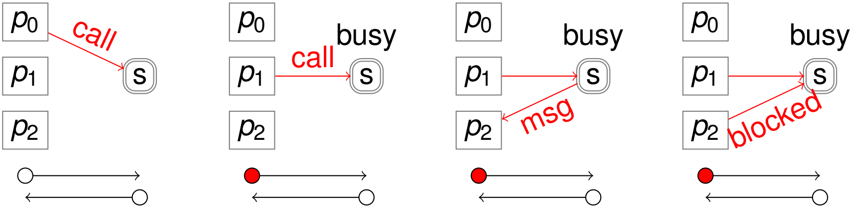
Implementation: avoiding deadlock

An important problem with limited buffering and memory

Many-to-one client-server relations can cause deadlock

If a client or set of clients attempt to access a busy server, the request message will become blocked in the network

If the server then tries to engage in a communication, there may be no available route in the network and it will become blocked



Implementation: avoiding deadlock

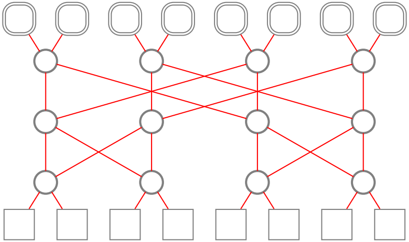
If the number of clients can be determined at compile-time and is small:

- provide sufficient buffering so that a server can record all client requests and service them when it is able to

Otherwise:

- implement many-to-one connections with a bounded-degree routing network
- server requests are taken off the network and queued in memory
- this creates back-pressure, blocking clients but keeping the network clear

Implementation: server routing network



Routing processes (non-deterministically) wait for messages on a set of channels

Messages are routed towards their destination

Can be implemented as a compile-time program transformation

Routing processes can also perform call combining if possible

Example: ray tracer

Problem:

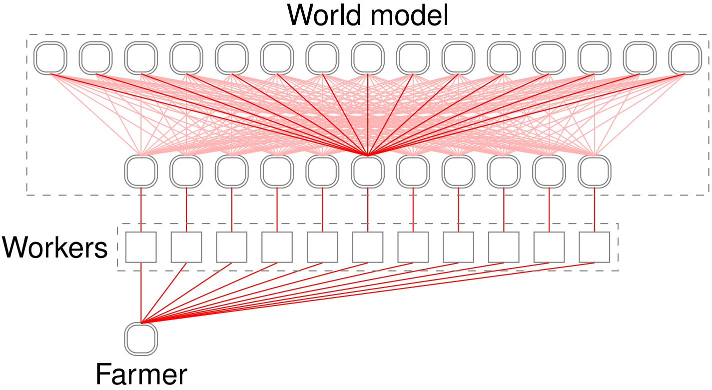
- large world model
- large number of independent tasks calculating ray intersections with unpredictable run time and world model access patterns

Example: ray tracer

Implementation:

- work distributed to a set of worker processes in a task farm
- world model stored in a distributed data structure accessible by all workers
- world model summary structure and object caching essential for good performance

Example: ray tracer



Summary structure replicated among access servers

Access servers also implement caching

To summarise...

Parallellism essential to improve performance and reduce power consumption

We want to employ distributed parallel architecture to scale these

Exciting opportunities and challenges in embedded computing

But current message passing approaches don't support data abstraction

Proposal: we can combine message passing with the concept of a shared resource (server) to separate data and develop abstractions independently of a computation

Any questions?

Abstract machine model

N processing tiles, each with a processor, private memory and communication interface

Each processor able to execute multiple *processes* simultaneously, with mechanisms to create, synchronise and destroy groups of threads

A process can communicate with any other process in the system via a *channel*

Communication channels consist of two *channel ends* that are local to a process

A channel end is connected (unidirectionally) by specifying the unique reference of the destination channel end