Richard Jefferson

CSCI 415

5/31/2025

Assignment 5: Sorting in Chapel

## 1. Is Chapel Competitive with C’s Sequential Implementation?

Chapel can be competitive with C’s `qsort()` for large datasets and distributed execution across multiple locales. C’s `qsort()` benefits from decades of compiler optimization and incurs minimal runtime overhead, making it ideal for single-locale, small to mid-sized sorts. In contrast, Chapel is designed for scalability and excels when parallelism or distributed memory becomes necessary.  
  
For example, sorting an array of 65,536 elements on four locales:  
- `qsort()` completed in 0.0089 seconds.  
- `bitonic.chpl` took 42.50 seconds.  
- `hybrid.chpl` completed in 3.04 seconds.  
  
While Chapel incurs a greater baseline overhead, the `hybrid` approach dramatically outperforms bitonic when parallelized properly, demonstrating its value at scale.  
  
Conclusion: Chapel is not faster than `qsort()` on small workloads, but becomes increasingly competitive (and potentially superior) for large datasets in multi-locale environments.

## 2. Which Chapel Program Is Fastest and in What Circumstances?

|  |  |  |  |
| --- | --- | --- | --- |
| Program | Best Use Case | Strengths | Weaknesses |
| bitonic.chpl | Very large datasets across many locales | High parallelism, simple structure | High overhead for small arrays |
| hybrid.chpl | Medium-to-large arrays, 4–16 locales | Fast local sorts + global merge efficiency | Requires locale-awareness, more complex logic |

Bitonic Sort (`bitonic.chpl`): Scales well across locales for massive datasets. Its simple recursive structure makes it easy to distribute, but also imposes significant runtime cost for smaller workloads.  
  
Hybrid Sort (`hybrid.chpl`): Performs better in practical scenarios by using local quicksorts (fast, optimized) and combining them with a global bitonic merge. This results in significantly better runtimes across moderate numbers of locales.  
  
For example, with 131,072 elements and 4 locales:  
- `bitonic.chpl` took 94.05 seconds  
- `hybrid.chpl` only took 6.45 seconds  
  
Conclusion: `hybrid.chpl` is typically the fastest Chapel solution across a broad range of practical use cases. `bitonic.chpl` is best suited for extremely large, uniformly distributed workloads on many-core systems.

## 3. Evaluation of Chapel as a Programming Language

Chapel offers a high-level, productive approach to parallel and distributed programming. It abstracts away many low-level concurrency mechanisms using readable constructs like `forall`, `on`, and `dmapped`, which allow developers to write scalable programs without explicit thread or message management.

* Strengths:
* Intuitive syntax for parallelism and data distribution.
* Strong support for distributed memory via locales and domain maps.
* Reduces development time compared to C with MPI/OpenMP.
* Encourages clean, readable code in scientific computing contexts.
* Limitations:
* Smaller ecosystem and less tooling than mature languages like C/C++.
* Performance tuning requires understanding of runtime behavior and memory layout.
* Overhead in simple cases can make Chapel slower than optimized native implementations.

Conclusion: Chapel is well-suited for large-scale parallel computation in research or HPC contexts. It is less ideal for small, performance-critical utilities or environments with strict runtime constraints.

## 4. Timing Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | Locales | quicksort.c Time | bitonic.chpl Time | hybrid.chpl Time |
| 2¹⁴ = 16,384 | 1 | 0.0019 sec | 9.60 sec | 0.78 sec |
| 2¹⁶ = 65,536 | 4 | 0.0089 sec | 42.50 sec | 3.04 sec |
| 2¹⁷ = 131,072 | 4 | 0.0201 sec | 94.05 sec | 6.45 sec |
| 2¹⁸ = 262,144 | 4 | 0.0431 sec | 173.17 sec | 10.19 sec |
| 2²⁰ = 1,048,576 | 4 | 0.1813 sec | 819.53 sec | 43.72 sec |