Alembic
Automatic Locality Extraction via Migration

Brandon Holt, Preston Briggs, Luis Ceze, Mark Oskin
Partitioned Global Address Space (PGAS)
Partitioned Global Address Space (PGAS)
Move data or computation?

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *PPOPP '95*, ACM.

† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *OOPSLA '93*. ACM.
Move data or computation?

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In PPOPP ’95, ACM.

† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. OOPSLA ’93. ACM.
Move data or computation?

Move data

Thread

Node 0
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7

Memory
Memory
Memory
Memory
Memory
Memory
Memory

Cores
Cores
Cores
Cores
Cores
Cores
Cores

Move data or computation?

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In PPOPP '95, ACM.
† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. OOPSLA '93. ACM.
Move data or computation?

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *PPOPP ’95*, ACM.

† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *OOPSLA ’93*. ACM.
Move data or computation?

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *PPOPP ’95*, ACM.

† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *OOPSLA ‘93*. ACM.
Move data or computation?

Node 0
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7

Interconnect

Data

Thread

Memory

Cores

move data

move computation

Thread

Memory

Cores

Node 0
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In PPOPP ’95, ACM.

† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. OOPSLA ’93. ACM.
Move data or computation?

Node 0
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6
Node 7

Interconnect

Data

Thread

Cores

Memory

Move data

move data

move computation

Thread

Cores

Memory

Thread migration*

RPC†

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In PPOPP ’95, ACM.
† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. OOPSLA ’93. ACM.
Move data or computation?

Thread

Data

Interconnect

Node 0

Node 1

Node 2

Node 3

Node 4

Node 5

Node 6

Node 7

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In PPOPP ’95, ACM.

† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. OOPSLA ’93. ACM.
Move data or computation?

Move data

Thread

Data

move data

move computation

Interconnect

Node 0

Memory

Cores

Node 1

Data

Thread

Memory

Cores

Node 2

Memory

Cores

Node 3

Memory

Cores

Node 4

Memory

Cores

Node 5

Memory

Cores

Node 6

Memory

Cores

Node 7

Memory

Cores

…

thread migration*

RPC†

explicit (on/at)

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In PPOPP ’95, ACM.

† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. OOPSLA ’93. ACM.
Move data or computation?

- *M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In PPOPP ’95, ACM.*
- †L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. OOPSLA ’93. ACM.

- Move data
- Move computation
- Thread generated by PGAS compilers
- Node 0
- Node 1
- Node 2
- Node 3
- Node 4
- Node 5
- Node 6
- Node 7
- Interconnect
- Data
- Thread migration
- RPC
- explicit (on/at)
Move data or computation?

- Move data or computation?

- Why not automatically move computation?

- Move data
  - move data
  - generated by PGAS compilers
  - Thread
generated by PGAS compilers

- Move computation
  - move computation
  - Why not automatically move computation?

- Interconnect

- Thread
  - move data
  - move computation

- Node 0
  - Memory
  - Cores

- Node 1
  - Memory
  - Data
  - Thread
  - move data
  - move computation

- Node 2
  - Memory
  - Cores

- Node 3
  - Memory
  - Cores

- Node 4
  - Memory
  - Cores

- Node 5
  - Memory
  - Cores

- Node 6
  - Memory
  - Cores

- Node 7
  - Memory
  - Cores

- thread migration*
  - RPC†
  - explicit (on/at)

- * M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *PPOPP ’95*, ACM.
- † L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *OOPSLA ’93*. ACM.
Move data or computation?

Alembic: automatically move computation to reduce communication

- generated by PGAS compilers
- move data
- move computation
- thread migration
- RPC

Why not automatically move computation?

* M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In PPOPP ’95, ACM.
† L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. OOPSLA ’93. ACM.
Alembic

Static optimizing migration algorithm
- Constrained by anchor points
- Greedy heuristic to reduce communication

Implementation for C++ in LLVM

Evaluation
- 6x better than naive compiler-generated communication
- 82% of hand-tuned performance
Alembic

Static optimizing migration algorithm
- Constrained by anchor points
- Greedy heuristic to reduce communication

Implementation for C++ in LLVM

Evaluation
- 6x better than naive compiler-generated communication
- 82% of hand-tuned performance
algorithm
algorithm

Locality analysis
- Identify anchor points
- Partition anchors into locality sets

Heuristic region selection
- Divide into regions that minimize communication
- Transform task to migrate at region boundaries
HOPS Benchmark

Node 0

Node 1

Node 2
HOPS Benchmark

Node 0

Node 1

Node 2

A:

count: 0
winner: 

B:

0 0 2

0 0 2

2 0 1

2 0 1

0 1 2

0 1 2
forall(0, B.size, [A,B](long i) {
    Counter global* a = A + B[i];
    long prev = fetch_add(&a->count, 1);
    if (prev == 0)   // first to arrive
        a->winner = i; // is winner
});
forall(0, B.size, [A,B](long i) {
  Counter global* a = A + B[i];
  long prev = fetch_add(&a->count, 1);
  if (prev == 0) // first to arrive
    a->winner = i; // is winner
});
HOPS Benchmark

forall(0, B.size, [A,B](long i) {
    Counter global* a = A + B[i];
    long prev = fetch_add(&a->count, 1);
    if (prev == 0) // first to arrive
        a->winner = i;
});

Anchor points
- memory locations are owned by one node
- so memory references are anchored to that node
- these are constraints on the thread's execution
forall(0, B.size, [A,B](long i) { 
    Counter global* a = A + B[i];
    long prev = fetch_add(&a->count, 1);
    if (prev == 0) // first to arrive
        a->winner = i; // is winner
});
algorithm

locality analysis

A

B

i

B[i]

fetch_add(&a->count, 1)

a->winner = i
Locality partitioning: *pessimistic value partitioning* (value numbering)
- each anchor starts in its own set
- merge sets if you can prove they are *congruent*
- for locality partitioning: *congruence means on the same node*

```c
fetch_add(&a->count, 1)
a->winner = i
```

Locality partitioning: *pessimistic value partitioning* (value numbering)
- each anchor starts in its own set
- merge sets if you can prove they are congruent
- for locality partitioning: congruence means on the same node

```c
fetch_add(&a->count, 1)
a->winner = i
```

Locality partitioning: *pessimistic value partitioning* (value numbering)
- each anchor starts in its own set
- merge sets if you can prove they are *congruent*
- for locality partitioning: *congruence means on the same node*

```
fetch_add(&a->count, 1)
```
```
a->winner = i
```

Plug in your own locality-congruence rules!

Region selection (heuristics optimization)

```c
[A, B](long i) {
    Counter global* a = A + B[i];
    long prev = fetch_add(&a->count, 1); // first to arrive
    if (prev == 0) // is winner
        a->winner = i;
}
```
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions
(or a DAG of basic blocks) which can
all execute on the same node

```c
[A, B](long i) {
    Counter global* a = A + B[i];
    long prev = fetch_add(&a->count, 1);
    if (prev == 0) // first to arrive
        a->winner = i; // is winner
}
```
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions
(or a DAG of basic blocks) which can
all execute on the same node

**communication cost heuristic:**
function of \# of messages and
message size (continuation size)

```c
[A, B](long i) {
    Counter global* a = A + B[i];
    long prev = fetch_add(&a->count, 1)
    if (prev == 0) // first to arrive
        a->winner = i // is winner
}
```
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions (or a DAG of basic blocks) which can all execute on the same node

**communication cost heuristic:**
function of # of messages and message size (continuation size)

```
a = _ + _
fetch_add(&a->count, 1)
if (prev == 0)
a->winner = i
```
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions (or a DAG of basic blocks) which can all execute on the same node

**communication cost heuristic:**
function of # of messages and message size (continuation size)
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions (or a DAG of basic blocks) which can all execute on the same node

**communication cost heuristic:**
function of # of messages and message size (continuation size)
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions (or a DAG of basic blocks) which can all execute on the same node

**communication cost heuristic:**
function of # of messages and message size (continuation size)
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions (or a DAG of basic blocks) which can all execute on the same node

**communication cost heuristic:**
function of *# of messages* and *message size* (continuation size)
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions (or a DAG of basic blocks) which can all execute on the same node

**communication cost heuristic:**
function of # of messages and message size (continuation size)

![Dependence Graph]
Region selection (heuristic optimization)

**region:**
contiguous sequence of instructions (or a DAG of basic blocks) which can all execute on the same node

**communication cost heuristic:**
function of # of messages and message size (continuation size)
Message cost experiment

Modified HOPS to use extra data after the remote operation

- 1 message
- payload includes the extra data

- 2 messages
- extra data stays
- fixed 8-byte payload
Message cost experiment

Modified HOPS to use extra data after the remote operation

- 1 message
- payload includes the extra data
- extra data stays fixed 8-byte payload

<table>
<thead>
<tr>
<th>Thread state size (bytes)</th>
<th>ops/sec (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>migrate</td>
</tr>
<tr>
<td></td>
<td>blocking</td>
</tr>
<tr>
<td>16</td>
<td>25</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>48</td>
<td>15</td>
</tr>
<tr>
<td>64</td>
<td>10</td>
</tr>
<tr>
<td>80</td>
<td>5</td>
</tr>
<tr>
<td>96</td>
<td>0.005</td>
</tr>
<tr>
<td>112</td>
<td>0.005</td>
</tr>
<tr>
<td>128</td>
<td>0.005</td>
</tr>
</tbody>
</table>

Data size (bytes)
```plaintext
A

B

i

B[i]

a = _ + _

fetch_add(&a->count, 1)

prev

if (prev == 0)

a->winner = i
```
Region selection (heuristic optimization)

```
A
---
B
---
i
---

B[i]

a = _ + _

fetch_add(&a->count, 1)

prev

if (prev == 0)

a->winner = i
```
Region selection (heuristic optimization)

- for each anchor, expand a region as far as possible
Region selection (heuristic optimization)
- for each anchor, expand a region as far as possible

```cpp
fetch_add(&a->count, 1)
prev = 0
if (prev == 0)
a->winner = i
```
Region selection (heuristic optimization)

- for each anchor, expand a region as far as possible
- at region intersections, compute cost heuristic for the possible choices
Region selection (heuristic optimization)

- for each anchor, expand a region as far as possible
- at region intersections, compute cost heuristic for the possible choices

```c
fetch_add(&a->count, 1)

if (prev == 0)
    a->winner = i
```
Region selection (heuristic optimization)
- for each anchor, expand a region as far as possible
- at region intersections, compute cost heuristic for the possible choices

```c
fetch_add(&a->count, 1)
if (prev == 0)
a->winner = i
```
Region selection (heuristic optimization)
- for each anchor, expand a region as far as possible
- at region intersections, compute cost heuristic for the possible choices
- evaluate intersections pair-wise, greedily choose the best in each case
Region selection (heuristic optimization)
- for each anchor, expand a region as far as possible
- at region intersections, compute cost heuristic for the possible choices
- evaluate intersections pair-wise, greedily choose the best in each case

```c
if (prev == 0)
    a->winner = i
fetch_add(&a->count, 1)
```
Transform thread to migrate at region boundaries

- create continuations for values that cross regions, and pack them into active messages
Transform thread to migrate at region boundaries
- create continuations for values that cross regions, and pack them into active messages

```c
[A,B](long i) {
    Counter global* a = A + B[i];
    long prev = fetch_add(&a->count, 1);
    if (prev == 0) // first to arrive
        a->winner = i; // is winner
}
```
Transform thread to migrate at region boundaries

- create continuations for values that cross regions, and pack them into active messages

```
[A,B](long i) {
    migrate(node(B+i), _);
}

[A,B,i] {
    Counter global* a = A + B[i];
    migrate(node(a), _);
}

[a,i] {
    long prev = fetch_add(&a->count, 1);
    if (prev == 0) // first to arrive
        a->winner = i; // is winner
}
Transform thread to migrate at region boundaries

- create continuations for values that cross regions, and pack them into active messages

```c
[A,B](long i) {
    migrate(node(B+i), _);
}
[A,B,i] {
    Counter global* a = A + B[i];
    migrate(node(a), _);
}
[a,i] {
    long prev = fetch_add(&a->count, 1);
    if (prev == 0)  // first to arrive
        a->winner = i;  // is winner
}
Alembic

Static optimizing migration algorithm
- Constrained by anchor points
- Greedy heuristic to reduce communication

Implementation for C++ in LLVM

Evaluation
- 6x better than naive compiler-generated communication
- 82% of hand-tuned performance
Implementation

C++ extensions to support global pointers

Anchor point / locality partitioning analysis pass

Region selection and continuation-passing transform pass
Alembic

Static optimizing migration algorithm
- Constrained by anchor points
- Greedy heuristic to reduce communication

Implementation for C++ in LLVM

Evaluation
- **6x better** than naive compiler-generated communication
- 82% of hand-tuned performance
Benchmarks
- Ported Grappa applications (irregular, data-intensive, ...)
  - BFS
  - Pagerank
  - Connected Components
  - Intsort

Performance (12 nodes)
- naive put/get compiler-generated communication
- hand-tuned migration decisions
- Alembic-generated migrations
Benchmarks
- Ported Grappa applications (irregular, data-intensive, ...)

BFS
Pagerank
Connected Components
Intsort

Performance (12 nodes)
- naive put/get compiler-generated communication
- hand-tuned migration decisions
- Alembic-generated migrations

```c
GlobalHashSet symmetric* set;
Graph symmetric* g;

void explore(VertexID r, color_t color) {
    Vertex global* vs = g->vertices();
    phaser.enroll(vs[r].nadj)
    forall<async>(adj(g,vs+r), [=](VertexID j){
        auto& v = vs[j];
        if (cmp_swap(&v.color, -1, color)){
            spawn([=]{ explore(j, color); });
        } else if (v.color != color) {
            Edge edge(color, v.color);
            set->insert(edge);
            phaser.complete(1);
        }
    });
    phaser.complete(1);
}
```
Benchmarks

- Ported Grappa applications (irregular, data-intensive, ...)

BFS
Pagerank
Connected Components
Intsort

Performance (12 nodes)

- naive put/get compiler-generated communication
- hand-tuned migration decisions
- Alembic-generated migrations

```c
GlobalHashSet symmetric* set;
Graph symmetric* g;

void explore(VertexID v, color_t color) {
    Vertex global* vs = g->vertices();
    phaser.enroll[vs[v].nadj]
    forall<async>(adj(g,vs+v), [=](VertexID j){
        auto& v = vs[j];
        if (cmp_swap(&v.color, -1, color)){
            spawn[=]{ explore(j, color); };
        } else if (v.color != color) {
            Edge edge(color, v.color);
            set->insert(edge);
            phaser.complete(1);
        }
    });
    phaser.complete(1);
}
```
Evaluation

- BFS
- Pagerank
- Connected components
- Intsort

Performance (MTEPS)

Data moved (GB)

Performance (MOPS)

put/get | alembic | manual

Better
Alembic

Algorithm to make automatic migration decisions
- Analyze locality by partitioning anchors
- Greedy optimization to reduce communication cost heuristic

LLVM implementation for Grappa C++

Performance — near hand-tuned, much better than PGAS baseline