A Communication Framework for Fault Tolerant Parallel Execution

Nagarajan Kanna   Jaspal Subhlok
Edgar Gabriel   Eshwar Rohit

University of Houston

David Anderson

UC Berkeley Space Science Lab

NSF

CS@UH

Volpex: Parallel Execution on Volatile Nodes

- **Key motivation:** Idle desktops represent a massive unused computation resource pool
- **BOINC & CONDOR**
  - BOINC: 500,000+ volunteer nodes worldwide, many application projects
  - CONDOR: job scheduler, widely used for desktops and clusters, 100s of installations
  - ***But, only Sequential and "bag of tasks" parallelism***
- **Volpex Goals:** Execution of communicating parallel programs ON volatile ordinary desktops
- **Key problem:** High failure rates AND coordinated execution

Example Application: REMD

- Collaboration with Prof. Margaret Cheung, UH Physics
- Studying the folding thermodynamics of small to modest size proteins in explicit solvent
- High computation requirements, modest communication. Use of "dataspace" for
  - Synchronization of processes
  - Store/Read energy values between neighbors
  - Exchange temperature values to drive next simulation step

REMD – Temperature swapping between replicas

STEP	P1	P2	P3	P4	P5	P6	P7	P8
1	270	280	290	300	310	320	330	340
2	280	270	300	290	320	310	330	340
3	290	270	300	280	320	310	330	340
4	290	270	300	280	310	320	340	330
5	280	270	310	290	300	330	340	320

- Application run with 8 scenarios (8 temperatures)
- Processes that swap temperatures at a step have same background color

Not all HPC applications push communication limits

Major Challenges in VOLPEX

Failure Management

- Replicated processes
- Independent process checkpoint/recovery, i.e., no coordination on checkpoints or restart
- Hybrid

Programming/Communication Model

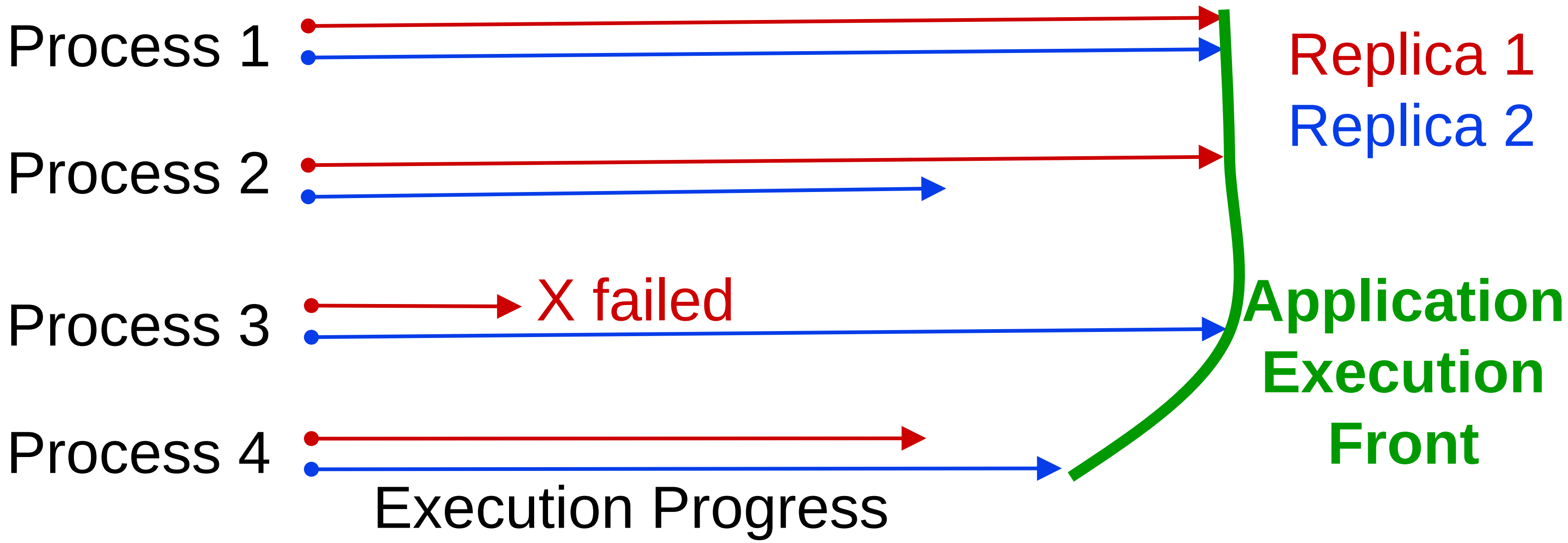
- **Volpex Dataspace API**
- **Volpex MPI**

Execution management

- Selection of “good” nodes for execution
- Integration with BOINC/Condor

Volpex Approach to Fault Tolerant Execution

Redundancy and/or independent checkpoint/restarts
→ *multiple physical processes per logical process*



- **Application progress tied to the fastest process replica**
- **Seamless progress despite failures**
- Minimum overhead of redundancy

Dataspace Programming Model

Independent processes communicate with one way PUT/GETs to abstract *dataspace (Linda, Javaspaces..)*

PUT (tag, data) place **data** in dataspace indexed with **tag**

READ (tag, data) return **data** matching the **tag**.

GET (tag, data) return and remove **data** matching **tag**

- Single variable length tag
  - No associative matching
- Blocking READ/GET
  - Synchronization tool. Non-blocking may come later
- PUTs can overwrite locations

Implementation with fault tolernace considered

Dataspace API with redundancy

LINDA implemented manyyy times!! What is new?

Fault tolerance approach (checkpoint_replication) implies redundant processes/execution

➔ a logical PUT/GET may be executed many times

- ➔ a late replica may PUT a value that is out of date
- ➔ a late replica may READ a value that has been overwritten

Consistent Execution with Redundant Process Replicas

Consider that a logical PUT / GET leads to multiple executable calls in temporal order

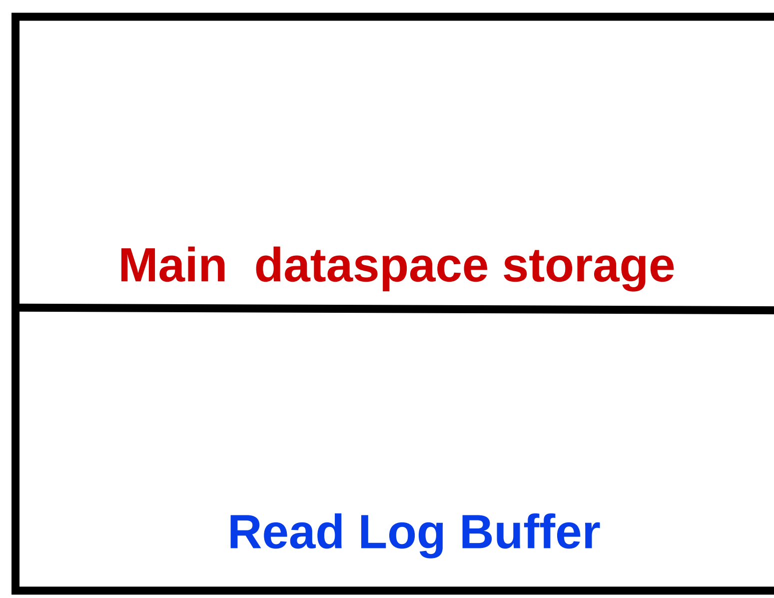
PUT1 , PUT2, PUT3… / GET1, GET2, GET3…

- New Consistency rules
  - PUT1 is executed normally. PUT2, PUT3,.. Ignored
  - GET1 gets the data object that matches at the time of its execution. GET2, GET3…. must also get a copy of the same data object.

Current Dataspace Implementation

API calls appended with <process id, request #> at client. Server can distinguish between first and replica calls.

- Replica PUTs identified and ignored
- First GET copies returned data object to a log. Replica GETs serviced from the log.
- (Assume determinism within a process – but not for application)



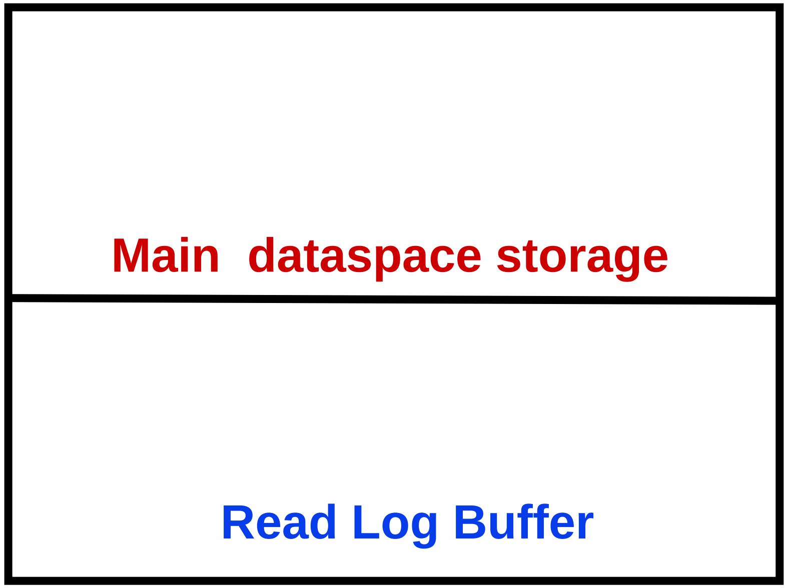
Future Dataspace Implementation

Optimistic Logging: Data object moved to log buffer only when overwritten

Log Buffer Management: Currently circular buffer in core. Can be on disk, smarter

Distributed

Multithreaded



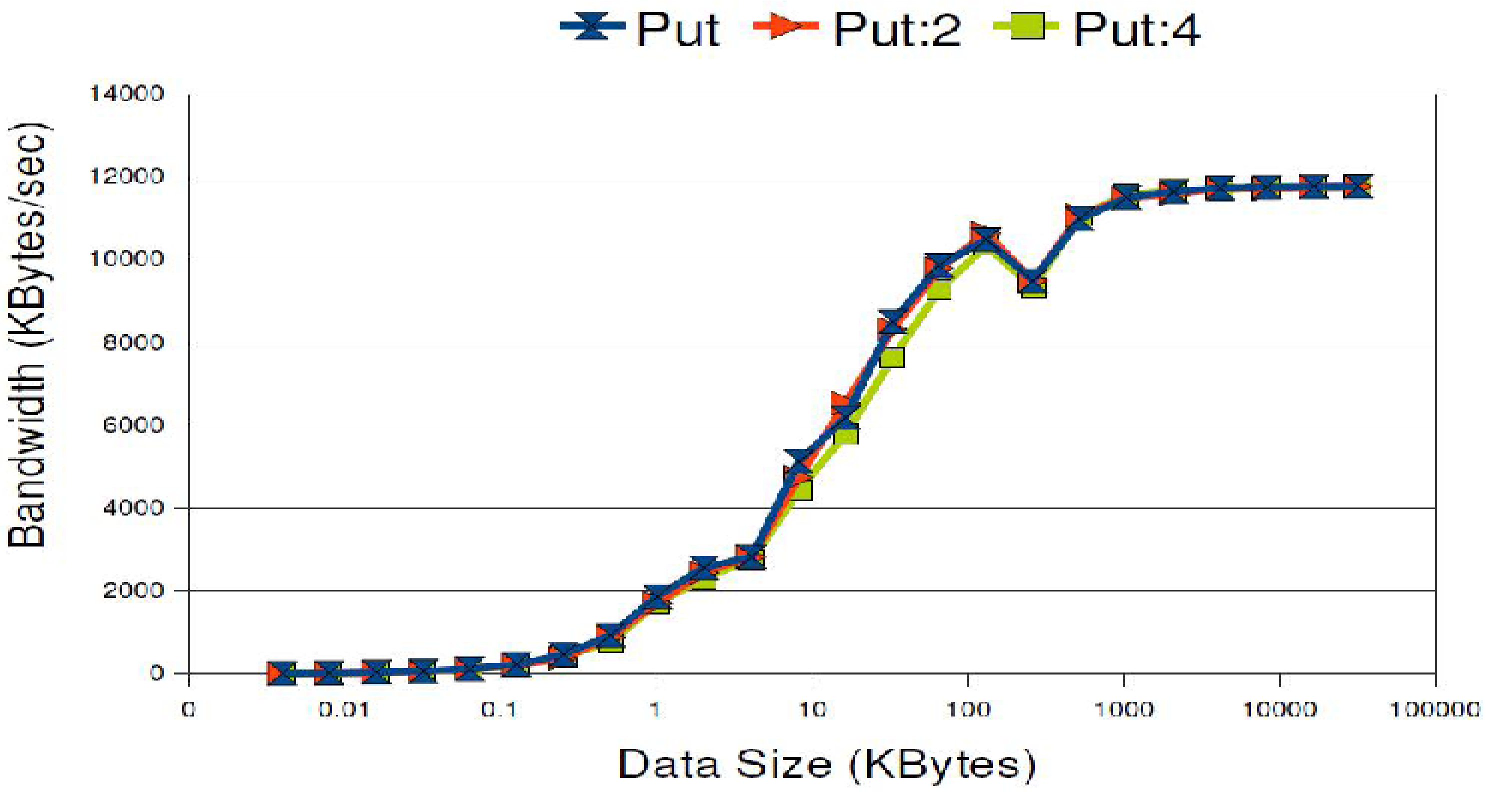
Implementation, Experiments, Results

- Applications/Examples
  - Replica Exchange Molecular Dynamics (REMD)
  - Implementation of Map-Reduce
  - Parallel Sorting by Regular Sampling (PSRS)
  - Sieve of Eratosthenes
  - Micro benchmarks

Failure tolerated with no impact on performance

- Testbed for Results
  - Clients: Atlantis Itanium2 1.3GHz dual core 4GB RAM
  - DSS: AMD Athlon 2.4GHz dual core, 2GB RAM

BANDWIDTH: 'PUT' WITH REPLICAS
(measured at Dataspace server)

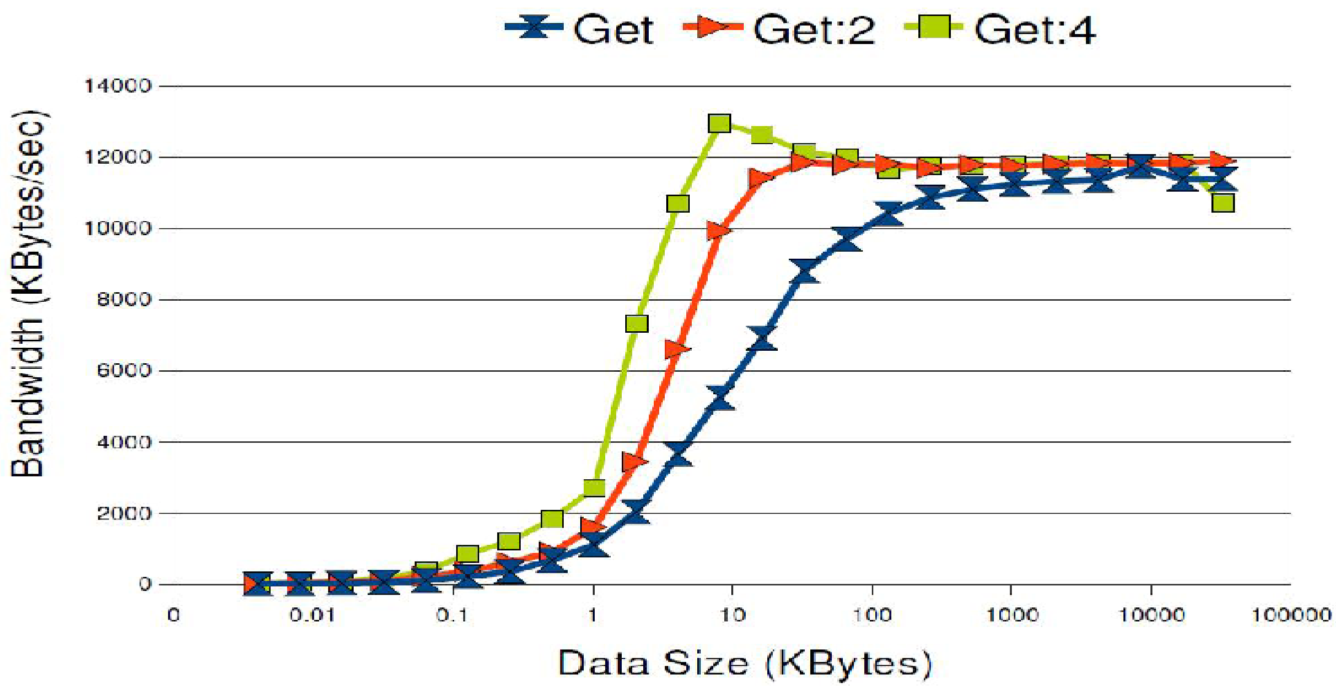


Little overhead of replica PUTs that are ignored.

CS@UH

BANDWIDTH: 'GET' WITH REPLICAS

(total bandwidth at server. identical for READ)



Replica Gets cause additional traffic.

The link is saturated early with replicas

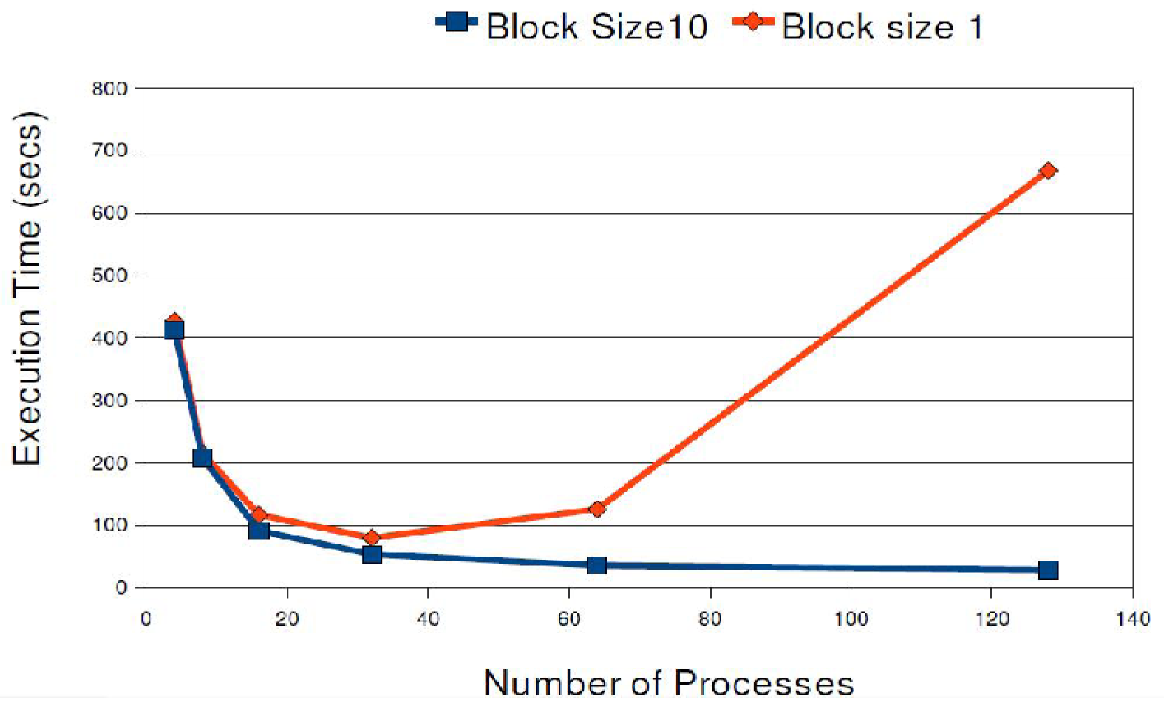
Example: Sieve of Erastothenes
(finding Primes)

In Parallel SoE: Numbers are distributed among processes. One process finds a prime and broadcasts to all. Others eliminate the multiples of the new prime.

- **Dataspace API:** one process PUTs a new prime, others READs it.

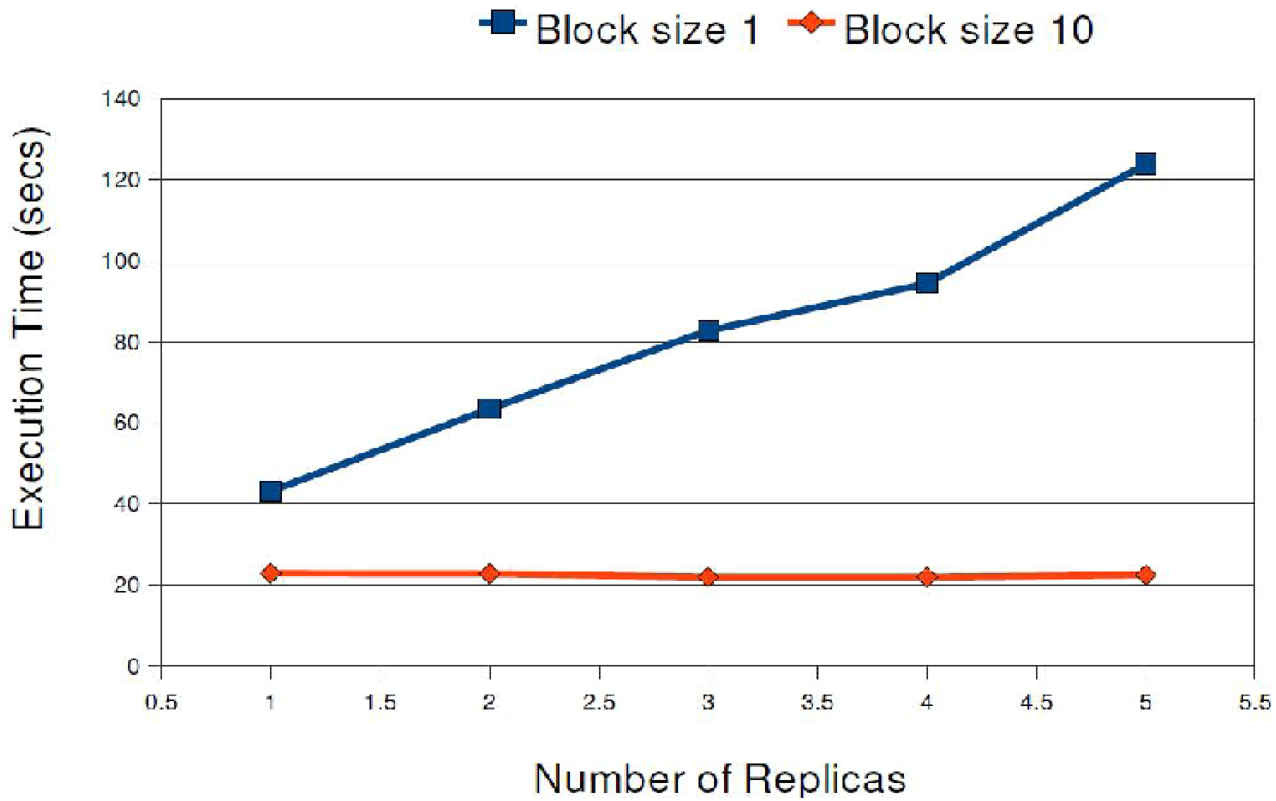
Blocking version: A group of prime numbers are discovered and broadcast as a group instead of indivudally.

Sieve of Erastothenes
(up to 2 billion numbers)



Blocked version scales well. Unblocked is communication intensive

Sieve of Erastothenes
(impact of replication)



Replication has no impact on blocked version.
Slows down the unblocked version significantly

Conclusions

Enabling a new class of algorithms and applications to run on idle ordinary desktops. Dataspace API offers a good communication solution.

Future work will

- Enhance the design and implementation of API
- Deploy on desktop virtual clusters with BOINC
- Apply to clusters – ideas are general

Code availanble on request **jaspal@uh.edu** **www.cs.uh.edu/~jaspal**

Thanks to NSF