Bale

Kernels for irregular parallel computation
(Not a benchmark)

Jason DeVinney (IDA Center for Computing Sciences) and John Gilbert (UC Santa Barbara)

20 years ago, parallel programming was fun

Jason learned parallel programming on a Cray T3E.

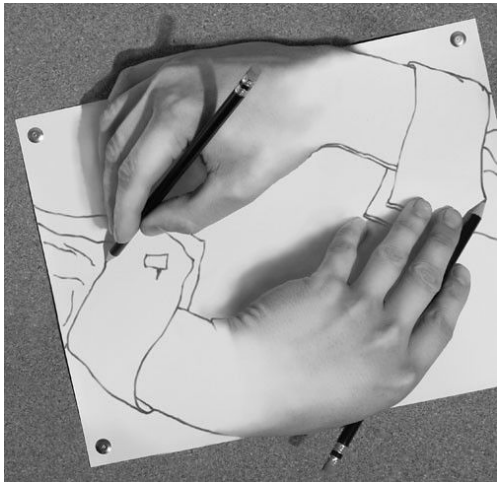
John taught parallel programming on a Cray X1.

Why was it so much fun?

Maybe we had the best of both worlds …

Memory models

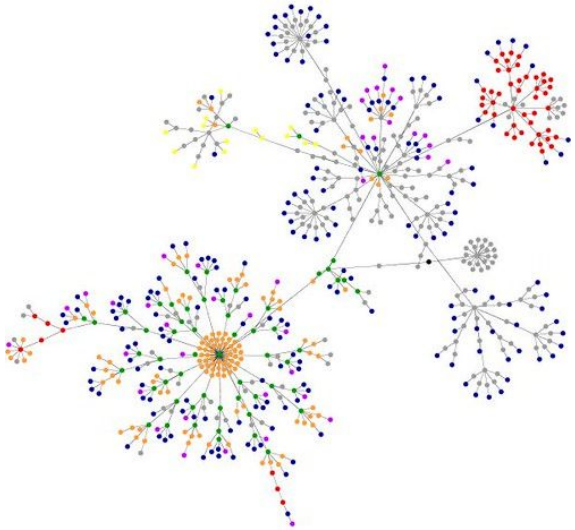
Shared memory is a natural fit for many parallel algorithms.



Distributed memory is how you can scale to a big machine.

Partitioned global address space

UPC gave us a controlled illusion of shared memory:
You can read and write anywhere, but you know
local accesses are faster. So the rule was:



Operate locally as much as possible,
and when you can't, don't sweat it;
use fine-grained communication.

For algorithms on distributed irregular sparse matrices and graphs, this can be awesome, *provided …*

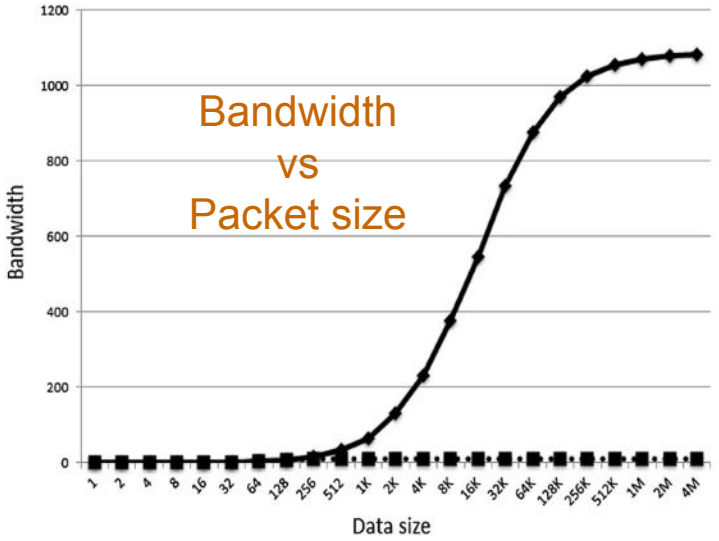
… you take care laying out data structures so simple accesses are local, and …

… your machine is not too bad at fine-grained global communication
(like the T3E and X1).

Aggregation: a rude awakening

10 or 12 years ago, we'd lost our T3E's and X1's.

Falling for the shared-memory illusion was no longer viable if you cared about performance (even on an **actual** shared memory machine!)



In some irregular algorithms, processors can save up their fine-grained reads/writes until they have "enough," and then execute them all at once.

We call this "aggregation," and on most HPC interconnects it wins big.

We quickly realized the power of communication aggregation.

Or perhaps, rather, we realized how pathetic things got if you didn't aggregate.

We wrote a library

We had mountains of code written in UPC in that “just pretend it’s shared memory and everything will be fine” way. And it was no longer fine.

Writing the code to set up the buffers for even a basic aggregator is tricky and easy to get wrong.

After doing it a couple of times we knew we needed a library.

So we wrote **exstack**.

(No relation to the DOE Xstack program).

A simple code...

Many random look-ups from a distributed table.

In UPC:

```
for(i = 0; i < my_num_things; i++)
    dest[i] = BigTable[index[i]];
```

In SHMEM:

```
for(i = 0; i < my_num_things; i++)
    shmem_get(&dest[i],
              &BigTable[index[i]/NPES],
              1, index[i]%NPES);
```

(index and dest are local arrays, BigTable is a distributed array)

… that code in exstack…

```
while( exstack_proceed(ex, (i==my_n_things))){
  i0 = i;
  // push out requests
  while(i < my_n_things) {
     l_indx = index[i]/NPES;
     pe  = index[i] % NPES;
     if(!exstack_push(ex, &l_indx, pe))
        break;
     i++;
  }
  //send requests
  exstack_exchange(ex);
```

```
  // pull requests, process and push back
  while(exstack_pop(ex, &idx , &fromth)){
    idx  = lBigTable[idx];
    exstack_push(ex, &idx, fromth);
  }
  lgp_barrier();
  // send responses
  exstack_exchange(ex);
  // process responses
  for(j=i0; j<i; j++) {
    fromth = index[j] % NPES;
    exstack_pop_thread(ex, &idx, fromth);
    dest[j] = idx;
  }
  lgp_barrier();
}
```

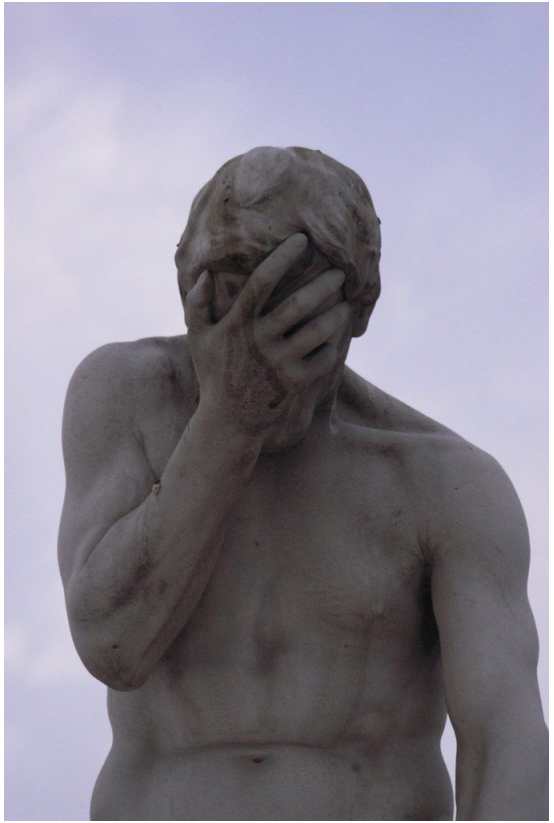
What have we done?

The exstack code is

- wicked fast

Also

- hard to write correctly
- hard to debug
- way too hard to read
- really easy to screw up changes
- hard to maintain in general



Looking for patterns

As we considered new models for aggregation, we went back to see if there were patterns in how we used exstack.

There were two very common patterns.

We call them:

- histogram
- indexgather

But there are lots of other patterns too!

histogram

The first pattern we saw, and the name of the first app in bale.

Pattern 1: Processors send data in a uniform random way to a distributed data structure. Could be updating a table with atomics, or just plain writes.

This is the most common pattern in our codes,
and luckily it is the most natural pattern for aggregators!

indexgather

The second most common pattern we saw, also the name of a bale app.

Pattern 2: All processors read from a distributed data structure.

This is not such a natural fit for aggregators since a processor needs to buffer up *requests* (locations to read); those requests are fulfilled on the remote side and then the *responses* are sent back. Kind of awkward.

A few other bale apps

toposort

- Permute a “morally triangular” matrix to upper triangular form
- Easy linear-time sequential algorithm
- Available parallelism depends heavily on the input data

triangle counting

- Benchmark in HPEC Graph Challenge
- Lots of choices for how to structure parallelism

single-source shortest paths

- Benchmark in Graph500
- Delta-stepping algorithm is a good example of “histogram” communication…
- … but the remote action is user-defined (edge relaxation), not a standard atomic op

bale

bale was initially released in 2018.

bale is **a vehicle for discussion.**

bale currently contains

- exstack, exstack2, and conveyors
- nine mini-apps:
  - histogram
  - indexgather
  - generate random permutation
  - triangle counting
  - toposort
  - single source shortest path
  - sparse matrix transpose
  - permute sparse matrix
  - sparse matrix read/write

Discussion 1: Hammer for all nails

We were blown away by the number of codes whose performance improved dramatically under exstack.

Yes, it required a lot of work, maybe even rethinking of the algorithm, but everywhere we looked we could apply exstack and it made a big difference.

Before exstack, we just *knew* that many of our algorithms absolutely *required* low latency fine grained communication (including atomics).

Now we saw that our algorithms could not only tolerate the higher latency… under exstack they thrive on it!

Discussion 2: Tool for algorithm design

Aggregation makes it harder to express many algorithms.

But for some algorithms, aggregation actually makes it easier.
This was news to us.

Other algorithms can be tweaked to allow a more natural fit with aggregation.

Parallel programmers shouldn’t have to force their algorithms
into a model that is not a good fit.

Discussion 3: Programmability

Writing code with our aggregators in C is **very challenging**. And parallel programming is hard enough as it is.

Future parallel programmers don't deserve this -- and probably won't accept it.

HPC needs to make it easier for new folks if we want to grow.

Can we create a programming model that makes aggregation feel more natural?

Cray-2 interior

Our quest

- We want to make parallel programming easier and more productive.
- We think that aggregation should be a fundamental part of any future programming model.
- We don’t want to be writing exstack / conveyor codes in 10 years.

Recent attempts - Rust

```
1.      {
2.          let mut session2 = Convey::begin(|item: (usize, i64), _from_rank| {
3.          // process response
4.              tgt[item.0] = item.1;
5.              total_returns += 1;
6.          });
7.
8.          {
9.              let mut session1 = Convey::begin(|item: (usize, usize), from_rank| {
10.                 session2.push((item.0, ltable[item.1]), from_rank);
11.                 total_requests += 1;
12.             });
13.
14.             // do the updates
15.             for i in 0..requests / num as u64 {
16.                 let index = die.sample(&mut rng);
17.                 let rank = index % num;
18.                 let offset = index / num;
19.                 session1.push((i as usize, offset), rank);
20.             }
21.             session1.finish();
22.         }
23.         session2.finish();
24.     }
```

* Thanks to Bill Carlson

Recent attempts -- C++

```
1.    bale::buffered::transfer_engine engine;
2.    bale::remote<int64_t[]> table(table_size);
3.    std::vector<int64_t> target(num_requests);
4.
5.    for (auto session = engine();
6.          auto &&[target, index, pe] :
7.            zip(
8.              target,
9.              bounded_random_stream(0, table_size, seed, my_proc),
10.             bounded_random_stream(0, n_procs, seed, my_proc + n_procs)))
11.  {
12.     session.get(target, table(pe)[index]);
13.  }
```

* Thanks to Nick Park

Recent attempts -- C++ with lambdas

```
1.  hclib::finish([=]() {
2.      igs_ptr->start();
3.      for(int64_t i=0; i<l_num_req; i++) {
4.          int64_t index = pckindx[i] >> 16;
5.          int64_t dest_rank = pckindx[i] & 0xffff;
6.          igs_ptr->send(REQUEST, dest_rank, [=]() {
7.              int64_t ret_val = ltable[index];
8.              igs_ptr->send(RESPONSE, sender_rank, [=]() {tgt[i] = ret_val;});
9.          });
10.     }
11.     // Indicate that we are done with sending messages to the REQUEST mailbox
12.     igs_ptr->done(REQUEST);
13. });
```

* Thanks to Vivek Sarkar, Akihiro Hayashi, Sriraj Paul

Recent attempts -- HABU

```
1.  habu_mem_t htable = habu_register_memory(table,sizeof(table[0]), 0);
2.  habu_mem_t htgt = habu_register_memory(tgt, sizeof(tgt[0]), 0);
3.
4.    for(i = 0; i < l_num_req; i++){
5.        habu_fop(htgt, i, MYTHREAD, htable, index[i],
            HABU_CYCLIC_DISTRIBUTION_PE, HABU_GET, NULL, 0);
6.    }
7.
8.  habu_barrier(0);
9.  habu_unregister_memory(htable);
10. habu_unregister_memory(htgt);
```

* Thanks to Nathan Wichmann (HABU is copyrighted by HPE and is under the MIT license)

Recent attempts -- Chapel

```
1.  use CopyAggregation;
2.  forall i in D with (var agg = new SrcAggregator(int)) do
3.    agg.copy(tmp[i], A[rindex[i]]);
```

* Thanks to Elliot Ronaghan

Thank you!

and thanks to…

Bale developers: Bill Carlson, Miller Maley, Phil Merkey, Dan Pryor

App implementors: Akihiro Hayashi, Nick Park, Sriraj Paul, Elliot Ronaghan, Vivek Sarkar, Nathan Wichmann

… and the Monday night hangout group

github.com/jdevinney/bale


github.com/jdevinney/bale

Image Credits

- "HAY BALES NEAR TRIESTE" by mariotto52 is licensed under CC BY-NC 2.0
- "Yes, it really IS a Cray T3E" by stiefkind is marked with CC0 1.0
- "MC Escher" by Conductive is licensed under CC BY-NC
- "NERSC Franklin" is licensed under CC BY-NC-ND 2.0
- "Graph of Der letze Schrei - Blog" by jÖrg is licensed under CC BY-SA 2.0
- "Bandwidth Curve" from Shim, C., Shinde, R. & Choi, M. Compatibility enhancement and performance measurement for socket interface with PCIe interconnections. *Hum. Cent. Comput. Inf. Sci.* 9, 10 (2019). https://doi.org/10.1186/s13673-019-0170-0 under CC 4.0
- "ouch!!!" by black.zack00 is licensed under CC BY-NC-SA 2.0
- "Facepalm" by Suzanne Hamilton is licensed under CC BY-NC-ND 2.0
- "Colorful Pattern" by Photoshop Roadmap is licensed under CC BY 2.0
- "child hammering nails" by Paul A Bischoff is licensed under CC BY 2.0
- "Cray-2" by Jason DeVinney
- "Ferdinand Leeke (German, 1859-1923), 'Parsifal in Quest of the Holy Grail'" by sofi01 is licensed under CC BY-NC 2.0
- "Bales in Saint-Girons June 2022" by John Gilbert