Composable Parallel Libraries in Charm++

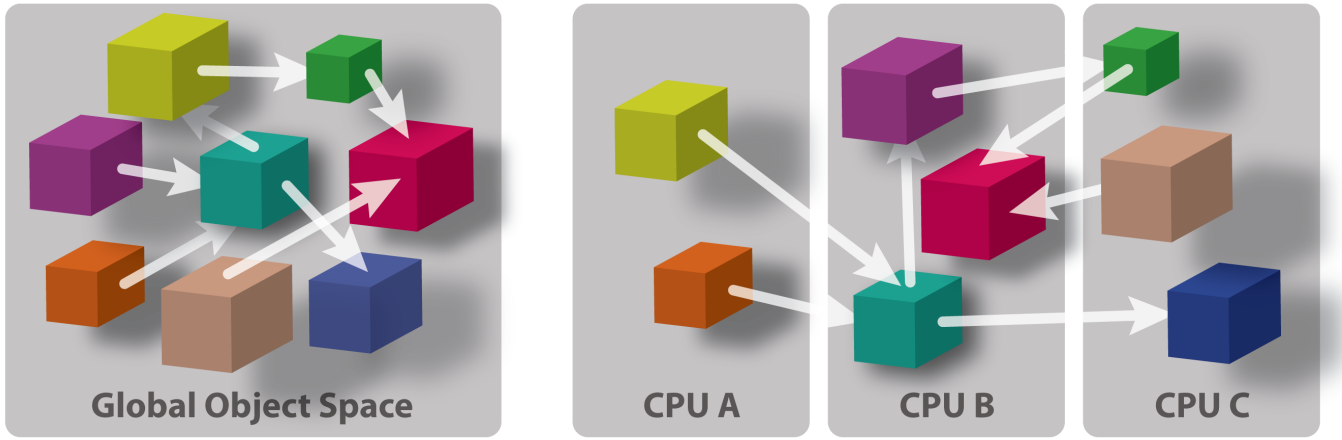
Phil Miller Laxmikant V. Kalé*

Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
*{mille121, kale}@illinois.edu

SIAM PP12: 15 February 2012

Charm++

Programming Model

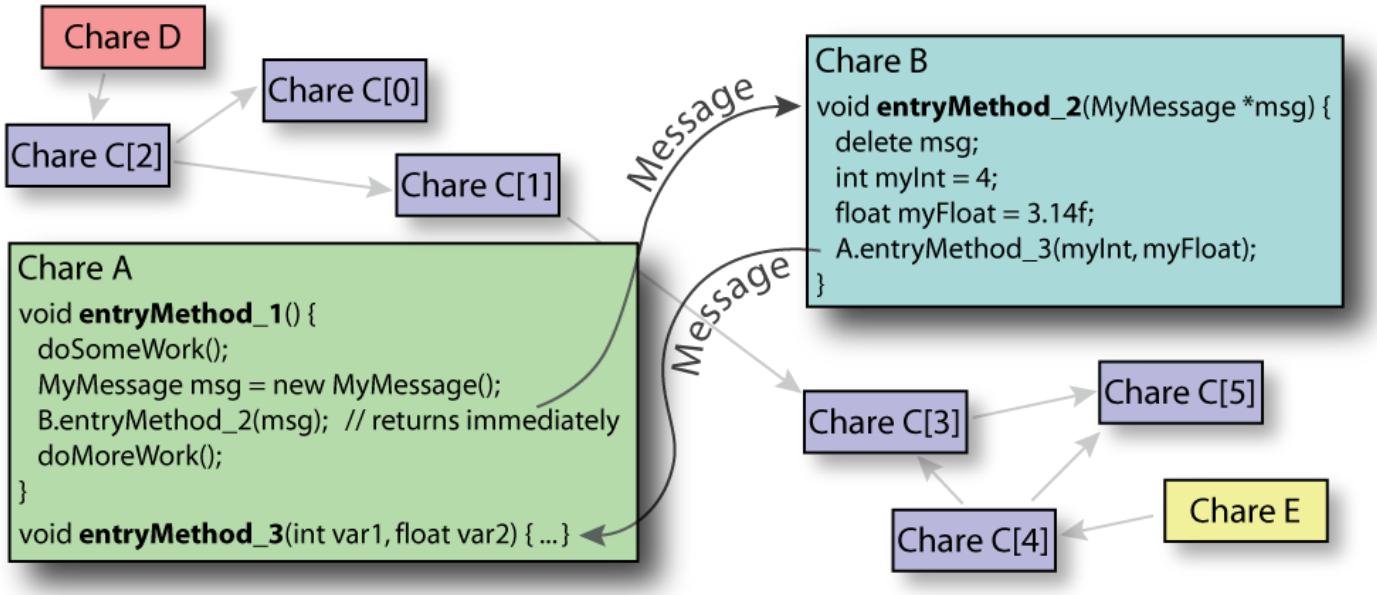


Object-based Express logic via indexed collections of interacting objects (both data *and* tasks)

Over-decomposed Expose more parallelism than available processors

Charm++

Programming Model



Message-Driven	Trigger computation by invoking remote *entry* methods
Non-blocking, Asynchronous	Implicitly overlapped data transfer
Runtime-Assisted	scheduling, observation-based adaptivity, load balancing, composition, etc.

Charm++

Capabilities

- Promotes natural expression of parallelism
- Supports modularity

Charm++

Capabilities

- Promotes natural expression of parallelism
- Supports modularity
- Overlaps communication and computation
- Automatically balances load

Charm++

Capabilities

- Promotes natural expression of parallelism
- Supports modularity
- Overlaps communication and computation
- Automatically balances load
- Automatically handles heterogenous systems
- Adapts to reduce energy consumption
- Tolerates component failures

For more info

http://charm.cs.illinois.edu/why/

Separation of Concerns

- Application developers focus on their algorithms and data
- Libraries should
  - not tie users' hands
  - share resources seamlessly
  - overlap
  - manage their own performance
- Strong runtime makes it possible!

LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)

LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)
- Block-centric
  - Algorithm from a block's perspective
  - Agnostic of processor-level considerations

LU: Capabilities

- Composable library
  - Modular program structure
  - Seamless execution structure (interleaved modules)
- Block-centric
  - Algorithm from a block's perspective
  - Agnostic of processor-level considerations
- Separation of concerns
  - Domain specialist codes algorithm
  - Systems specialist codes tuning, resource mgmt etc

	Lines of Code			Module-specific Commits	
	CI	C++	Total		
Factorization	517	419	**936**	472/572	83%
Mem. Aware Sched.	9	492	501	86/125	69%
Mapping	10	72	82	29/42	69%

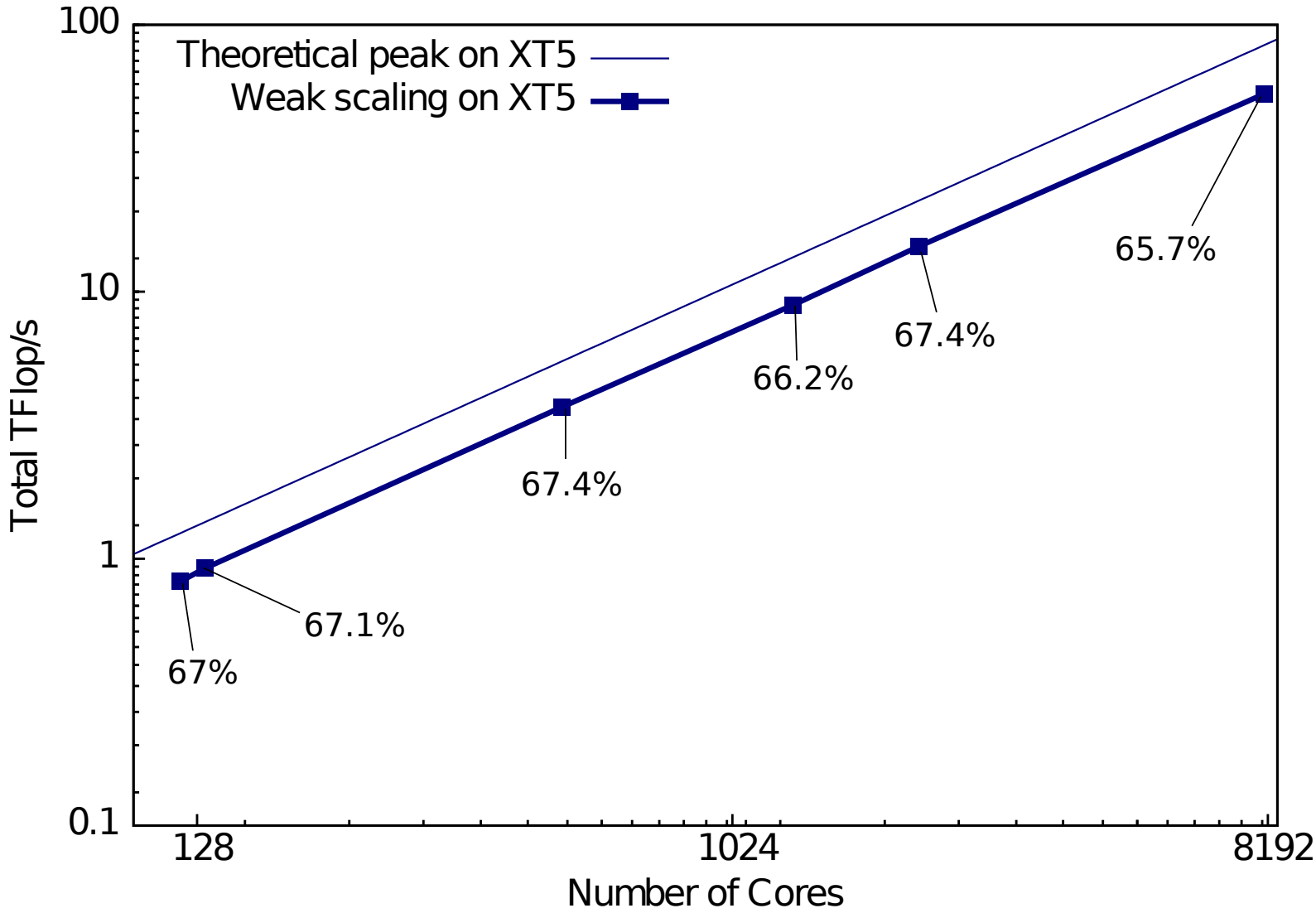
LU: Capabilities

- Flexible data placement
  - Don't mind client's layout - transposition is cheap
  - Variations don't impose on client
  - Can improve performance[1]
- Memory-constrained dynamic lookahead

[1] Lifflander et al., IPDPS 2012

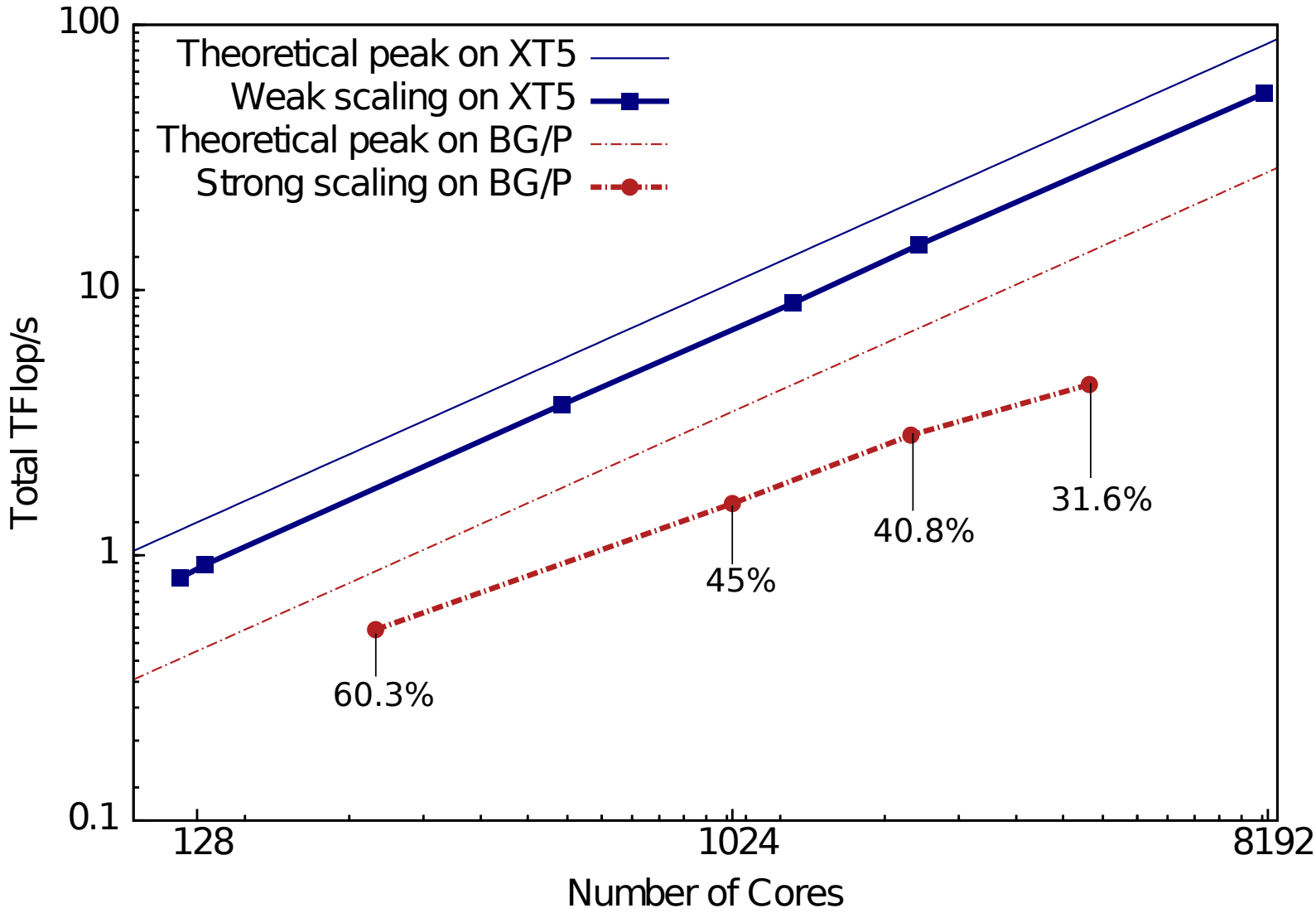
LU: Performance

Weak Scaling: (N such that matrix fills 75% memory)



LU: Performance

... and strong scaling too! (N=96,000)

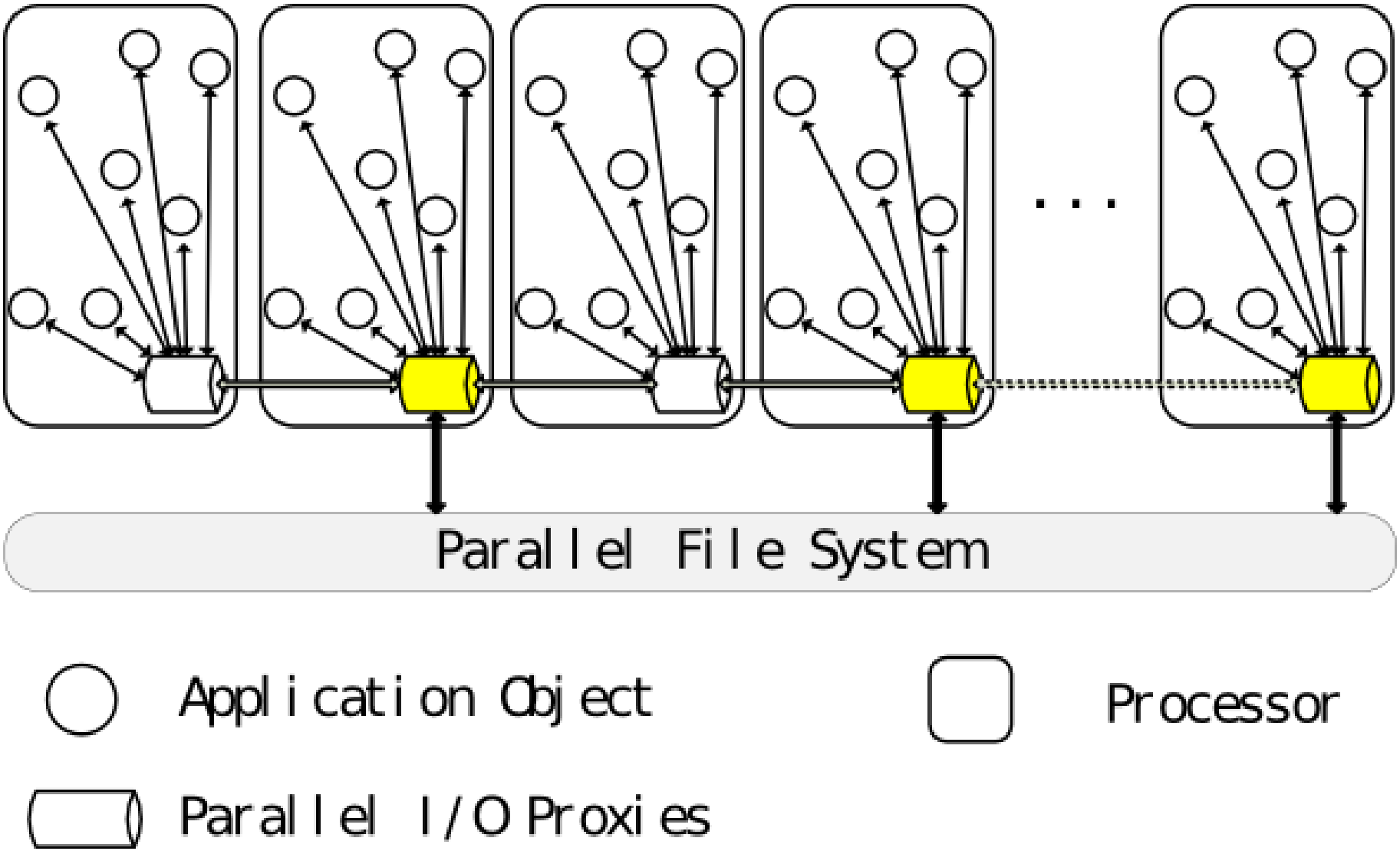


Parallel IO

MPI-IO is selfish, still demands dedicated nodes
Overlap IO in-line with the application!

Parallel IO

Architecture



Parallel IO

Implementation notes

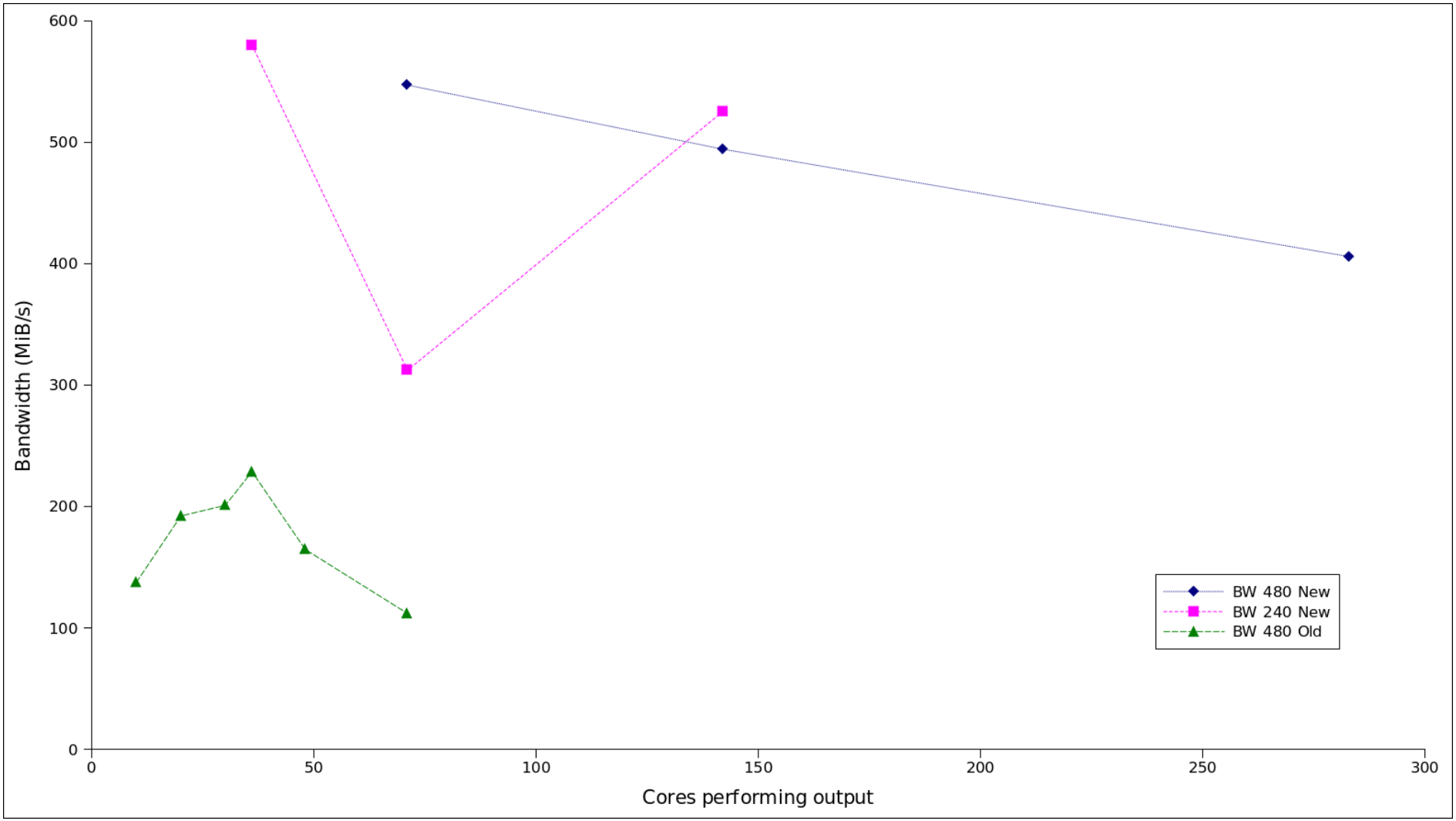
- Forward data to selected processors for stripe-disjoint access
- Buffer to write whole stripes (not in results shown)

Parallel IO

Implementation

```
void Manager::write(Token token, const char *data,
                    size_t bytes, size_t offset) {
  Options &opts = files[token].opts;
  do {
    size_t stripe = offset / opts.peStripe;
    int pe = opts.basePE + stripe * opts.skipPEs;
    size_t bytesToSend =
      min(bytes, opts.peStripe - offset % opts.peStripe);
    thisProxy[pe].write_forwardData(token, data,
                                    bytesToSend, offset);
    data += bytesToSend;
    offset += bytesToSend;
    bytes -= bytesToSend;
  } while (bytes > 0);
}
```

Parallel IO


600
500
400
300
200
100
0
Bandwidth (MiB/s)
0
50
100
150
200
250
300
Cores performing output
BW 480 New
BW 240 New
BW 480 Old

Conclusion

- Parallel libraries needn't be call and return
- Need to respect resource bounds
- Applications can find other work to do
- Let developers fully utilize system resources