A Distributed OpenCL Platform for Cluster Architectures

Matthias Noack
ma.noack@tu-cottbus.de

Brandenburgische Technische Universität
Lehrstuhl Verteilte Systeme / Betriebssysteme

2011-11-11

Outline

b-tu

1. Introduction
   - GPGPU
   - OpenCL
   - GPU clusters
2. Related Work
   - Existing approaches
3. Solution
   - COPE
   - TACO
   - Collective Memory Operations
4. Results
   - Collective Memory Operations
   - OpenCL Application Benchmarks
5. Summary

Outline

1. Introduction
   - GPGPU
   - OpenCL
   - GPU clusters
2. Related Work
   - Existing approaches
3. Solution
   - COPE
   - TACO
   - Collective Memory Operations
4. Results
   - Collective Memory Operations
   - OpenCL Application Benchmarks
5. Summary

General-purpose computing on GPUs (GPGPU)

b-tu

Why?

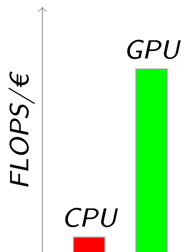
- Speed-up factors from 5 to 100 (vs. CPU)
- More FLOPS per Watt and Euro

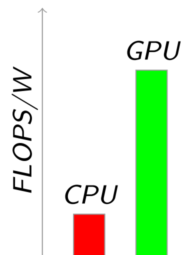
How?

- Highly parallel SIMD architecture
- Lots of arithmetic vector processing units
- Simple control logic
- Relaxed memory consistency models

Where?

- High throughput algorithms with element-wise calculations
- Simple data structures
- Simple control flows





Programming

b-tu

Different APIs:

- OpenCL (vendor and platform independent)
- CUDA (Nvidia only)
- DirectCompute (Windows only)

Similarities:

- Distinguish host and kernel code
- C-like kernel code languages

Differences:

- Supported platforms and devices
- Toolchain
- Host code
- Resource and kernel code management

OpenCL in a Nutshell

b-tu

Target:

- Portable and efficient code

Supports multiple kinds of processors:

- GPUs, CPUs (SSE/AVX), Cell Broadband Engine, ...

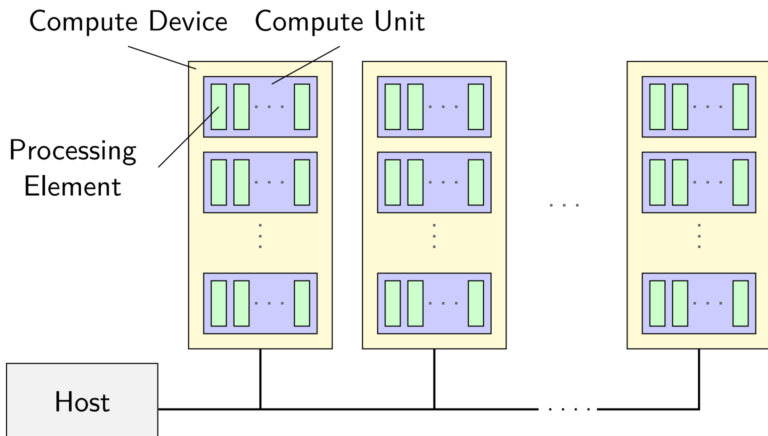
Different implementations:

- AMD, Nvidia, Intel, IBM, ...

Consists of 3 major parts:

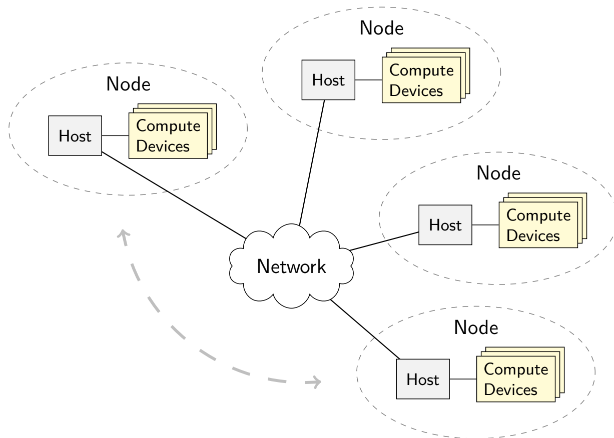
- OpenCL Platform Layer
- OpenCL Runtime Layer
- OpenCL C Programming Language (kernel code)

OpenCL Platform Model



GPU clusters

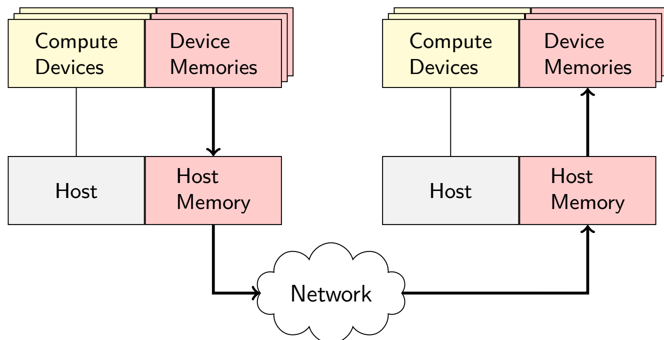
Clusters where all or some nodes have GPUs



GPU clusters

Problems:

- Increased computation power, same network
- Long path for memory transfers
- Keeping compute devices busy
- No programming model



Outline

1 Introduction
- GPGPU
- OpenCL
- GPU clusters

2 Related Work
- Existing approaches

3 Solution
- COPE
- TACO
- Collective Memory Operations

4 Results
- Collective Memory Operations
- OpenCL Application Benchmarks

5 Summary

GPGPU approaches for Clusters

MPI and local API

- The hard way...

CudaMPI

- Adds MPI-like device memory transfers

Shared (Device) Memory

- Lacks performance and scalability

Charm++ integration

- Kernel execution requests with automatic memory transfers
- Lacks flexibility

MOSIX Virtual OpenCL Layer

- Single system image: one host with all devices
- Works with existing OpenCL programmes
- No solution for distributed host code

Outline

1. Introduction
   - GPGPU
   - OpenCL
   - GPU clusters
2. Related Work
   - Existing approaches
3. Solution
   - COPE
   - TACO
   - Collective Memory Operations
4. Results
   - Collective Memory Operations
   - OpenCL Application Benchmarks
5. Summary

COPE: Cluster OpenCL Programming Extensions

b-tu

Idea: Lift the OpenCL C++ API into a global object space

- Global object space is provided by TACO
- OpenCL across node boundaries via RMI
- Simplify programming with object groups

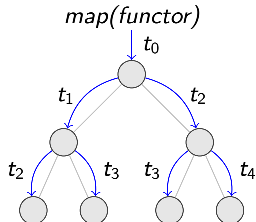
COPE implementation involved:

- Wrapping the whole OpenCL API to make it RMI proof
- Reference counted global pointers
- OpenCL group types
- Group RMIs with groups as arguments
- Collective Memory Operations

TACO: Topologies And COllections

TACO (Topologies And COllections):

- C++ template library for parallel programming
- Supports multiple backends
  - MPI, TCP, Intel SCC, ...
- Distributed objects in a global object space
- RMI via global references and functors
- Collective group operations



Example Code

```
// single objects:
ObjectPtr<MyClass> obj = allocate<MyClass>(node)(ctorArg);
result = obj->invoke(m2f(&MyClass::someMethod, arg)); // RMI
// object groups:
Cyclic mapping; // object to node mapping
GroupOf<MyClass> group(ctorArg, groupSize, mapping);
group.map(m2f(&MyClass::someMethod, arg)); // group RMI
```

Code Example

b-tu

Local OpenCL Code

```
Event event; // event object for synchronisation

// enqueue the kernel and bind the command to the event:
queue.enqueueNDRangeKernel(kernel, NullRange,
    NDRange(n), NullRange, NULL, &event);

event.wait(); // block until kernel execution is finished
```

COPE Code

```
// extract mapping from existing group and create an event group:
GroupMapping mapping(queueGroup);
EventGroup eventGroup(mapping);

// enqueue a group of kernels and bind the command to the event group:
queueGroup.step(gm2f(queueGroup,
    &CommandQueue::enqueueNDRangeKernel,
    multiParam(kernelGroup), NullRange, NDRange(n),
    NullRange, NullEventList, multiParam(eventGroup)));

// initiate remote event-synchronisation and local sync on RMI:
eventGroup.step(m2f(&Event::waitDeferred));
```

Collective Memory Operations

b-tu

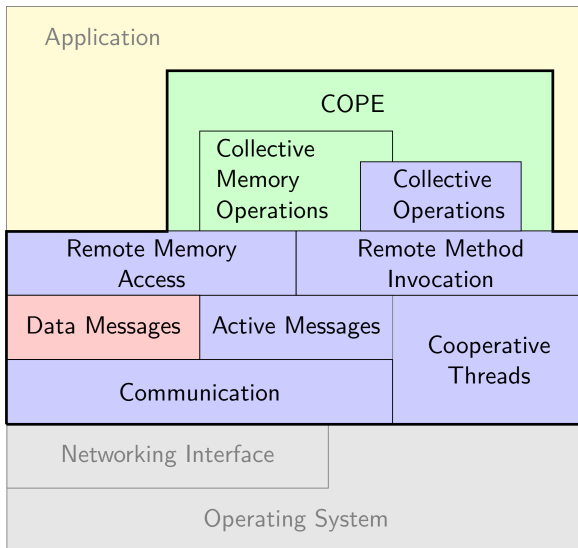
- Independent library for TACO
- Compatible with OpenCL's memory layout
- Provides:
  - Broadcast
  - Scatter
  - Gather
  - All-to-all
  - Neighbour-exchange

Example Code

```
// create group and allocate memory:
Cyclic mapping; // distribution strategy
BroadcastGroup<DataType> bg(dataSize, groupSize, mapping);

// broadcast local or remote data:
bg.broadcast(sourceBuffer);
```

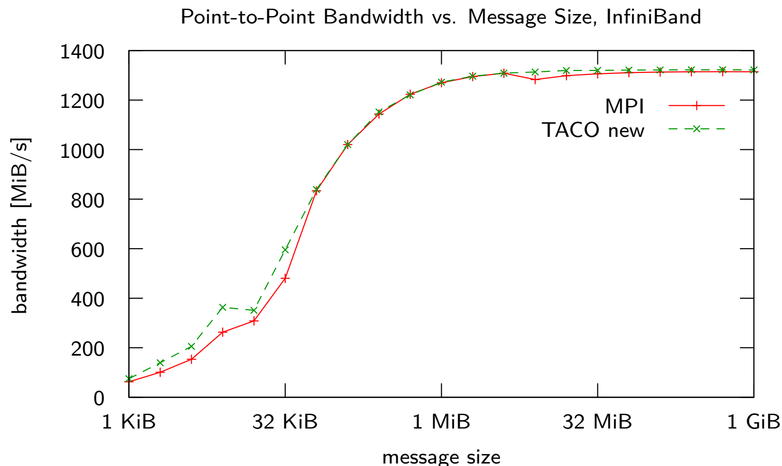
Architecture



Outline

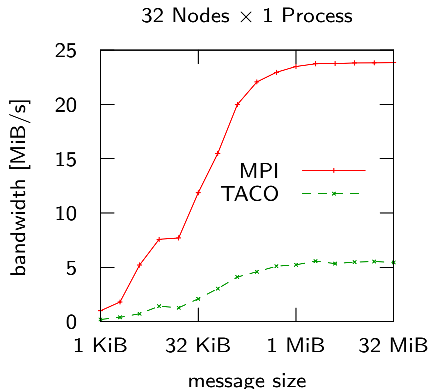
1. Introduction
   - GPGPU
   - OpenCL
   - GPU clusters
2. Related Work
   - Existing approaches
3. Solution
   - COPE
   - TACO
   - Collective Memory Operations
4. Results
   - Collective Memory Operations
   - OpenCL Application Benchmarks
5. Summary

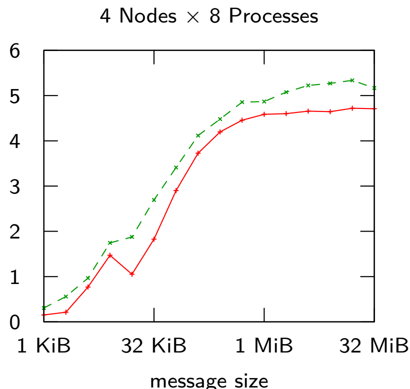
Pure MPI vs. Optimised MPI backend



All-To-All Results

All-to-all Bandwidth vs. Message Size, 32 processes



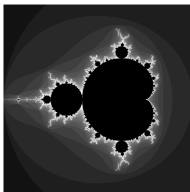


Application Benchmarks

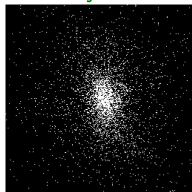
b-tu

Three examples for typical application patterns:

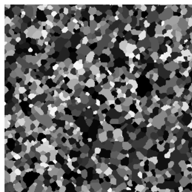
- Mandelbrot



- N-Body



- Cellular automaton



Setting:

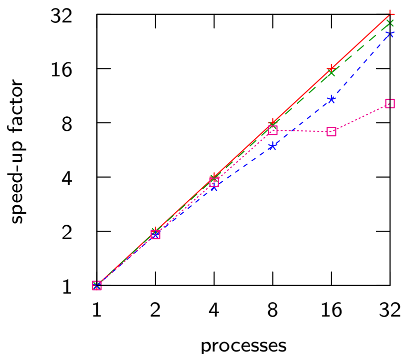
- AMD Opteron cluster, 32 nodes a 8 cores
- OpenMPI, InfiniBand
- OpenCL: AMD APP SDK 2.5
- CPU devices, 1 process per node
- Processes: 1 - 32
- Problem sizes: constant and proportionally growing

Speed-Ups

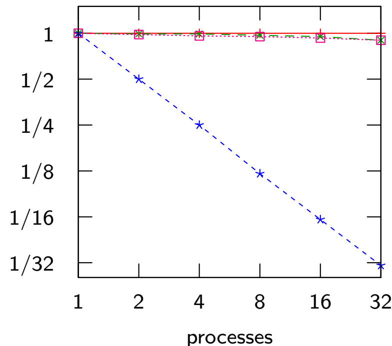
b-tu

- Ideal, $O(n)$
- Mandelbrot, $O(n)$
- N-Body, $O(n^2)$
- Cellular automaton, $O(n)$

Constant Problem Size



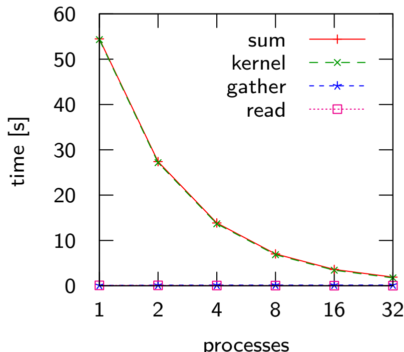
Growing Problem Size

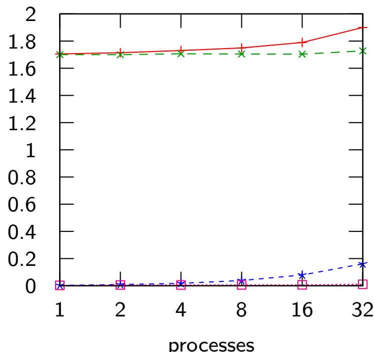


Mandelbrot

- Embarassingly parallel
- Problem size:
  - Constant: 64 mega pixel
  - Growing: 2 mega pixel per process

Mandelbrot Runtime vs. Number of Processes





Overall Results

COPE:

- All applications show good scaling behaviour
- Enough room for faster compute devices
- Overall overhead at most 10 % of theoretical speed-up
- On very large systems: current group argument implementation might become a bottleneck

Collective Memory Operations:

- Scatter and gather perform like MPI
- MPI's broadcast and all-to-all are faster in pure multicast networks

Other:

- Weighted reference counting reduces 98 % of the reference copies to the cost-scale of non-counted copies

Outline

1 Introduction
- GPGPU
- OpenCL
- GPU clusters

2 Related Work
- Existing approaches

3 Solution
- COPE
- TACO
- Collective Memory Operations

4 Results
- Collective Memory Operations
- OpenCL Application Benchmarks

5 Summary

COPE features

b-tu

- One programming model for GPU host code and CPU code
- Most kernel code needs no modification

- Complete OpenCL C++ API features
- Mixing of local OpenCL API and COPE is possible
- Easy porting of existing applications

- Groups effectively reduce programming effort
- Foundation for domain-specific high-level abstractions

Future Work

Technical aspects:

- Improved group argument implementation
- New OpenCL 1.2 standard coming soon

Acquire experience:

- More benchmarks on different GPU clusters
- Connect with potential users
- Port and develop applications

Thank you.

Outline

6 More Code

7 More OpenCL

8 More Benchmark Results

Code Example

Local OpenCL Code

```
// event object for synchronisation
Event event;
// enqueue the kernel and bind the command to the event
queue.enqueueNDRangeKernel(kernel, NullRange,
    NDRange(n), NullRange, NULL, &event);
// block until kernel execution is finished
event.wait();

// Alternative: synchronisation on the queue
queue.finish();
```

Code Example

b-tu

COPE Code

```
// extract mapping from existing group
GroupMapping mapping(queueGroup);
// create a distributed event group with this mapping
EventGroup eventGroup(mapping);
// execute kernel
queueGroup.step(gm2f(queueGroup,
    &CommandQueue::enqueueNDRangeKernel,
    multiParam(kernelGroup), NullRange, NDRange(n),
    NullRange, NullEventList, multiParam(eventGroup)));
// initiate remote waiting on the events and wait
   until all associated commands finish
eventGroup.step(m2f(&CommandQueue::waitDeferred));

// Alternative:  deferred synchronous with Future
Future<void> sync;
eventGroup.step(m2f(&CommandQueue::waitDeferred), sync);
// [do something else here], then re-synchronise
sync.wait();
```

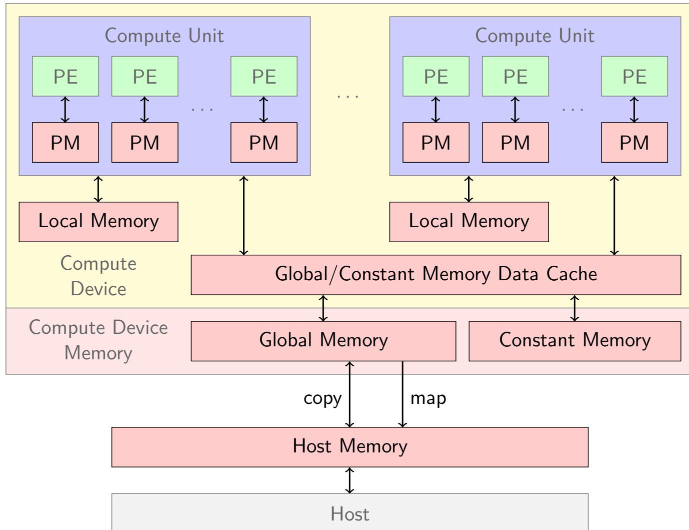
Outline

6 More Code

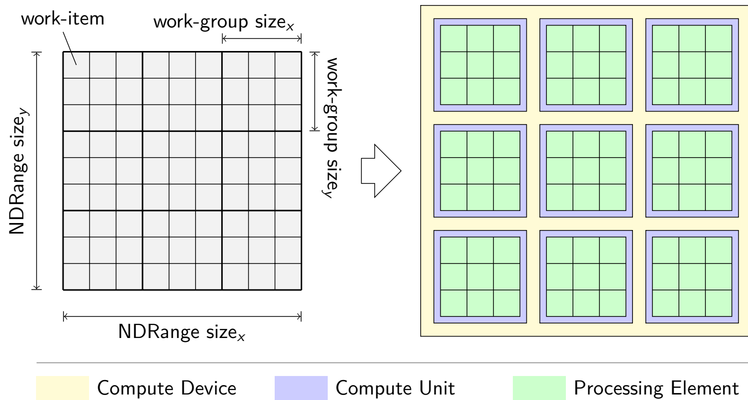
7 More OpenCL

8 More Benchmark Results

OpenCL Memory Architecture



OpenCL NDRange Execution



Outline

6 More Code

7 More OpenCL

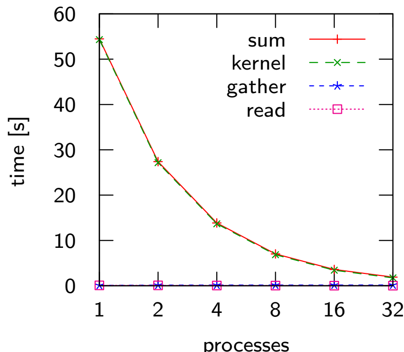
8 More Benchmark Results

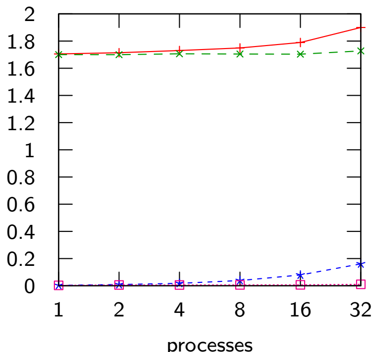
Mandelbrot

b-tu

- Embarassingly parallel
- Problem size:
  - Constant: 64 mega pixel
  - Growing: 2 mega pixel per process

Mandelbrot Runtime vs. Number of Processes

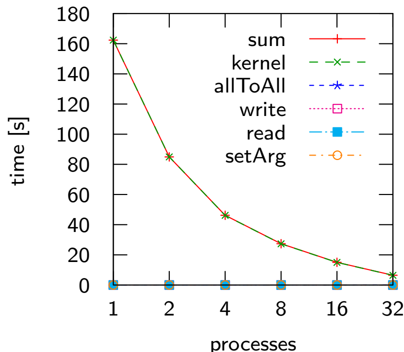


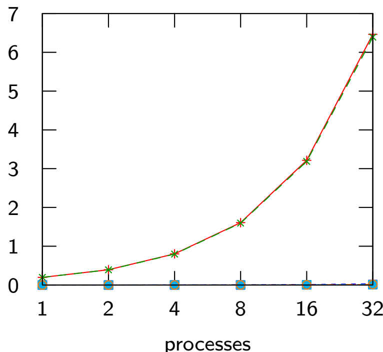


N-Body

- $O(n^2)$, complete replication
- Problem size:
  - Constant: $n = 2^{18} = 262\,144$ bodies
  - Growing: $n = 2^{13} = 8\,192$ bodies per process

N-Body Runtime vs. Number of Processes



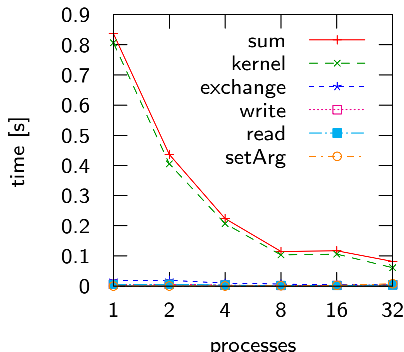


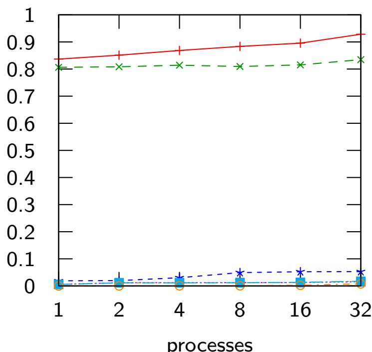
Cellular Automaton

b-tu

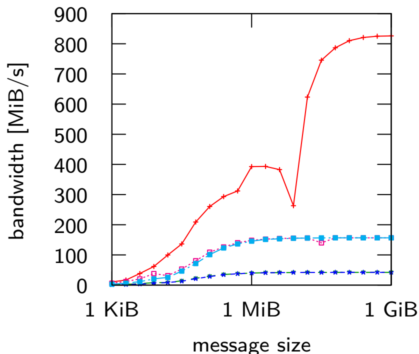
- Neighbour-exchange
- Problem size:
  - Constant: $2^{24} = 256^3 = 16\,777\,216$ cells
  - Growing: $2^{24} = 256^3 = 16\,777\,216$ cells per process

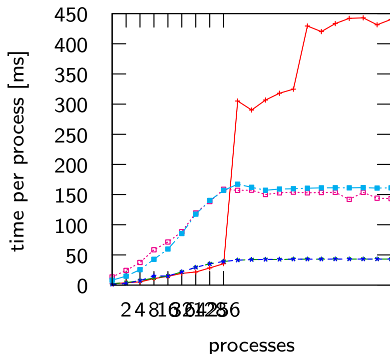
Cell Runtime vs. Number of Processes





Broadcast Bandwidth



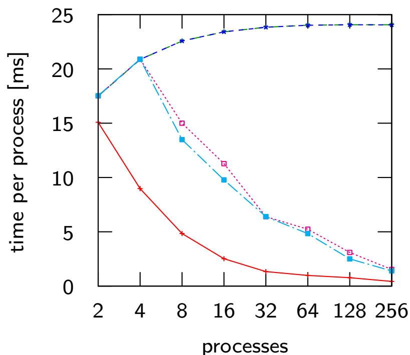


Broadcast Scaling

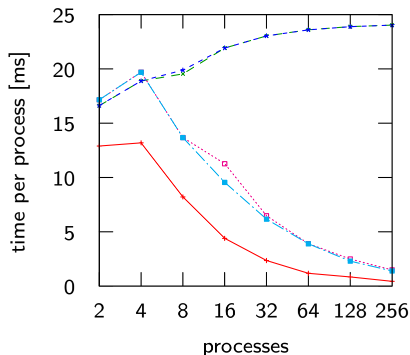
MPI
direct
simple

group pull
group push

Network first, 32 MiB

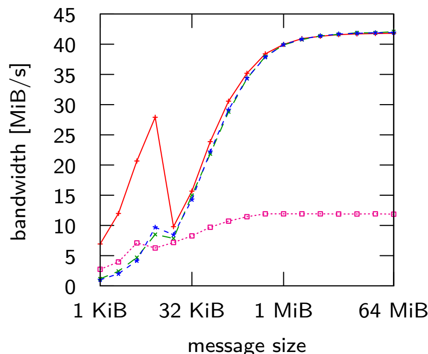


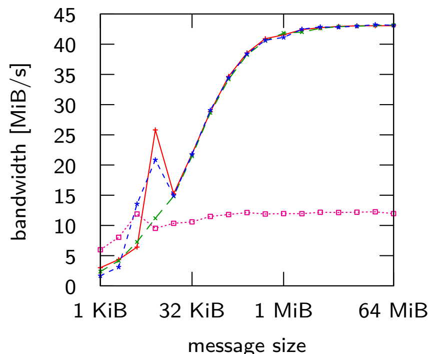
Shared Memory first, 32 MiB



Scatter Bandwidth

b-tu

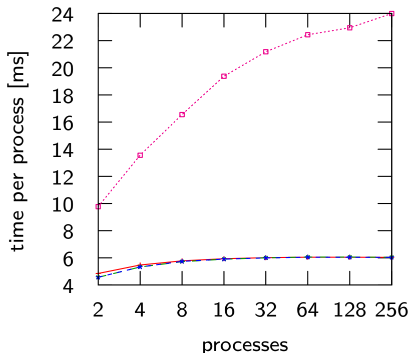




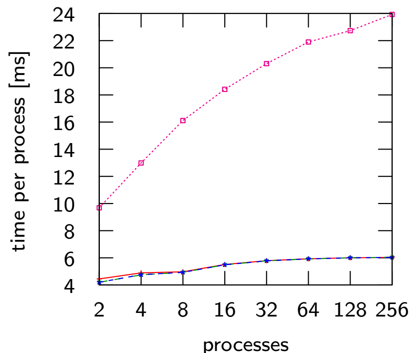
Scatter Scaling

MPI — direct — simple — group

Network first, 8 MiB



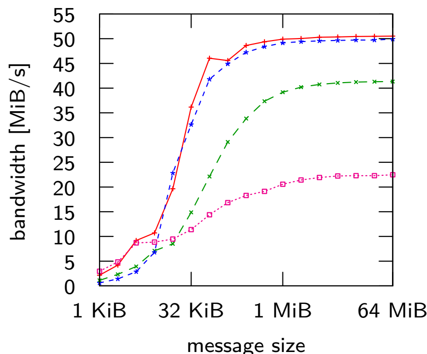
Shared Memory first, 32 MiB

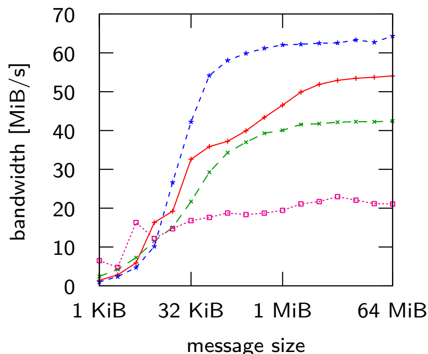


Gather Bandwidth

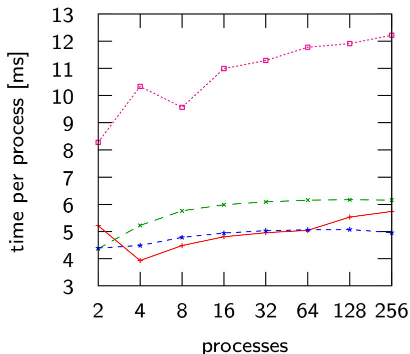
b-tu

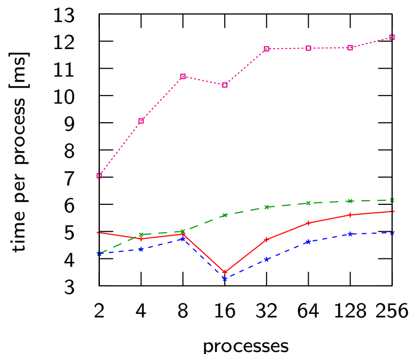
MPI — direct — simple — group





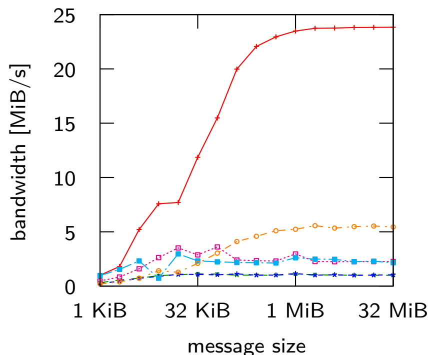
Scatter Scaling

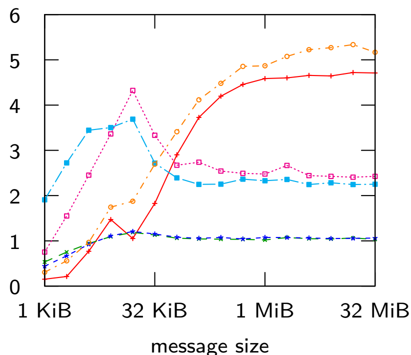




All-To-AllBandwidth

b-tu



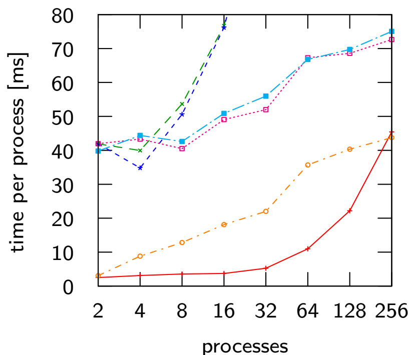


All-To-All Scaling

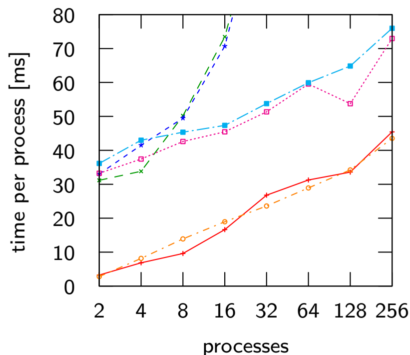
b-tu

MPI
direct
simple
simple Group
group
pipelined

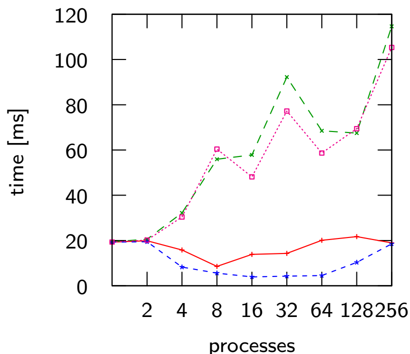
Network first, 8 MiB

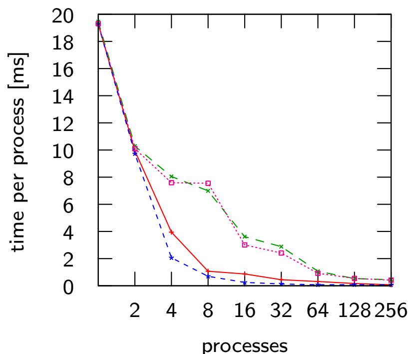


Shared Memory first, 32 MiB

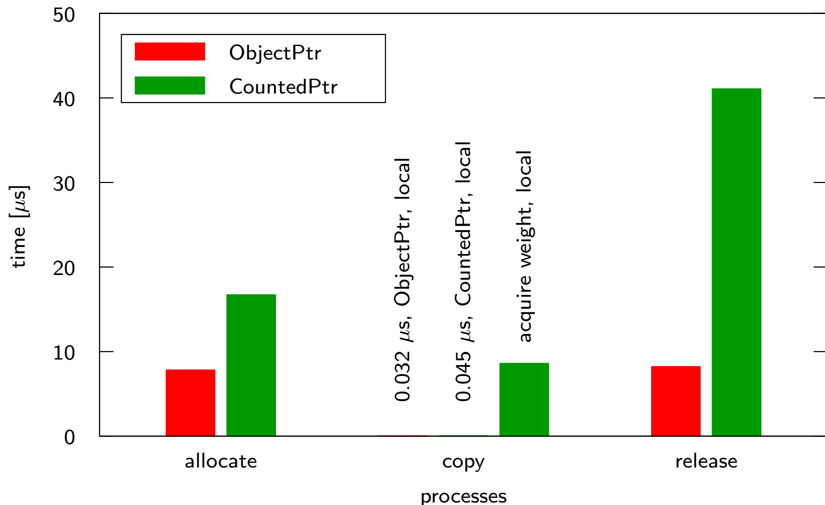


Exchange Scaling

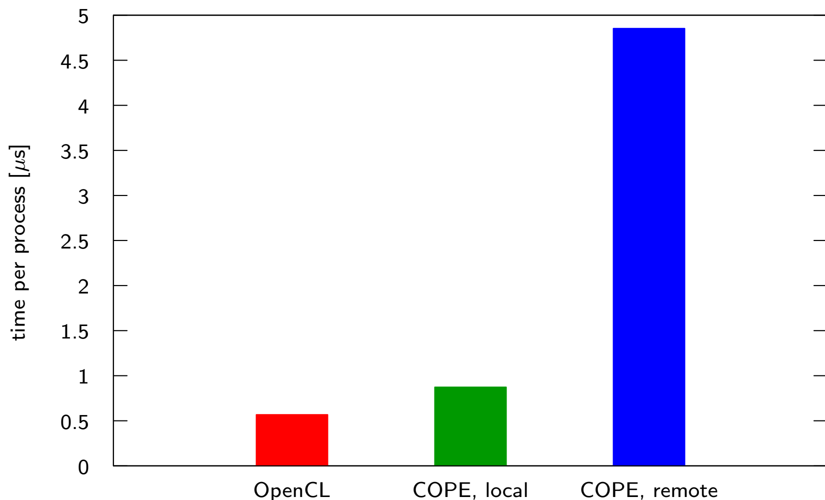




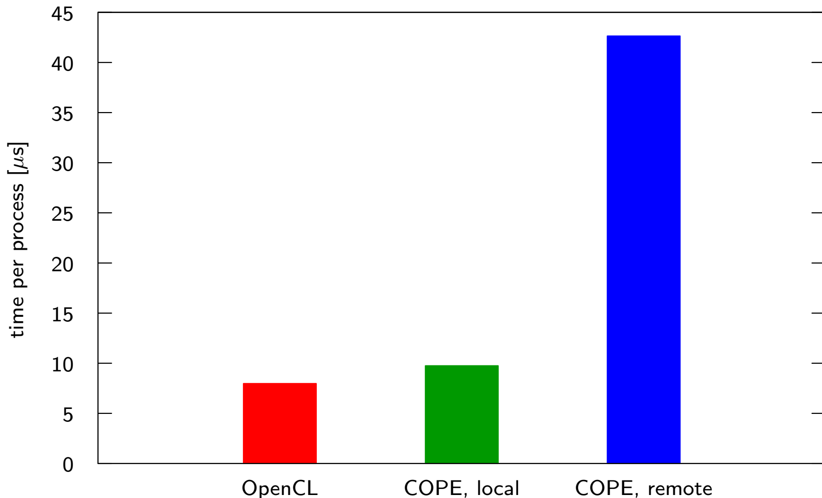
Reference Counting



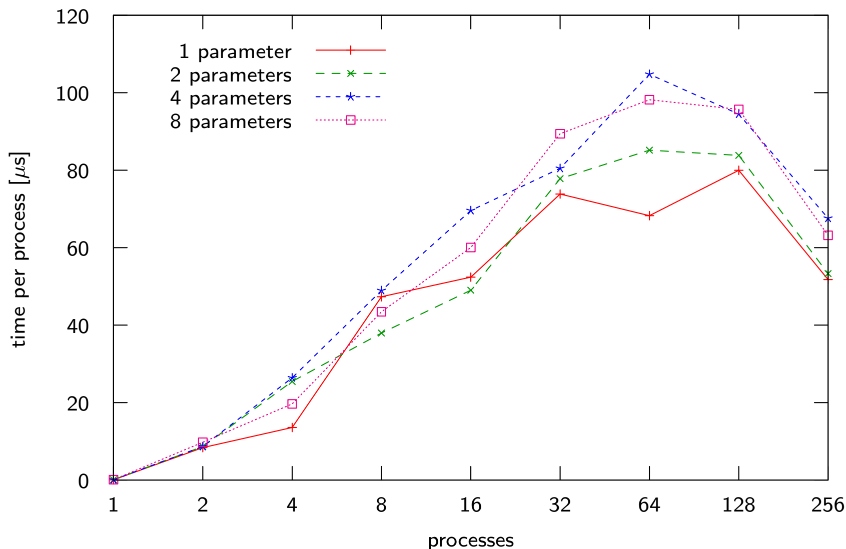
Asynchronous Kernel Invocation Times



Synchronous Kernel Invocation Times vs. Number of Processe



m2f



gm2f