Programming on the Lucata data-first architecture

Jason Riedy

28 January 2022

Lucata Corporation
(Atlanta branch)
*Everything here is **my** opinion.*

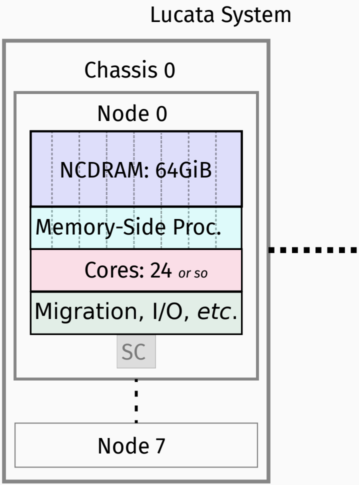
Underserved Application: Parallel Graph Analysis

Known issues:

- Scattered memory access w/ small(?) seq. bursts
  - Cache lines provide fraction of avail. BW
  - Prefetechers fire up then mis-predict
- Large bursts (high degree) $\Rightarrow$ load imbalance
  - Combined with 90% diameter ≤ 8.
  - Plenty of load-balancing pre-processing...
- Streaming: The graph is changing.
  - Pre-processing can hurt where and when changes are interesting.

CPU+cache systems have one set of coping mechanisms. GPGPUs / flex. vectors another. *And Lucata has yet more...*

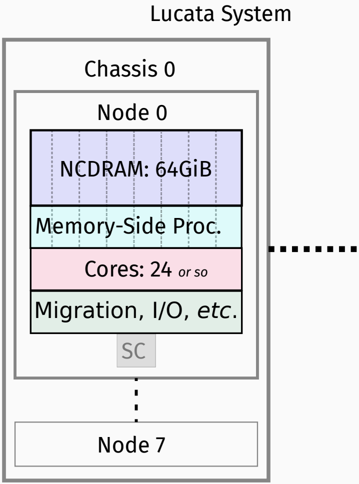
The Lucata Pathfinder PGAS architecture



Four chassis system is a 2 TiB NSF CCRI resource at crnch.gatech.edu

- **Optimized for weak locality**
  - Scattered jumps and seq. access
- Stationary core for OS per node + SSDs
- Hardware partitioned global address space (PGAS) with a twist
  - Plenty of network BW, low latency
  - Details in a moment…
- Multithreaded multicore LCE (or GC)
  - Currently 1536 threads per node, 12k per chassis, 50k per 4 chassis.
  - “Helps” with load balance
  - No cache.

Lucata's PGAS Twist for Weak Locality

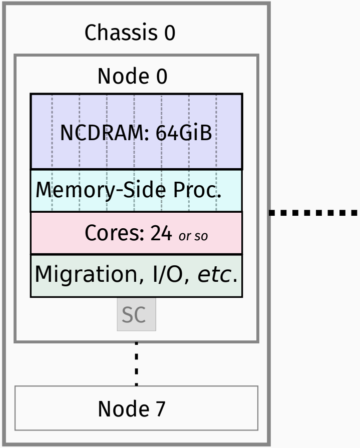


Four chassis system is a 2 TiB NSF CCRI resource at crnch.gatech.edu

- Threads write **remotely**, always read **locally**
- Writes: 8 Memory-side processors (MSPs)
  - Writes+ don't touch the cores.
  - Handle some arithmetic ops. (FPADD)
  - Deep queue, no control flow
- Reads ⇒**migrate**⇐.
  - *Hardware*: Remote read ⇒ package and send the thread context
  - Read latency is local up to migration.
  - **Control flow depends on reads.**
  - Contrast with Tera MTA / Cray XMT: Need far fewer threads, far less network bandwidth.

Programming the Beast: Not Painful.

Lucata System



Chassis 0

Node 0

NCDRAM: 64GiB

Memory-Side Proc.

Cores: 24 *or so*

Migration, I/O, *etc.*

SC

Node 7

Four chassis system is a 2 TiB NSF CCRI resource at crnch.gatech.edu

- PGAS: Read and write directly. Everywhere.
- Memory *views* implemented in hardware
    - Intra-node `malloc`, node-striped “`mw_malloc1d`”, node-blocked “`mw_malloc2d` ”...
    - Implemented by pointer bitfields
- Fork/join parallelism: Cilk+ **+** extensions
    - Fast: Spawning a thread ≈ function
    - Composes: “Serial elision.”
- Reached the middle of the Graph500 BFS list on scale 31 with little effort.
    - Scalable: Buy more, scale more.
    - `https://lucata.com/resources/benchmarks/`

Ok, a Little “Painful”

Think differently:
It’s about the data, not the execution units.

- Algorithms typically need working space.
  - Decide amount and use *that* to guide parallelism.
  - So scale according to **memory**!
  - Kinda obvious, but amazing how few frameworks think that way. *side-eye at ECP*
- Identifying not-quite-locality
  - Naïve analysis kernels may migrate excessively.
  - Need aggregation-ish ⇒ working space

Sketchy Example: Sparse Matrix Product

```
class SpGEMM {
// Bind ”global” things and allocate striped workspace
// in the constructor, then...
void operator() (size_t i) {
 auto A_i = A[i];
 const size_t A_i_deg = A_i.size();
 if (A_i_deg == 0) return;

  for (auto [k, a_i_k] :
               jx_iter_adapt<TypeA>(A_i)) {
     const size_t B_k_deg = b_row_buf.set_row(B, k);
     if (0 == B_k_deg) continue;

     while (auto b_row = b_row_buf.fetch()) {
          for (auto [j, b_k_j] : b_row) {
               TypeC val;
               mult(val, a_i_k, b_k_j);
               c_row.accumulate(j, val, add);}}}
     // Build the C_i block.
     c_row.fill_block(C_i);
     REMOTE_ADD ((long*)&nvals, C_i.size());}}; //<= A freebie!
// And eventually... (work-in-progress)
lucata::apply(SpGEMM{C, A, B});
```

Fun Is in the Details

- `lucata::apply(SpGEMMC, A, B);`
  - Ok, where does the object live? Migrating on de-referencing `*this` for members?
- Goes back to *views* and *replication*.
  - Getting this right is critical but subtle.
  - Very work-in-progress, but at least C++ makes it library-level.
  - For C, structures as arguments are clear.
  - An extension, `cilk_spawn_at`, moves the call frame to be co-located with an address.

Program the data...

More Fun...

- "Atomic" operations
    - Any read migrates.
    - So an `fetch_and_add` migrates.
    - But a simple `REMOTE_ADD` does *not*!
- And never use `NODE()`...
    - Always use data locations, not processing locations.
    - Using `NODE()` can lead to subtle surprises.
    - `b_row_buf` in SpGEMM:
        - My first version compared node numbers...
        - So the launch location was critical.
        - Really wanted to test if pointers were co-located, not where the computation lives.

And Opportunities...

- Global addressing plus views for re-mapping parallelism!
  - Large-degree vertices cause load-balancing issues.
  - But our architecture can stripe those across memories!
- Don't need to be SIMD / SIMT
  - Many *different* queries / algs simultaneously
  - Think streaming... And no cache coherency...
  - Can modify the graph / data while processing[1]

Again, think of the *data* and not the processing.

---

[1] Chunxing Yin and J.R. Concurrent Katz Centrality for Streaming Graphs. HPEC 2019. DOI 10.1109/HPEC.2019.8916572

Want To Play?

We’re hiring.
Software, hardware, and the whole stack.

There is a 32-node system in CRNCH at Georgia Tech:

`https://crnch-rg.cc.gatech.edu/near-memory-and-in-memory/`

Of if you want a system for yourself:

- Richard Sheroff `<rsheroff@lucata.com>`