cuNumeric II

CS315B

Lecture 4

Includes slides from Michael Bauer, Nvidia Research

CuNumeric

- To use CuNumeric, replace

  import numpy as np
  by
  import cunumeric as np

- Tranforms NumPy programs to run in parallel
  - On multiple GPUs
  - Across large clusters

- Today: How does this work?

A Simple NumPy Code

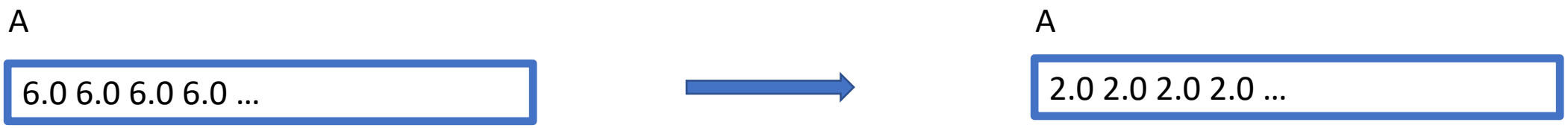
```
n = 1000
I = 100
v = np.zeros(n)
v[int(n/2)] = 1000
center = v[1:-1]
left = v[:-2]
right = v[2:]
for i in range(I):
        center[:] = (center + left + right) / 3.0
```

A Simple NumPy Code

```
n = 1000
I = 100
v = np.zeros(n)   -- initialize an array of all zeroes
v[int(n/2)] = 1000
center = v[1:-1]  -- define three views
left = v[:-2]
right = v[2:]
for i in range(I):
    center[:] = (center + left + right) / 3.0  -- two array adds and a divide
```

A Simple Case

A = A / 3.0

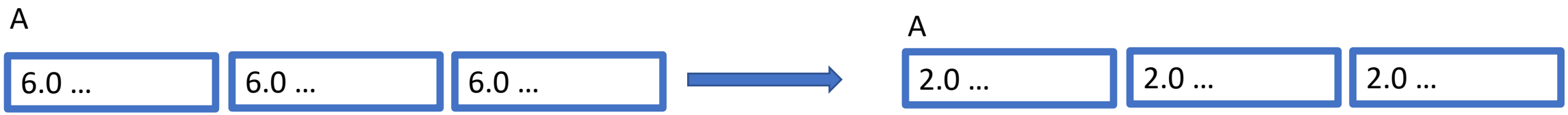


Data Parallelism

A = A / 3.0

Idea

- Partition A into multiple chunks
- Perform the computation on each chunk in parallel



Tasks

```
fun div(A,B,n) =
    for i in A.indices:
        B[i] = A[i] / n
```

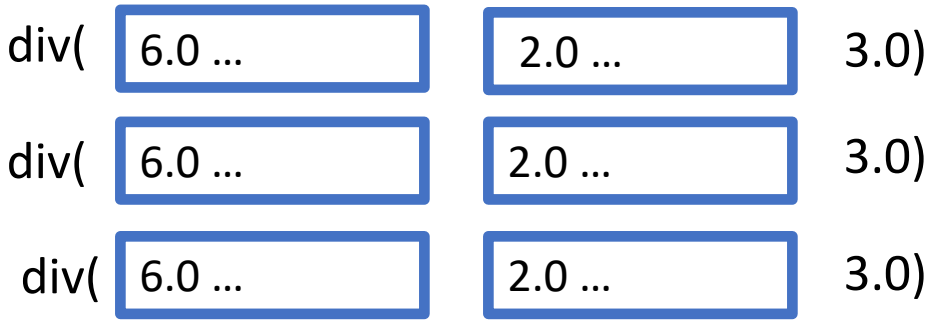
A *task* is a function on arrays

- No side effects on anything but the function arguments
- Some array arguments may be read, written, or both

Task Calls

cuNumeric

- Partitions the arrays into subarrays
- Translates array operations into task calls
- That can be offloaded to GPUs or CPUs



Partitioning

- NumPy arrays are partitioned into subarrays
  - Array views are also partitioned
- Partitioning does not allocate space
  - Names the elements of subarrays
  - But does not create copies
- Partitioning is done by a collection of heuristics:
  - Do partition arrays into equal parts
  - Don’t partition arrays that are too small
  - Try to partition views in ways compatible with original array partition

Partitioning

```
n = 1000
I = 100
v = np.zeros(n)
v[int(n/2)] = 1000
center = v[1:-1]
left = v[:-2]
right = v[2:]
for i in range(I):
  center[:]  = (center + left + right) / 3.0
```

Assuming 4 pieces, views could be partitioned:

center	left	right
v[1:249]	v[0..248]	v[250]
v[250:499]	v[259..498]	v[251..500]
v[500:749]	v[499..748]	v[501..750]
v[750:998]	v[749..997]	v[751..999]

Call these partitions

C[0..3] L[0..3] R[0..3]

E.g., C[0] = center[1:249]

Exploiting Partitions

```
for i in range(I):
  center[:] = (center + left + right) / 3.0
```

is transformed to

```
for i in range(I):
 for j in range(4):
  C[j] = (C[j] + L[j] + R[j]) / 3.0
```

Tasks on entire arrays (or views) are replaced by sets of tasks on the subarrays of the partitions.

The number of tasks depends on the number of components of the partition. Here we have one task per element of the partition, captured as a loop.

Exploiting Partitions, Cont.

```
for i in range(I):
  for j in range(4):
    T1[j] = C[j] + L[j]
    T2[j] = T1[j] + R[j]
    C[j] = T2[j] / 3.0
```

Compound statements are broken up into sequences of statements with one array operation each.

Intermediate results are stored in temporary arrays.

Rewriting Using Tasks

```
for i in range(I):
  for j in range(4):
    T1[j] = C[j] + L[j]
    T2[j] = T1[j] + R[j]
    C[j] = T2[j] / 3.0
```

```
for i in range(I):
  for j in range(4):
    add(T1[j], C[j], L[j])
    add(T2[j], T1[j], R[j])
    div(C[j], T2[j], 3.0)
```

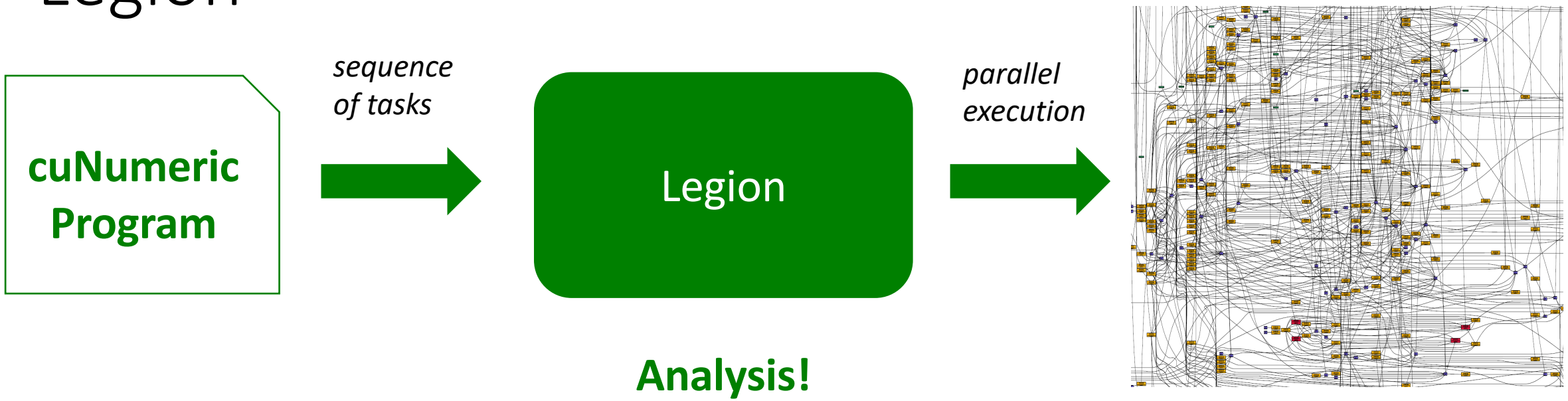
Issuing Tasks

```
for i in range(I):
  for j in range(4):
    add(T1[j], C[j], L[j])
    add(T2[j], T1[j], R[j])
    div(C[j], T2[j], 3.0)
```

The program issues the sequence of task calls:

```
add(T1[0], C[0], L[0])
add(T2[0], T1[0], R[0])
div(C[0], T2[0], 3.0)
add(T1[1], C[1], L[1])
add(T2[1], T1[1], R[1])
div(C[1], T2[1], 3.0)
...
```

Legion



Analysis!

Legion is a task-based distributed runtime

- sequential semantics
- parallel execution

cuNumeric is a Legion library

Legion

- A distributed, task-based runtime system
- Analyzes a stream of tasks sent from a client application
- Infers *dependences* between tasks
  - Task B depends on task A if their array arguments overlap and one of the tasks writes one of those overlapping arrays
  - The execution order of dependent tasks must be preserved to guarantee sequential semantics
  - Tasks that are independent can run in parallel

Recall

```
for i in range(I):
  for j in range(4):
    add(T1[j], C[j], L[j])
    add(T2[j], T1[j], R[j])
    div(C[j], T2[j], 3.0)
```

The program issues the sequence of task calls:

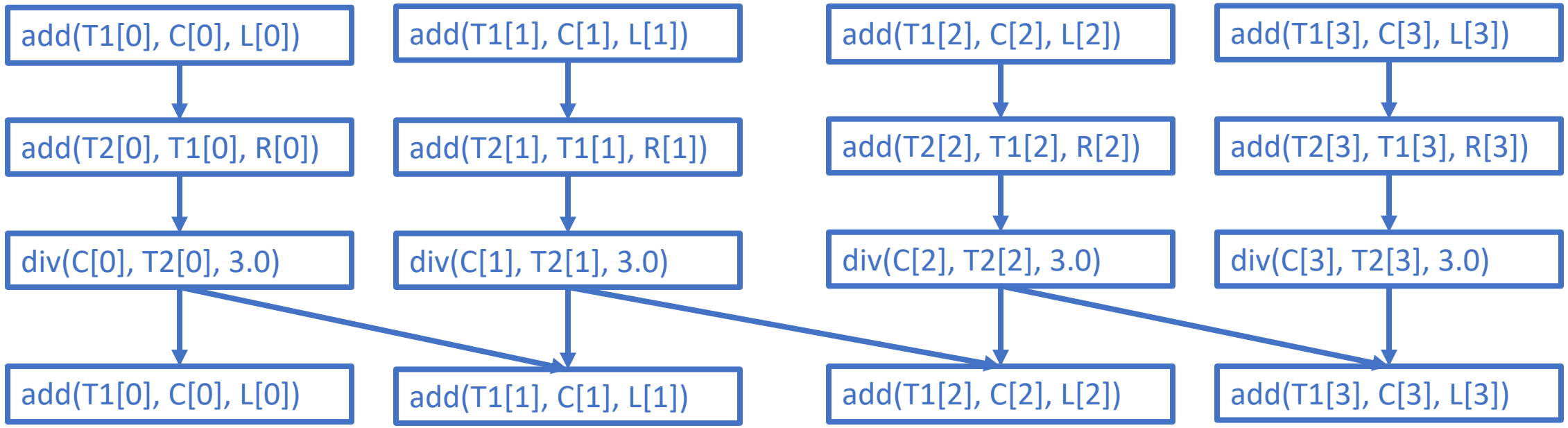
```
add(T1[0], C[0], L[0])
add(T2[0], T1[0], R[0])
div(C[0], T2[0], 3.0)
add(T1[1], C[1], L[1])
add(T2[1], T1[1], R[1])
div(C[1], T2[1], 3.0)
...
```

Dependence Graph

The program issues the sequence of task calls:

add(T1[0], C[0], L[0])
add(T2[0], T1[0], R[0])
div(T3[0], T2[0], 3.0)
add(T1[1], C[1], L[1])
add(T2[1], T1[1], R[1])
div(C[1], T2[1], 3.0)
…

Dependence Graph

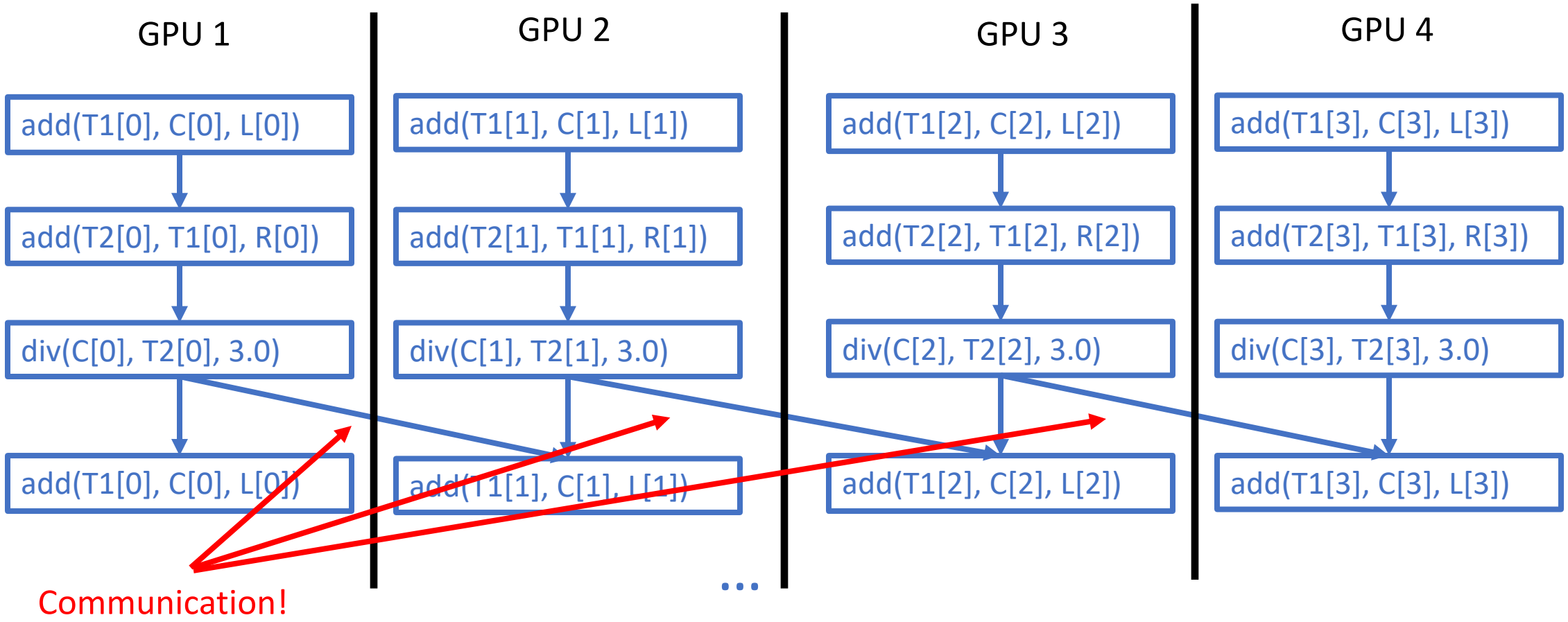


...

Execution

- This dependence graph is distributed across the machine
- The number of subarrays of partitions is chosen to match the number of processors available
  - More processors = more subarrays

Dependence Graph

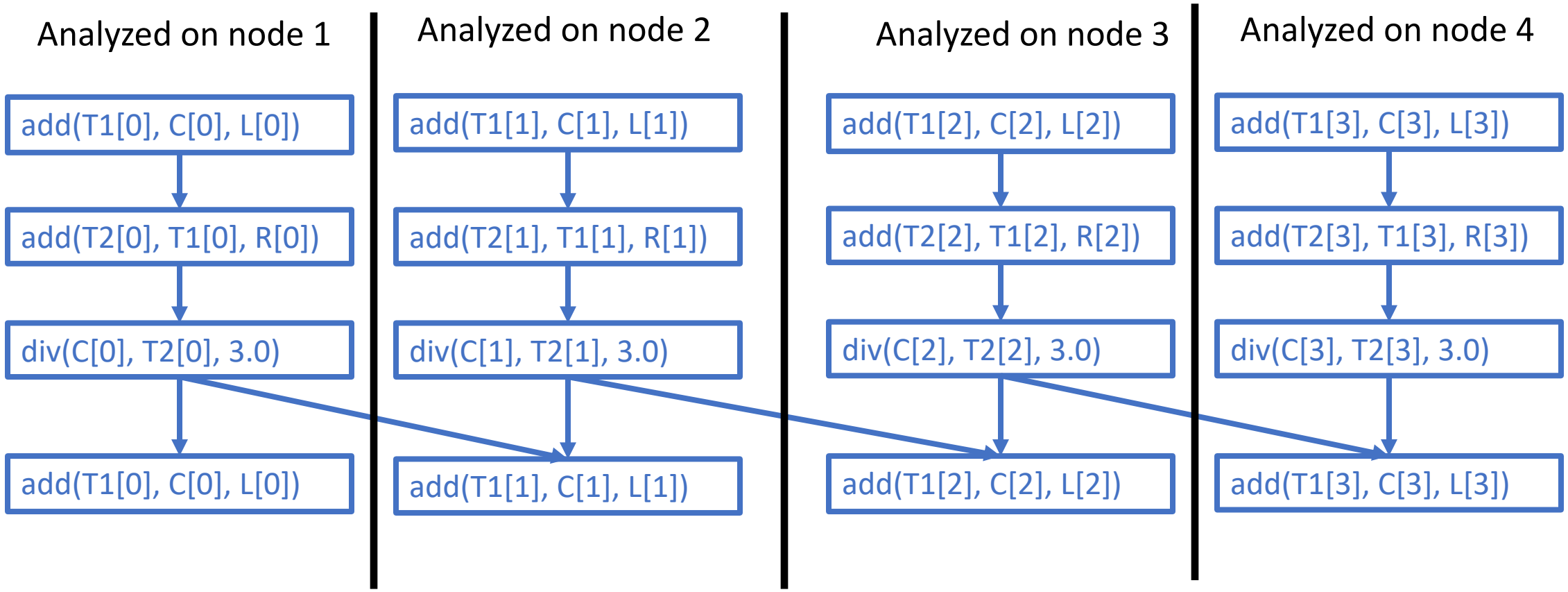


Distributed Analysis

- Where is the analysis of tasks done?
  - Find dependences
  - Decide where to run tasks
  - Figure out communication
  - Issue copy commands to move data
  - All of this is not cheap!

- Distribute the analysis work across the machines, too
  - Split the work the same way that work is assigned to the different processors
  - But note the analysis is always done on CPUs

Distributed Analysis



Does this remind you of anything?

Summary

- cuNumeric
  - Partitions arrays
  - Creates tasks for operations on subarrays
  - Issues those tasks in program order to Legion

- Legion
  - Analyzes the incoming tasks (in SPMD fashion)
  - Finds dependences
  - Sends tasks to compute resources
  - Guarantees the semantics of the original NumPy program is preserved

Other Task-Based Systems

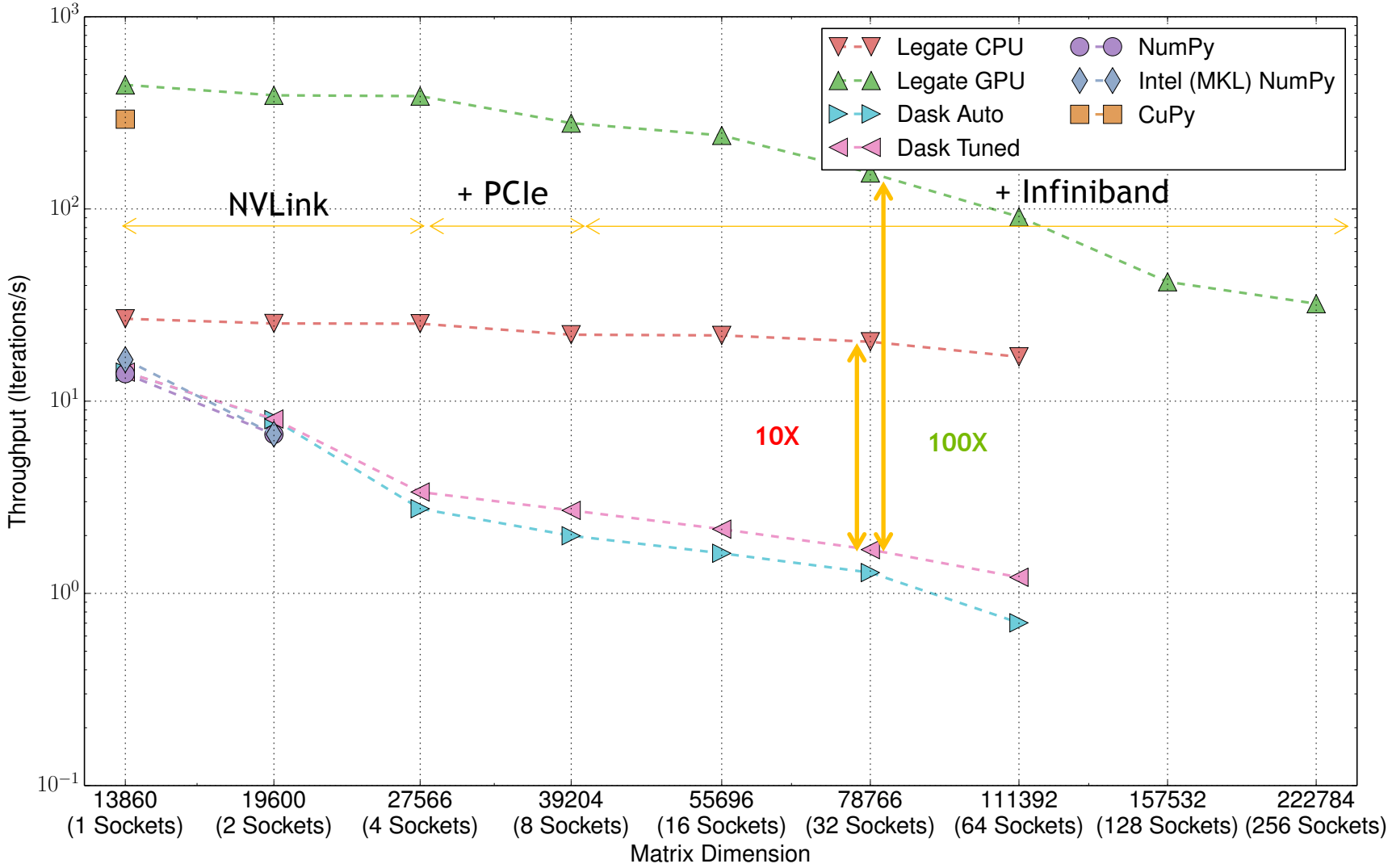
- Task-based systems are quite popular in data analytics and ML

- Examples
  - TensorFlow
  - PyTorch
  - Spark
  - Dask
  - Pathways

- We'll talk about some of these systems later in the course

Jacobi Solver

```
import numpy as np

A = np.random.rand(N,N)
b = np.random.rand(N)

x = np.zeros(A.shape[1])
d = np.diag(A)
R = A - np.diag(d)
for i in xrange(b.shape[0]):
  x = (b - np.dot(R,x)) / d
```



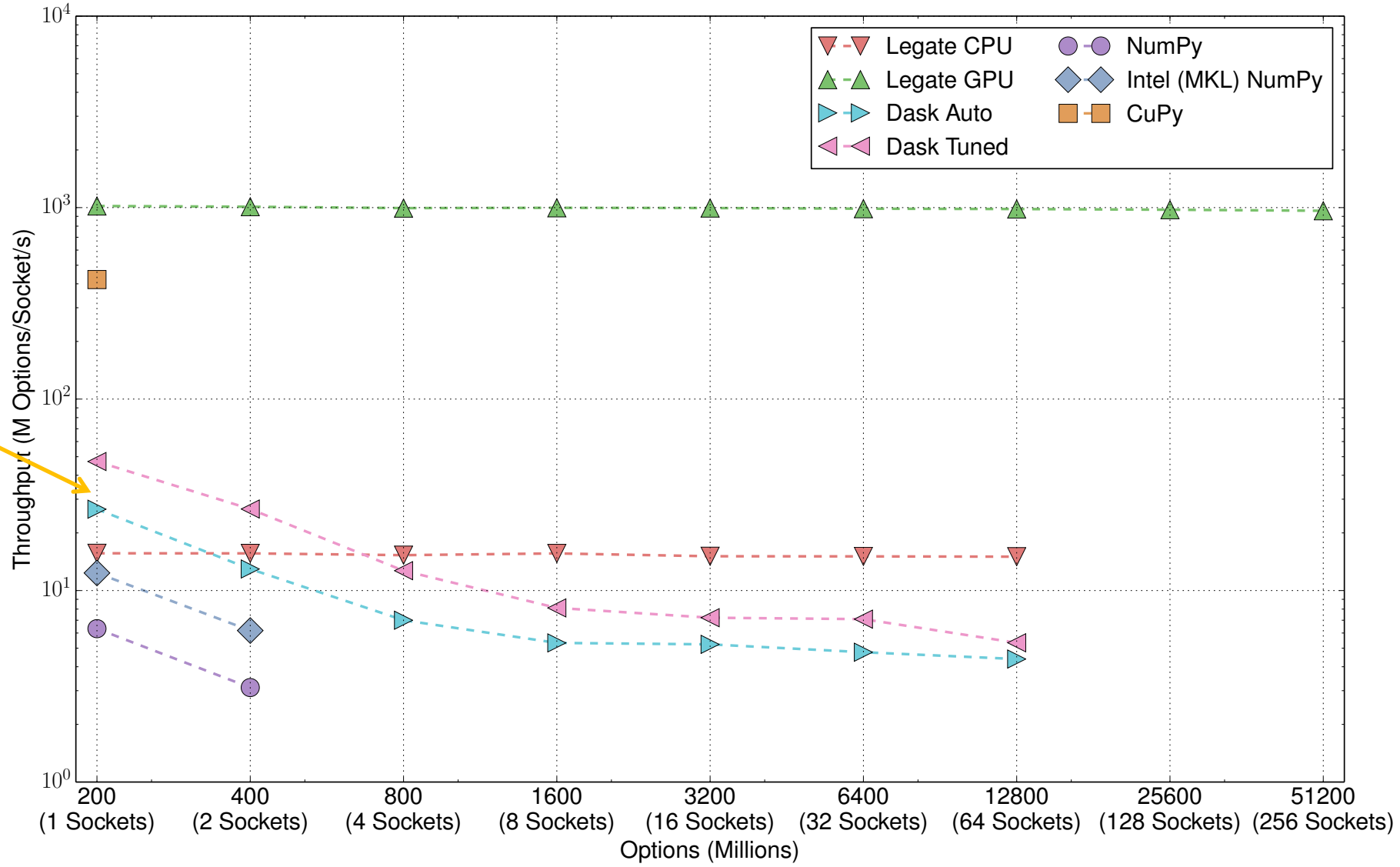
Black Scholes

No (application) communication

Expect perfect weak scaling

Dask starts out faster…
Why? Operator Fusion

… but has to trade off parallelism for task granularity to scale

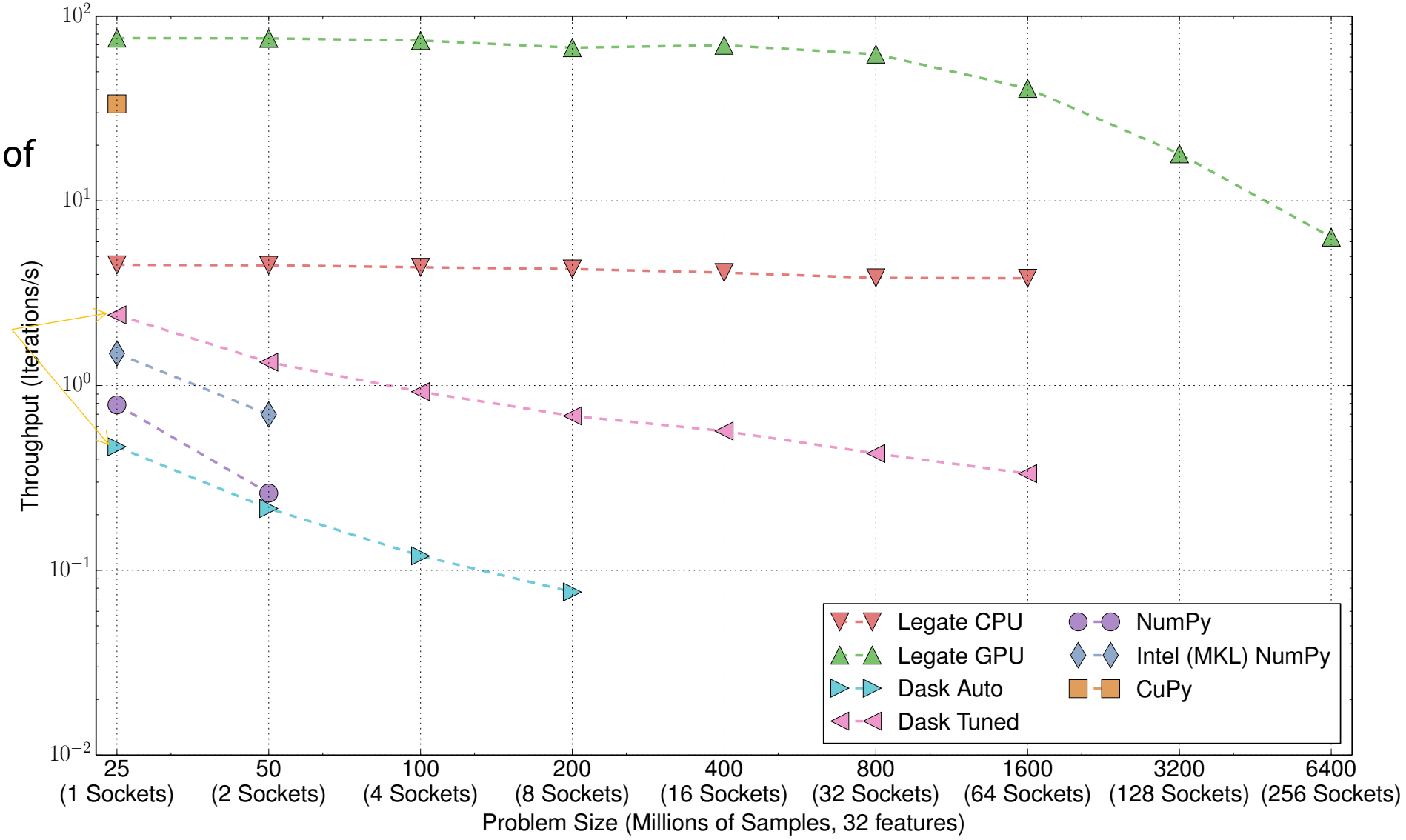


Logistic Regression

Unusually shaped GEMVs

Difficult parallel reduction of weights with broadcast

Large difference in Dask: picking chunks matter

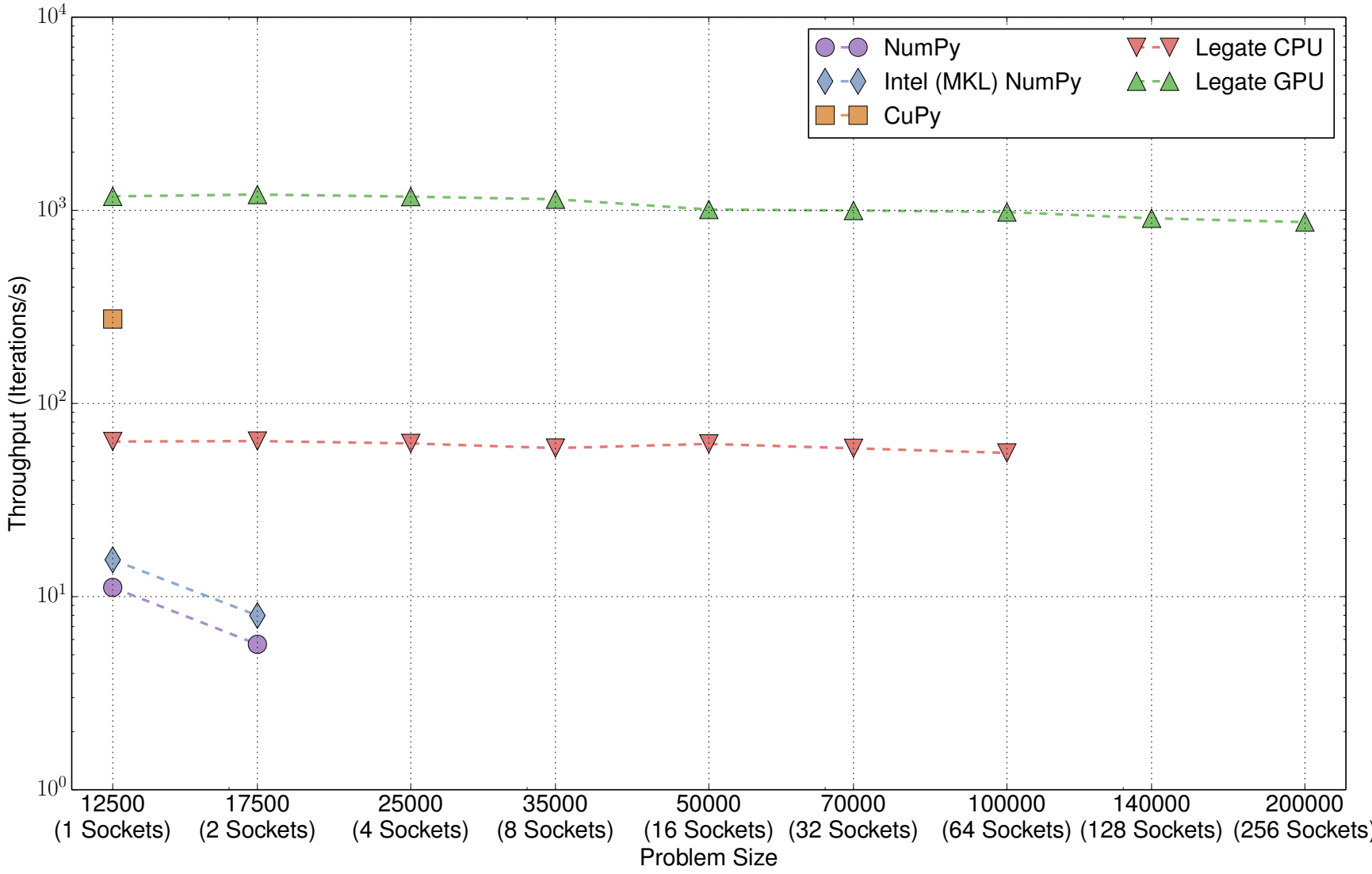


Legate infers nearest neighbor communication

Dask does not support some indexing features

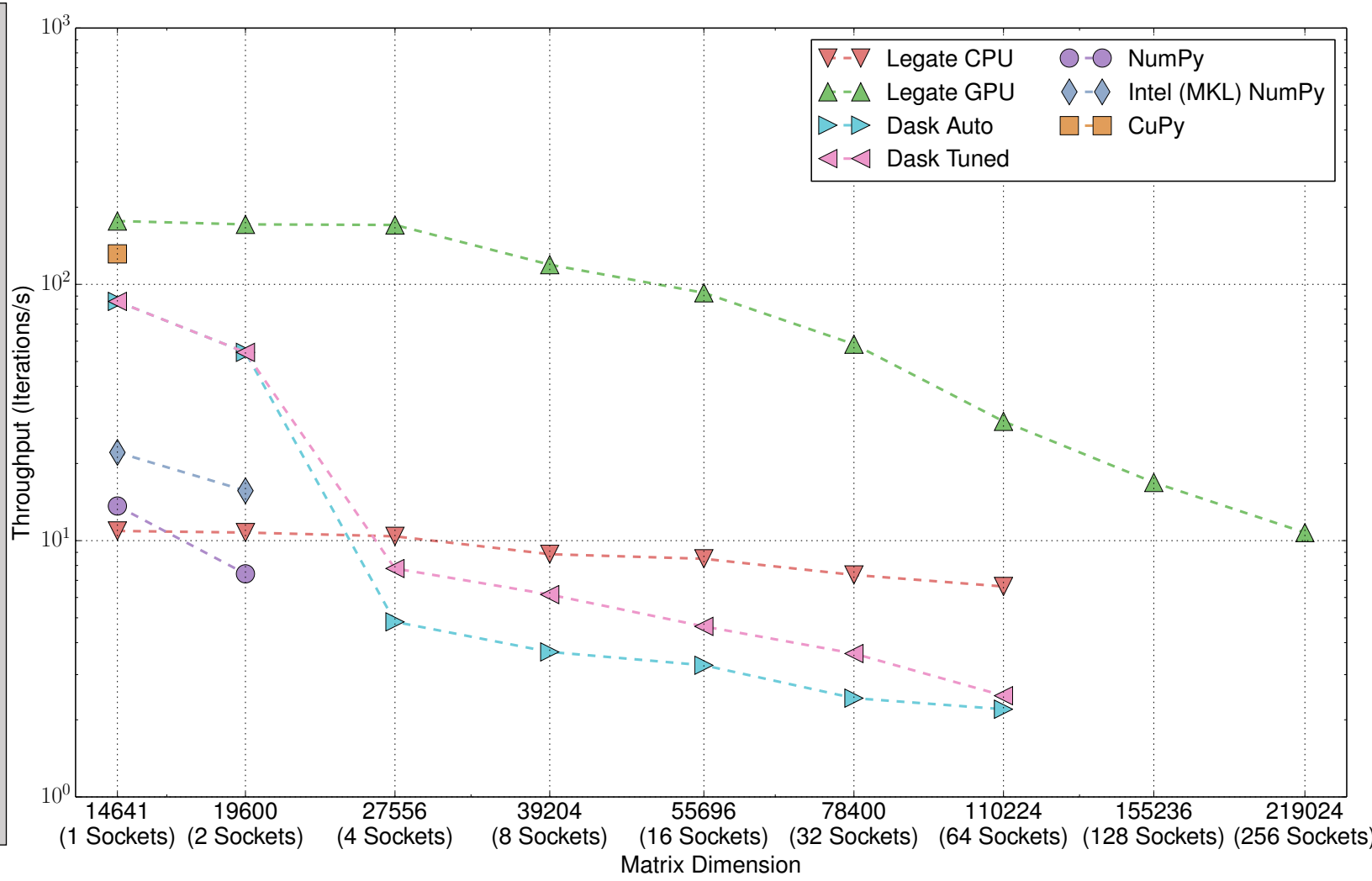
```
import numpy as np

grid = np.zeros((N+2,N+2))
grid[:,0]  = -273.15
grid[:,-1] = -273.15
grid[-1,:] = -273.15
grid[0,:]  = 40.0

center = grid[1:-1, 1:-1]
north  = grid[0:-2, 1:-1]
east   = grid[1:-1, 2:  ]
west   = grid[1:-1, 0:-2]
south  = grid[2:  , 1:-1]
for i in range(I):
  average = (center + north +
             east + west + south)
  work = 0.2*average
  delta = np.sum(
    np.absolute(work-center))
  center[:] = work
```

Heat Diffusion Stencil



Preconditioned CG Solver

```
def preconditioned_solve(A, M, b):
    x = np.zeros(A.shape[1])
    r = b - A.dot(x)
    z = M.dot(r)
    p = z
    rzold = r.dot(z)
    for i in xrange(b.shape[0]):
        Ap = A.dot(p)
        alpha = rzold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rznew = r.dot(r)
        if np.sqrt(rznew) < 1e-10:
            break
        z = M.dot(r)
        rznew = r.dot(z)
        beta = rznew / rzold
        p = z + beta * p
        rzold = rznew
    return x
```



Conclusions

- cuNumeric automatically
  - Partitions data and computation
  - Distributes work across a cluster
  - Guarantees correct execution

- Performance can be excellent
  - Not guaranteed, because of heuristics used in partitioning
  - But in general quite good
  - And the user doesn’t need to understand parallel programming to get some benefit