# Locality auto-optimization and productive distributed-memory programming

Chris Keefe

**Laziness:** The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful and document what you wrote so you don't have to answer so many questions about it.

- Larry Wall, author of Perl

# A Machine-Learning-Based Framework for Productive Locality Exploitation

Kayraklioglu, Favry, El-Ghazawi

- Data locality is important
- It is costly to optimize manually especially on distributed systems
- Everyone's hardware is different
- Can we let the computer handle it?

# Wait – locality like cache misses?

- Locality describes “distance” data must travel to the processor
- Here, we’re talking locality like intra-node communication
- This is focused on truly HP compute - many nodes

# Wisdom from Stack Overflow?

“If you really want it to be fast then you may want to consider getting down to the bare metal to make sure you make best use of specific CPU features like SIMD instructions, branch prediction and cache coherence, at the expense of portability.”

- @Jason Williams

# Wisdom from Stack Overflow?

“If speed is your concern, there are highly optimized algorithms available that include optimizations for specific instruction sets (e.g. SIMD), implementing those all by yourself offers no real benefit”

- @Jim Brissom

# Use a Library – or a language?

# Chapel Background

- Chapel is a *parallel programming language*
- Chapel is focused on programmer productivity
- Chapel supports the PGAS memory model

# First-Class parallelism

- `locale` - the unit of locality, generally one ***node***
- `domain` - an index set
- `distribution` - the "shape" into which data should be split across nodes
- `coforall` - task-parallel loop
- `forall` - data-parallel loop

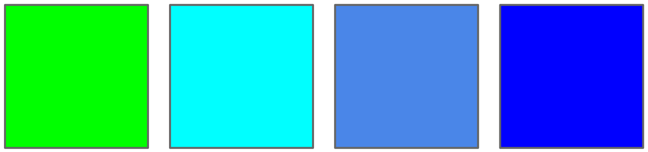
# PGAS - Partitioned Global Address Space

- A global memory model with the ability to differentiate hardware [2]
- Processes/Threads/Tasks have affinity with particular memory devices
- Possible to exploit locality, unlike “flat” GAS models

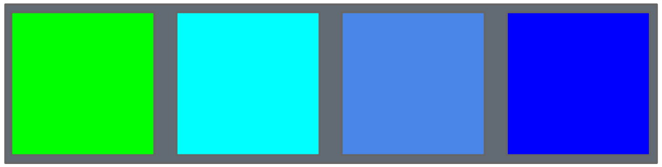
# PGAS - Partitioned Global Address Space

- Direct access to local and remote variables
- “Distributed arrays”
- Only outperforms MPI “if the programmer spends enough effort on exploiting locality”

# Distributed Arrays – the mental model

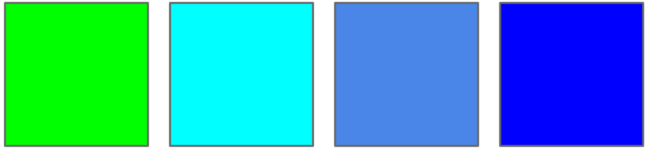


MPI: I have ¼ of my array on each node.



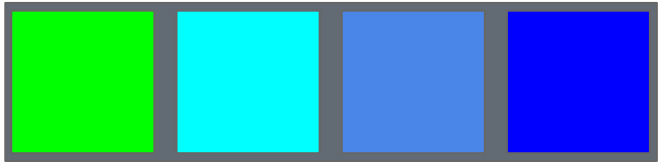
Chapel: I have an array on four nodes.

# Distributed Arrays – the mental model



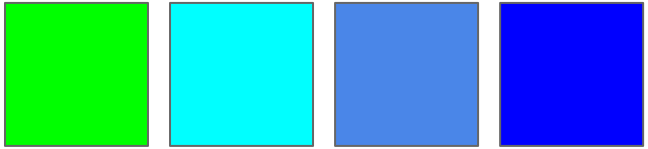
MPI: I have ¼ of my array on each node.

```
// each rank creates a ¼-size array
int *localArrA[N / nprocs];

// each rank populates it
for (int i = 0; i < N / nprocs; i++){
  // put data in
}

// Each rank can then calculate on local
array values, but nonlocal values
require explicit communication
```



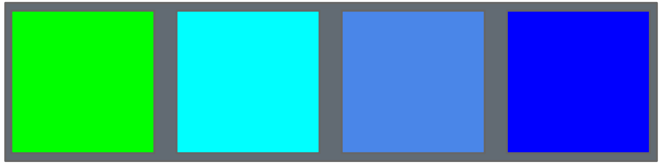
Chapel: I have an array on four nodes.

# Distributed Arrays – the mental model



MPI: I have ¼ of my array on each node.

```
// each rank creates a ¼-size array
int *localArrA[N / nprocs];

// each rank populates it
for (int i = 0; i < N / nprocs; i++){
  // put data in
}

// Each rank can then calculate on local
array values, but nonlocal values
require explicit communication
```



Chapel: I have an array on four nodes.

```
// our data should be distributed in blocks
use BlockDist;
// our indices fit a 2d array
var space = {1..N, 1..N};
// we map our “index space” into blocks
var domain = space dmapped Block(space);
// and create one distributed array
var  A: [dom] int;

// we can iterate over our array
forall (i, j) in A.domain do
  // ... some stuff!
```

# Task parallelism - `coforall`

- "creates a separate task for each iteration of the loop"
- Can be applied explicitly over `Locales` or user-defined `domains`

```
coforall i in 1..n {
  writeln("4: output from spawned task 1 (iteration ", i, ")");
  writeln("4: output from spawned task 2 (iteration ", i, ")");
}
```

Kinda like OpenMP:

```
#pragma omp parallel for
for(int i = 1; i < 100; ++i)
{
    ...
}
```

# Data Parallelism – `forall`

Forall loops parallelize loops in a data-driven fashion.

```
forall i in 1..n {
  A[i] = i;
}
```

# Data Parallelism – `forall`

Forall loops parallelize loops in a data-driven fashion.

```
forall i in 1..n {
  A[i] = i;
}
```

If A is a distributed array, each loop iteration is executed on the locale where the corresponding array element is.

# PGAS - Partitioned Global Address Space

- A global memory model with the ability to differentiate hardware [2]
- Processes/Threads/Tasks have affinity with particular memory devices
- Possible to exploit locality, unlike “flat” GAS models

# Naive implementations aren't fast...

**Listing 2.** Basic Matrix Transpose in Chapel

```
use BlockDist;
var space = {1..N, 1..N};
var dom = space dmapped Block(space);
var A: [dom] int, B: [dom] int;
for niter in 1..numIters do
 forall (i,j) in B.domain do
  B[i,j] = A[j,i];
 forall a in A do
  a += 1;
```

# Naive implementations aren’t fast...

**Listing 2.** Basic Matrix Transpose in Chapel

```
use BlockDist;
var space = {1..N, 1..N};
var dom = space dmapped Block(space);
var A: [dom] int, B: [dom] int;
for niter in 1..numIters do
 forall (i,j) in B.domain do
  B[i,j] = A[j,i];
 forall a in A do
  a += 1;
```

# … and manual implementations aren’t cheap/easy.

**Listing 3.** Transpose With Manual Communication

```
coforall l in Locales do on l {
 var bLocSubDom = B.localSubdomain();
 var tDom = {bLocSubDom.dim(2),
             bLocSubDom.dim(1)};
 var localA: [tDom] real;
  for i in 1..numIter {
   localA = A[transposeDom];
   forall (i, j) in bLocSubDom do
    B[i,j] = localA[j,i];
  forall (i, j) in A.localSubdomain() do
   A[i,j] += 1.0;
 }
}
```

# LAPPS - Locality-Aware Productive Prefetching Support

**Listing 4.** Transpose With LAPPS

```
A.transposePrefetch();
for niter in 1..numIters do
 forall (i,j) in B.domain do
  B[i,j] = A[j,i];
 forall a in A do
  a += 1;
```

# But what if our prefetching needs aren’t common?

**Listing 6.** Transpose With Custom Prefetch Method Supported by LAPPS

```
var accessTbl: [{0..#numLocales}] domain(2);
coforall l in Locales do on l {
  var myDomain = B.localSubdomain();
  var tDom = {myDomain.dim(2),
              myDomain.dim(1)};
   accessTbl[here.id] = tDom;
}
A.customPrefetch(accessTbl);
for niter in 1..numIters do
  forall (i,j) in B.domain do
   B[i,j] = A[j,i];
  forall a in A do
   a += 1;
```

This paper presents machine-learning-driven automation of locality optimization

# High-level architecture

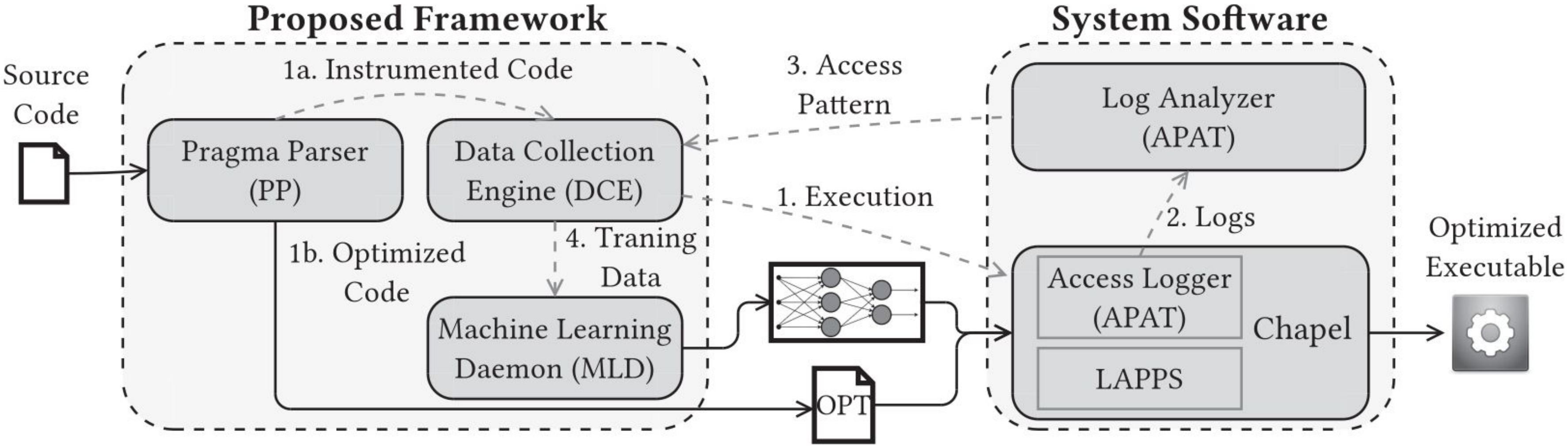



Fig. 1. High level overview of the framework and its interactions with the existing software stack.

# High-level architecture

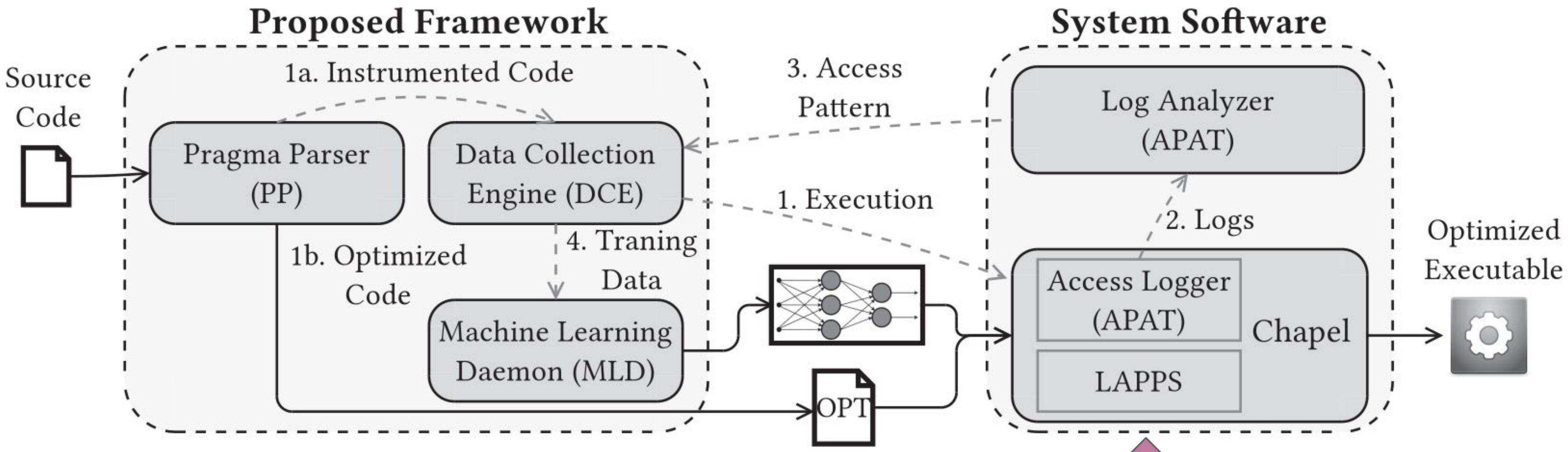



Fig. 1. High level overview of the framework and its interactions with the existing software stack.

# So now we can `pragma optimize arrays(A)`

**Listing 7.** Transpose With Pragma

```
use BlockDist;

// command line options
config const numIters = 10;
config const arrSize = 8192;

// create arrays
const arrIndices = {1..arrSize, 1..arrSize};
var A = newBlockArr(arrIndices, real);
var B = newBlockArr(arrIndices, real);

pragma optimize arrays(A)
for niter in 1..numIters {
 forall (i,j) in B.domain do
  B[i,j] = A[j,i];
 forall a in A do
  a += 1;
}

writeln(B);
```

Chapel runs your naive code, measures access patterns, and uses an Elastic Net regressor to define an optimal data access pattern to insert with LAPPS

# High-level architecture

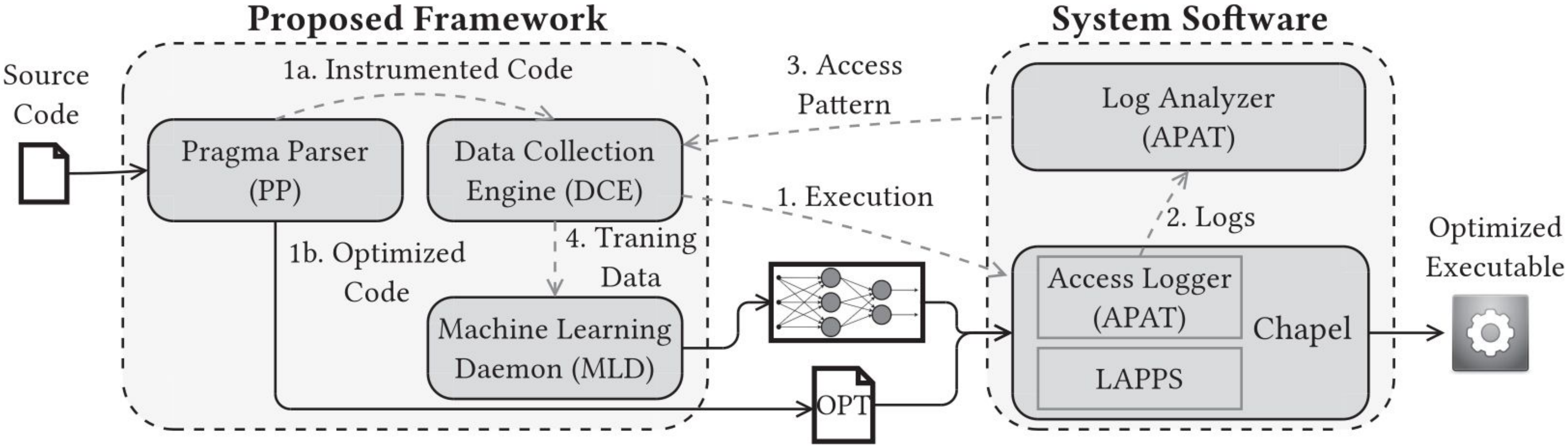



Fig. 1. High level overview of the framework and its interactions with the existing software stack.

# High-level architecture

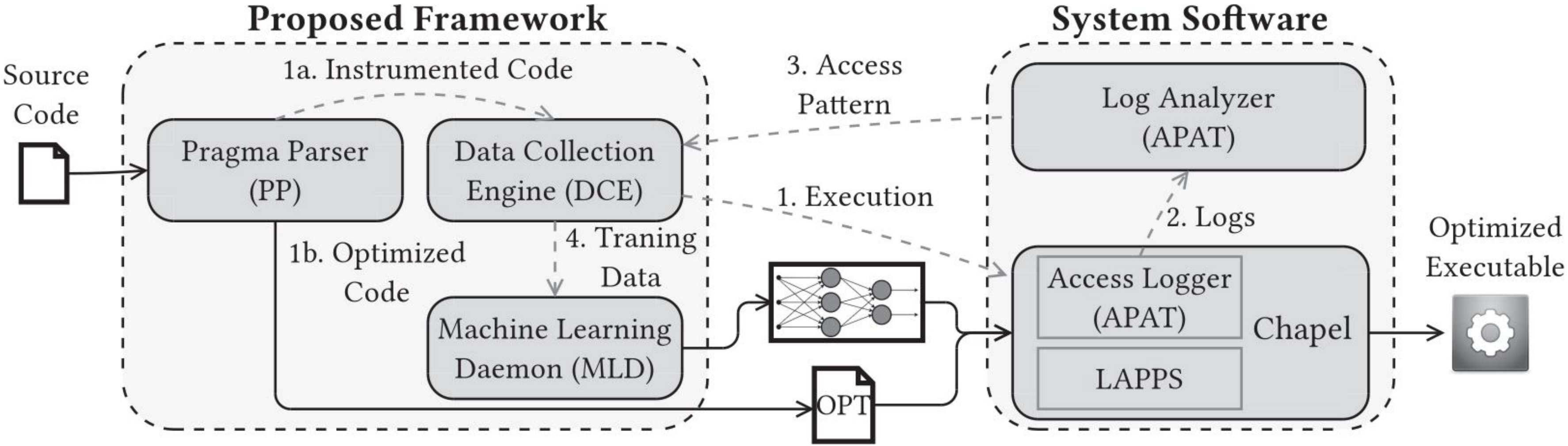
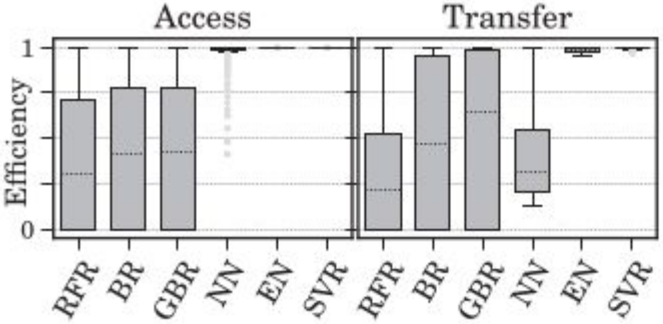



Fig. 1. High level overview of the framework and its interactions with the existing software stack.

# What makes a good ML model here?

- Efficiency of data access and communication
- Resistance to overfitting
- Effective with small data sets (workflow concerns)
- “Practicality” - compile time costs mostly

# Access/Transfer Efficiency




c: 300 seconds (n=569)

# Elastic Net wins.

TABLE 3
Summary Of Observations About Regressors

| Predictor | Accuracy | | | Performance | | Logistics |
|---|---|---|---|---|---|---|
| | Data Size | Subject to Overfitting | Universality | Training | Prediction | Saved Model Size |
| Random Forest Regressor (RFR) | Large | No | No | Fast | Slow | Large |
| Bagging Regressor (BR) | Large | No | No | Fast | Medium | Medium |
| Gradient Boosting Regressor (GBR) | Large | No | No | Medium | Fast | Medium |
| Neural Network (NN) | Medium | Mild | Yes | Slow | Fast | Small |
| Support Vector Regression (SVR) | Small | Yes | No | Slow | Fast | Small |
| *Elastic Net (EN)* | *Small* | *No* | *Yes* | *Fast* | *Fast* | *Small* |

# But what about the compile time overhead?

- 3 minutes on a cluster, 30 minutes on a workstation
- Compile once, then run repeatedly
- Results are near-identical on both platforms
- How important is optimization to your application?

# Measuring Success

- “inference accuracy”
- Training time
- Run time

# Many access patterns benchmarked with EN

- STREAM
- PRK-Stencil

- PRK-Transpose
- PRK-DGEMM

- LULESH
- PARACR

# Normal scale performance

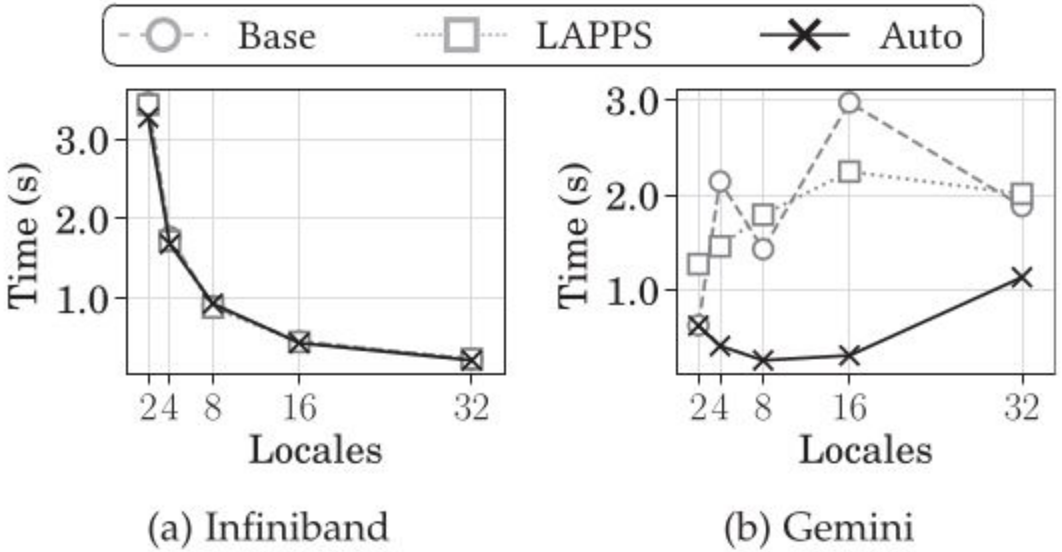
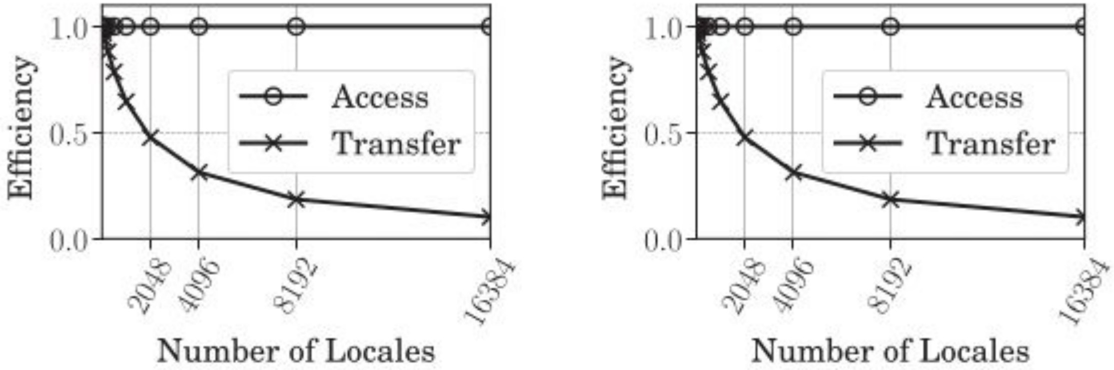



Fig. 18. PRK-stencil strong scaling results.

# Extreme scale efficiency




(a) Strong Scaling (b) Weak Scaling

Fig. 23. STREAM.

# No free lunch

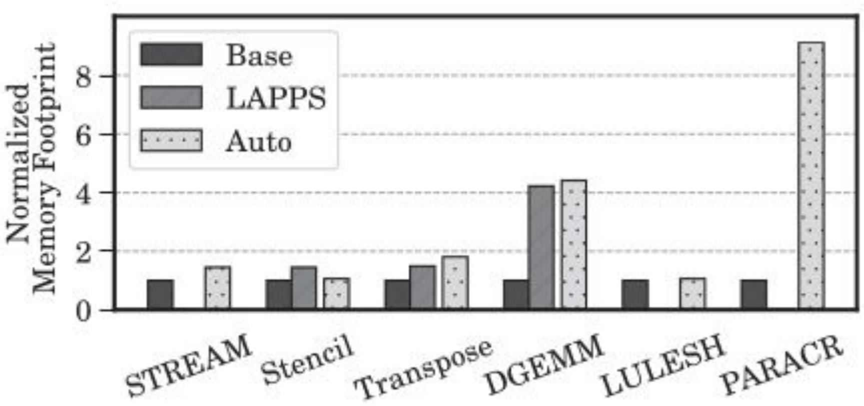



Fig. 27. Memory footprint of different versions of benchmarks.

# Final thoughts

- Meets language programmer-productivity goals
- “Solves” the requirement of manual communication implementation for most cases
- Optimization is architecture-agnostic
- Compile time and memory costs are reasonable

# References

[1] E. Kayraklioglu, E. Favry, and T. El-Ghazawi, “A Machine-Learning-Based Framework for Productive Locality Exploitation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1409–1424, Jun. 2021, doi: 10.1109/TPDS.2021.3051348.

[2] “Partitioned global address space,” Wikipedia. Jan. 17, 2021, Accessed: Apr. 07, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Partitioned_global_address_space&oldid=1000961794.

[3] “Scalability,” *Wikipedia*. Mar. 19, 2021, Accessed: Apr. 07, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Scalability&oldid=1012936862.

# 99%

Of great programmers are lazy.
Laziness is, arguably, the whole point of programming.