Programming At Extreme Scale

William Carlson
IDA Center for Computing Sciences
APRIL 26, 2021
Extreme Scale Systems: A decades long trend towards local computation

1992 T3E:
Global: 0.2B/s/node
Local: 0.8B/s/node
Ops: 1.2B/s/node

2019 Summit:
Global: 0.025T/s/node
Local: 1T/s/node
Ops: 42T/s/node
Extreme Scale systems: better bandwidth

...But harder to use

- Latency: the time for a single message to get to its destination
  - Is limited by design (pipeline stages) and physics (speed of light)
  - Neither of these has changed much in the last few decades 😊

- Overhead: the computation consumed by communication
  - Is limited by design (we are often using commodity processors)
  - Fortunately, this is getting better due to the wealth of local computation
When communication gets harder...

- **Avoid it**: choose to do computation which don’t need much communication
  - Sometimes works when you have a choice
- **Endure it**: if we are using any part of the system fully, that should be acceptable
- **Reduce it**: develop new algorithms which need less communication
  - This often requires significant human effort
- **Optimize it**: use what we have more effectively
  - This often requires great care in both data layout and program coordination
Our chief weapon is increasing message size

- Organize computation to send larger blocks of data
  - Works well in “blocked” algorithms
- Aggregate many small blocks to create larger messages
  - Has been done ad-hoc for many years
  - We have been experimenting with more organized approaches
Our chief weapons are increasing message size and moving computation to data.

- Increasing the semantic content of a message makes it more efficient
  - “Increment a remote value” is hugely better than get, increment, put
    - Half the number of messages
    - And no data races, if done right!
  - The more complexity you can include, the better it gets
    - “Insert value in hash table”, for example
- Actor model of computation may be a useful theoretical basis
Conveyors:

a library for aggregation and moving computation

- Ties together remote computation and data type with a communication channel
- Efficiently sorts communication elements based on destination
  - Takes advantage of local thread-level parallelism when available
- Efficiently delivers items to be processed to remote nodes
- Scales to large numbers of nodes with efficient memory usage
- Most effective when there is enough parallelism to hide large latencies of sorting
  - A surprising number of algorithms fit this model!
- Available at [https://github.com/jdevinney/bale](https://github.com/jdevinney/bale)
Amongst our chief weapons are ... Talented people!

- Creating programs at extreme scale is very labor intensive
  - Often 10x the code for an extreme scale program
- Creating programs at extreme scale takes a special talent
  - Often developed over many years
- And the tools for these talented people are often not the best
  - Because extreme scale is niche market
Where do we go from here?

• We are getting huge quantities of local operations
• This makes programs which need non-local interaction relatively more difficult
• We have some handy tools which have helped us
• And we have some great people trying hard to be productive at extreme scale

How can we use these resources to win at extreme scale?
Productivity = \frac{Value}{Effort}

- Value at extreme scale should be thought of as
  - Output for a given computational resource applied
    - That output must achieve the goals of the person who invoked it
- Effort at extreme scale should be thought of as
  - Time spent creating the program plus improving it
- Increasing productivity seems obvious: Increase Value, Reduce Effort
  - But many attempts either increase both or reduce both
We are pretty good at increasing value.

Single word increments get full injection bandwidth at extreme scale on almost any system.
We are pretty good at increasing value.

Single word increments get full injection bandwidth at extreme scale on almost any system.

```c
int status = EXIT_FAILURE;
convey_t* conveyor = convey_new(SIZE_MAX, 0, NULL, convey_opt_SCATTER);
if (!conveyor) { printf("ERROR: histo_conveyor: convey_new failed!\n"); return (-1.0); }

ret = convey_begin(conveyor, sizeof(int64_t));
if (ret < 0) { printf("ERROR: histo_conveyor: begin failed!\n"); return (-1.0); }

lgp_barrier();
tm = wall_seconds();
i = 0;
while (convey_advance(conveyor, i == data->l_num_ups)) {
  for (; i < data->l_num_ups; i++) {
    col = data->pckindx[i] >> 20;
    pe = data->pckindx[i] & 0xffffffff;
    assert(pe < THREADS);
    if (!convey_push(conveyor, &col, pe))
      break;
  }
  while (convey_pull(conveyor, &pop_col, NULL) == convey_OK) {
    assert(pop_col < data->lnum_counts);
    data->lcounts[pop_col] += 1;
  }
}
```
We are pretty good at increasing value

Single word increments get full injection bandwidth at extreme scale on almost any system
We are pretty good at increasing value. Single word increments get full injection bandwidth at extreme scale on almost any system.

```c
int status = EXIT_FAILURE;
convey_t* conveyor = convey_new(SIZE_MAX, 0, NULL, convey_opt_SCATTER);
if(!conveyor){printf("ERROR: histo_conveyor: convey_new failed!\n"); return(-1.0);} ret = convey_begin(conveyor, sizeof(int64_t));
if(ret < 0){printf("ERROR: histo_conveyor: begin failed!\n"); return(-1.0);} lmp_barrier();
tm = wall_seconds();
i = 0UL;
while(convey_advance(conveyor, i == data->l_num_ups)) {
  for(; i < data->l_num_ups; i++){
    col = data->pckindx[i] >> 20;
    pe = data->pckindx[i] & 0xffff;
    assert(pe < THREADS);
    if(!convey_push(conveyor, &col, pe))
      break;
 }
while(convey_pull(conveyor, &pop_col, NULL) == convey_OK){
  assert(pop_col < data->lnum_counts);
  data->lcounts[pop_col] += 1;
}
```
But not so good at reducing effort

- Maybe the code could have been
  - `for (i=0; i<N; i++) data->counts[data->pckindex[i]] += 1;`
- But that would have had much less Value
- Side comment on atomic operations and compilers
  - We really like “atomic operations” which would do this easily
  - But often we want something different than the set of ”operations” offered
  - We also have seen compilers that can do some things like this automatically.
There is hope

• A number popular programming languages are developing an interesting set of features
  • Closures which can package up functionality
  • Support for asynchronous operations like futures and promises
  • Type/Object systems which can provide cleaner interfaces
• There is cost associated with these approaches, but it is mostly “local”
  • And we have resources available to cover them!
• Can we adopt these to get both increased value and reduced effort?
Rust

- First introduced in 2010, but really came to popularity in 2018
- Designed for systems programming, like C, so performance is at its core
  - First developed at Mozilla intended for browser.
  - Strict memory tracking and safety, no garbage collection because no garbage
  - Safe concurrency (thread level)
  - “Monomorphization” creates a compile-time specialized function for generic interfaces
- Asynchronous programming support added in 2019
  - An “async” function can “await” the completion of a blocking operation
Rust + Conveyors

- Our experimental approach to productive extreme scale programming
- Uses the OpenShmem1.4 library which is widely supported and fast
  - Rust interface allows safe creation of shared objects across the system
- Adapts the Conveyor API to Rust
  - Allows the creation of a “session”, monomorphized to a data type and remote function
  - Also adds value-based collective functions $x = \text{Convey}::\text{reduce}_\text{sum}(42.0)$;
- Performance looks good!
- https://github.com/wwc559/convey
A Step in the right direction

Single word increments *should* get full injection bandwidth at extreme scale on almost any system.
A Step in the right direction

Single word increments *should* get full injection bandwidth at extreme scale on almost any system.

```rust
convey.simple(
    0..updates).map(|x| convey.offset_rank(die.sample(&mut rng)),
    |item: usize, _from_rank| {
        local[item] += 1;
        total_updates += 1;
    },

```
A Step in the right direction

Single word increments should get full injection bandwidth at extreme scale on almost any system.
Conclusions and next steps

• We feel we are beginning to gain traction on the fundamental concept of adapting Rust to the extreme scale environment (others are working on this too)

• For codes with frequent communication, available local compute is plenty to provide for the overheads involved.

• Rust is not the only answer, in fact these techniques should work well in JavaScript, Modern C++, Python, etc.
Media Credits

- Slide 2 T3E: https://en.wikipedia.org/wiki/Cray_T3E#/media/File:T3E-900t.jpg CC BY-SA 2.5