EVALUATING MODERN PROGRAMMING MODELS USING THE PARALLEL RESEARCH KERNELS

Jeff Hammond

Intel HPC

(wearing my academic hat today)

Acknowledgements

- **Tim Mattson, Intel**
- **Rob van der Wijngaart, Intel (now at NVIDIA)**
- Apo Kayi, Gabi Jost, Tom St John, Srinivas Sridharan (ex-Intel)
- John Abercrombie, Jacob Nelson (U. Washington)
- Kiran Pamnany, Intel (now at Caltech CliMA)
- Yijian (Tim) Hu, Georgia Tech (undergrad)
- Lisandro Dalcin, KAUST
- Brad Chamberlain, Cray (now HPE)
- Alex Duran, Alexey Kukanov, Pablo Reble, Xinmin Tian, Martyn Corden, Steve Lionel (Intel)
- Christian Trott (Sandia), Tom Scogland (RAJA)
- Alessandro Fanfarillo (then NCAR)
- Thomas Hayward-Schneider
- Many others, not listed here.

P|R|K

PARALLEL RESEARCH KERNELS

https://github.com/ParRes/Kernels/ has all the goods

Programming model evaluation

Standard methods

- NAS Parallel Benchmarks
- Mini Applications
  (e.g. Mantevo, LULESH)
- HPC Challenge

There are numerous examples of these on record, covering a wide range of programming models, but is source available and curated?

What is measured?

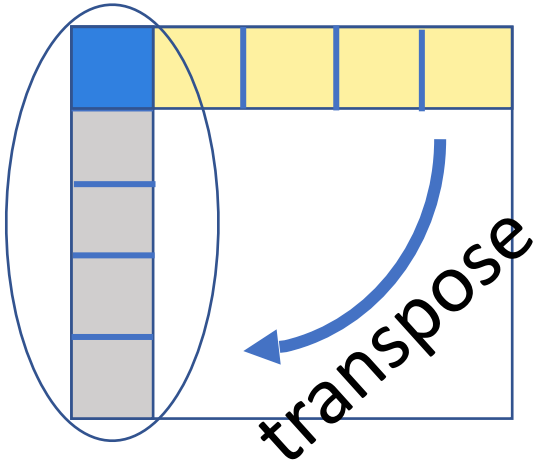
- Productivity (?), elegance (?)
- Implementation quality
  (runtime or application)
- Asynchrony/overlap
- Semantics:
  - Automatic load-balancing (AMR)
  - Atomics (GUPS)
  - Two-sided vs. one-sided, collectives

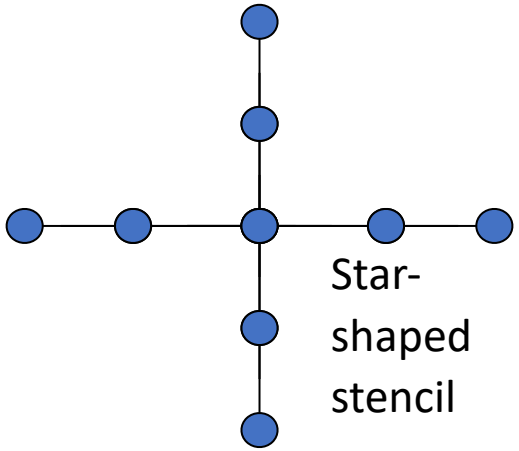
Goals of the Parallel Research Kernels

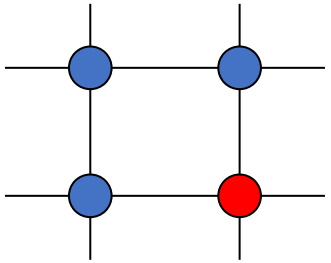
- **Universality**: Cover broad range of performance critical application patterns.
- **Simplicity**: Concise pencil-and-paper definition and transparent reference implementation. No domain knowledge required.
- **Portability**: Should be implementable in any sufficiently general programming model.
- **Extensibility**: Parameterized to run at any scale. Other knobs to adjust problem or algorithm included.
- **Verifiability**: Automated correctness checking and built-in performance metric evaluation.
- ~~Hardware benchmark~~: No! Use HPCChallenge, Xyz500, etc. for this.

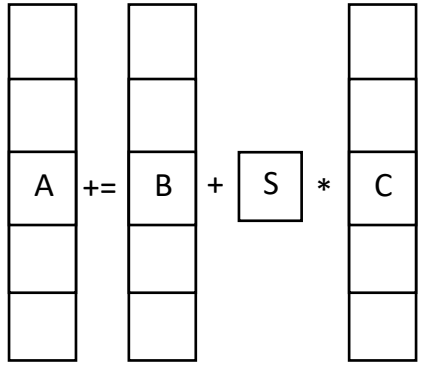
Outline of PRK Suite

- **Dense matrix transpose**
- Synchronization: global
- **Synchronization: point to point**
- **Scaled vector addition**
- Atomic reference counting
- Vector reduction
- Sparse matrix-vector multiplication
- Random access update
- **Stencil computation**
- Dense matrix-matrix multiplication
- Branch
- Particle-in-cell









$$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$$

Commentary

- Nstream (Not STREAM) accumulates, which means it does not use x86 nontemporal/streaming stores, which simplifies analysis.
- Stencil is 2D, which is the tortuous limit of domain-decomposed 3D but also covers image convolutions.
- Transpose requires a 1D decomposition to generate an all-to-all pattern of small messages or strided access.
- Synch_p2p is quite different with shared and distributed memory.
  - Distributed -> baton-passing synchronization.
  - Shared -> loop-carried dependency -> many barriers or fine-grain dataflow.

Project History

- Created by Tim Mattson a long time ago to understand CPU architecture.
- C89-based MPI-1 and OpenMP-3 ports by Tim and Rob.
- Used for Intel exascale software study in 2014-2016.
  - UPC is C99, Charm++ and Grappa are C++ - we should move beyond C89.
  - Chapel vs C89/MPI is not a useful comparison (hence “pretty” versions).
- Extended to C++ and Fortran with offload models because exascale was always going to be heterogeneous.
- Multi-GPU study currently in-progress.
- Non-HPC language ports started as a hobby project.

ParRes / Kernels

Watch 36 | Star 261 | Fork 79

Code | Issues 17 | Pull requests 5 | Actions | Projects | Wiki | Security | Insights

default | 11 branches | 6 tags | Go to file | Code

jeffhammond Fix nstream memcpy target (#538) ... e894147 10 days ago 3,237 commits

.github	simplify template	last month
AMPI	avoid overflow	2 years ago
C1z	Fix nstream memcpy target (#538)	10 days ago
CHARM++	avoid overflow	2 years ago
Csharp	make input parsing work with older C# compilers	last month
Cxx11	device not thread num	17 days ago
FENIX	Finalizing Fenix Transpose kernel.	4 years ago
FG_MPI	remove C99 flag (#454)	11 months ago
FORTRAN	Fix nstream memcpy target (#538)	10 days ago
GO	DGEMM works	last month
GRAPPA	Fixing makefile comment about default shape for stencils.	4 years ago
JAVA	make may not be the right thing for Java but it is okay for now	7 months ago

About

This is a set of simple programs that can be used to explore the features of a parallel platform.

groups.google.com/forum/#!forum/...

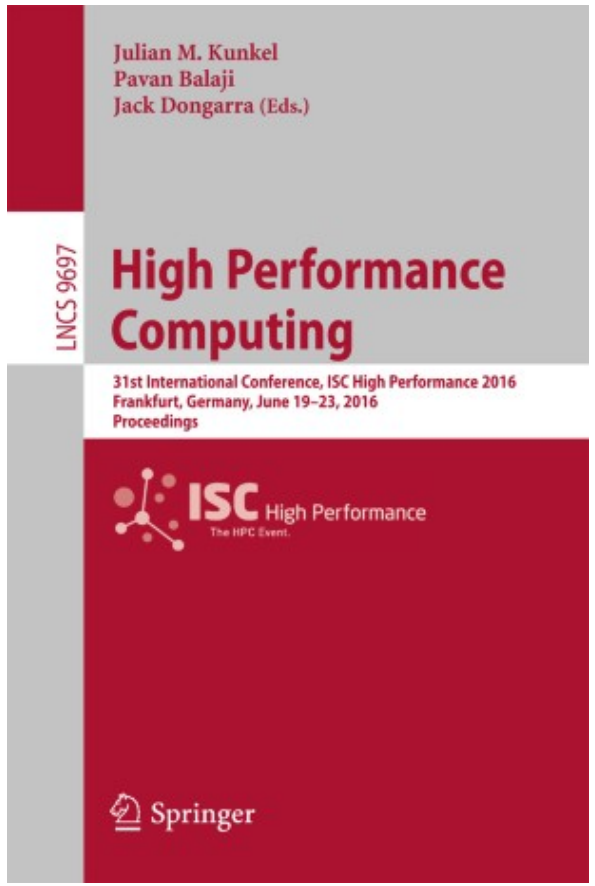
c | c-plus-plus | travis-ci | julia | opencl | boost | openmp | mpi | parallel-computing | python3 | pgas | coarray-fortran | threading | tbb | kokkos | shmem | charmplusplus | sycl | parallel-programming | fortran2008

Readme

View license

Releases

6 tags



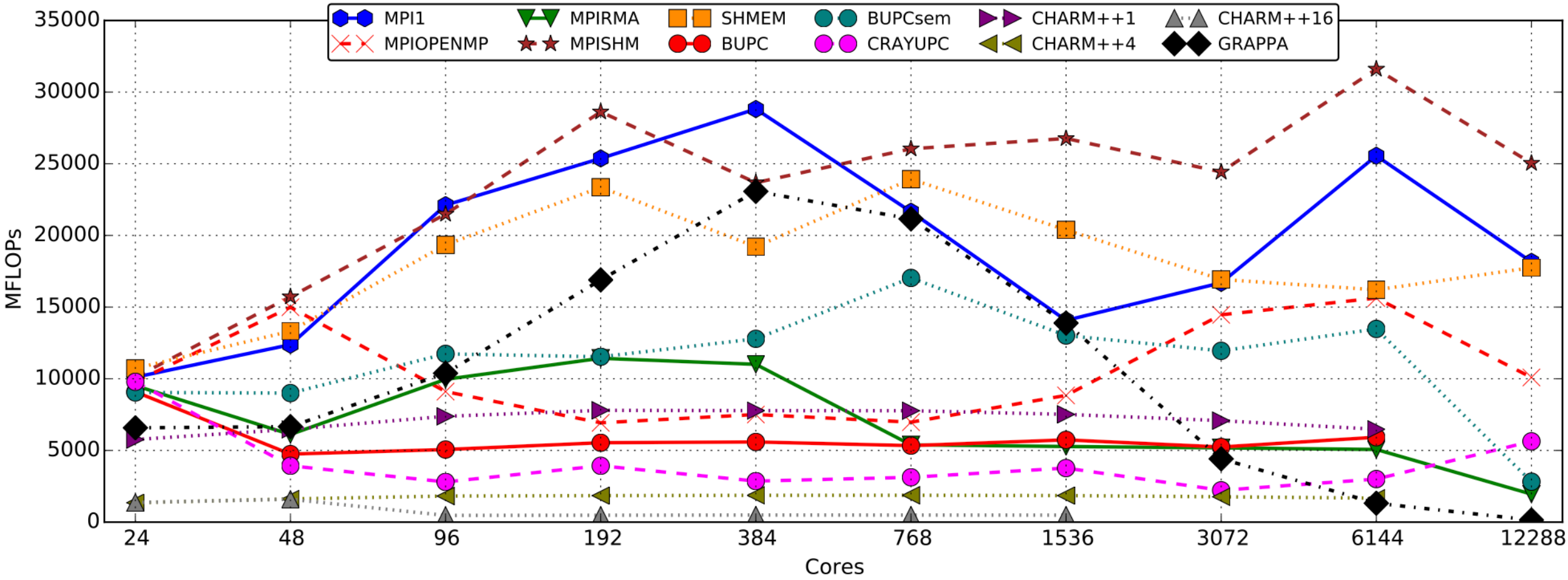
Comparing runtime systems with exascale ambitions using the Parallel Research Kernels

R. F. Van der Wijngaart[1], A. Kayi[1], J. R. Hammond[1], G. Jost[1], T. St. John[1], S. Sridharan[1], T. G. Mattson[1], J. Abercrombie[2], and J. Nelson[2]

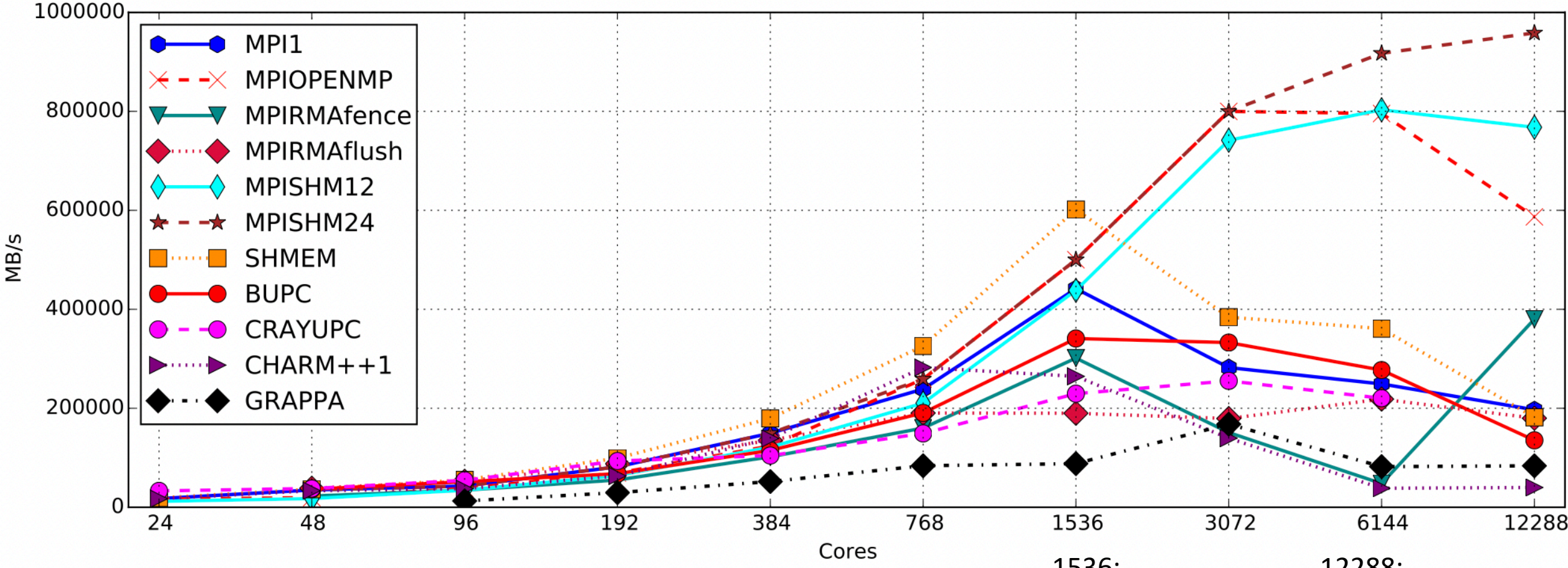
[1] Intel Corporation, Hillsboro, Oregon, USA.
[2] University of Washington, Seattle, WA, USA.

https://link.springer.com/book/10.1007/978-3-319-41321-1

synch_p2p


MPI1
MPIOPENMP
MPIRMA
MPISHM
SHMEM
BUPC
BUPCsem
CRAYUPC
CHARM++1
CHARM++4
CHARM++16
GRAPPA
MFLOPs
35000
30000
25000
20000
15000
10000
5000
0
24
48
96
192
384
768
1536
3072
6144
12288
Cores

transpose



1536:
8 KiB/PE (flat)
4.5 MiB/PE (hybrid)

12288:
128 B/PE (flat)
72 KiB/PE (hybrid)

MPI for Exascale! Now what?

Something tells me that C89 and OpenMP 3.0 aren’t enough…

Exascale hypotheses and goals

- Modern C++ will be important: e.g. Kokkos and RAJA.
- Almost everyone who doesn’t use modern C++ will use OpenMP.
- Users of proprietary/non-portable programming models will need to be convinced with data that standards are good enough.
- Below the top 10, we will see increased use of new or otherwise “non-HPC” programming models, e.g. Python and Julia.
- Other new things will emerge (e.g. Khronos SYCL) that need to be evaluated objectively.

Language	Seq.	OpenMP	MPI	PGAS	Threads	Others?
C89	√	√	Many	SHMEM		
C11	√	√√√		UPC	√	~~Cilk~~, ISPC, PETSc
C++17	√	√√√	RMA (WIP)	Grappa	√	Kokkos, RAJA, TBB, PSTL, SYCL 1.2.1 and 2020, OpenCL 1.2, CUDA, HIP, …
Fortran	√	√√√		Coarrays, GA		“pretty”, OpenACC
Python	√		√			Numpy
Chapel	√		?	√		

√√√ = Traditional, task-based, and target are implemented similarly in Fortran, C and C++.

C++ analysis

Evaluating data parallelism in C++ programming models using the Parallel Research Kernels

2018: https://www.ixpug.org/images/docs/KAUST_Workshop_2018/IXPUG_Invited2_Hammond.pdf

2019: https://drive.google.com/file/d/1yNQiG-wjBI4Iu6yDPV6WcQL-r8Yt9RSV/view?usp=sharing

https://dl.acm.org/doi/10.1145/3318170.3318192

C++ Summary

Model	for	for^N	Reduce, Scan	Memory management	Hardware
TBB parallel_for	√	√	√	C++	CPU
C++17 PSTL	√	-	√	C++	CPU, ~GPU
RAJA	√	√	√	√	CPU, CPU
Kokkos	√	√	√	√	CPU, GPU
Boost.Compute	√	-	√	√	See OpenCL
OpenCL	√	3	-	√	CPU, GPU, FPGA, etc.
SYCL 1.2.1	√	3	-	√	See OpenCL
SYCL 2020	√	3	√	√+	CPU, GPU, FPGA, etc. (OpenCL optional)
OpenMP 5	√	√	√	√	CPU, ~GPU

What we have observed is that HPC programmers want multidimensional arrays (3+) and loops, “collectives” and flexible memory management (host, device, unified). There is still a lot of work to be done to bring these features into the mainstream ISO languages.

Performance Experiments

https://github.com/ParRes/Kernels/tree/default/Cxx11

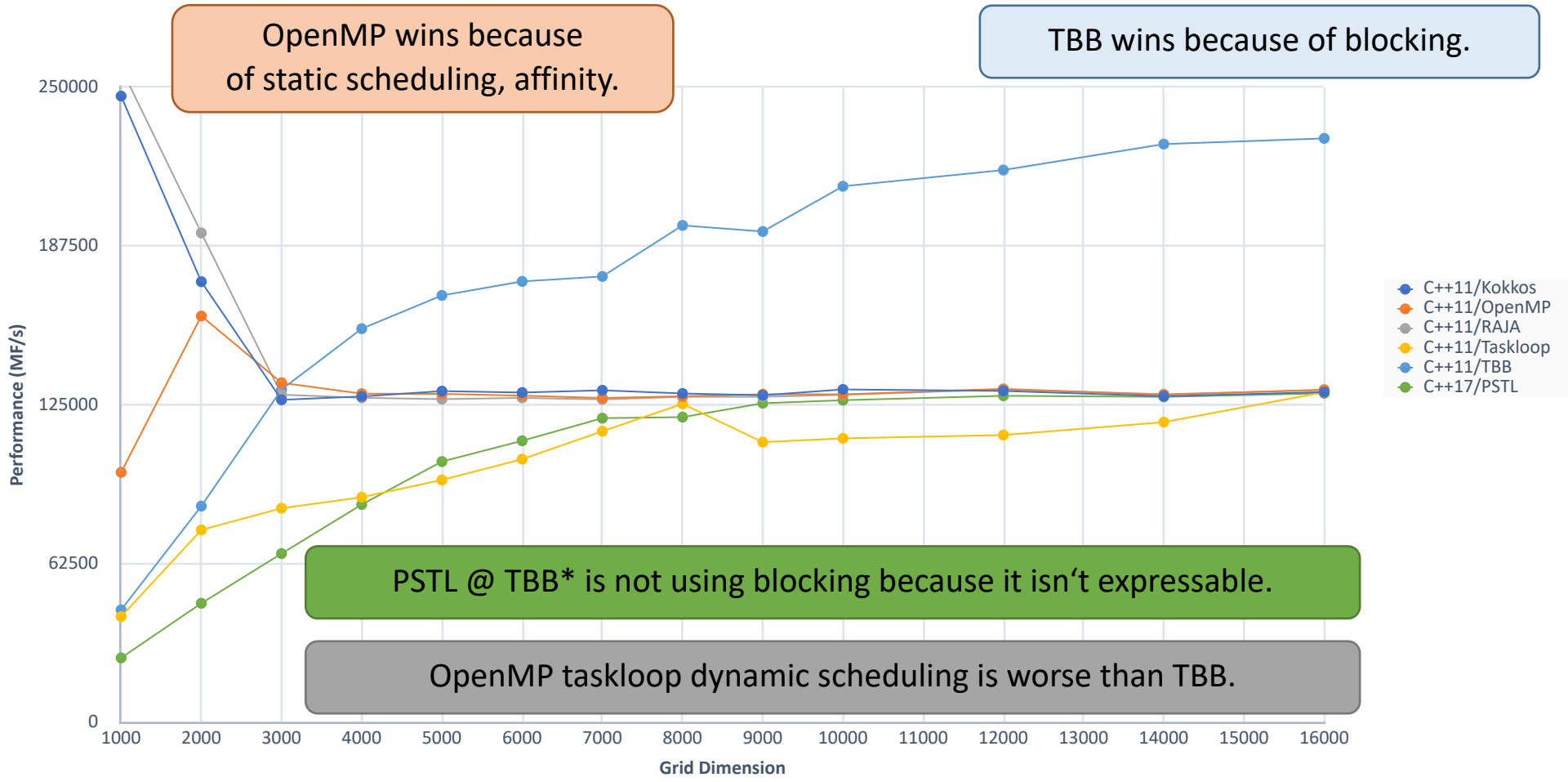
https://github.com/ParRes/Kernels/tree/default/C1z

https://github.com/ParRes/Kernels/tree/default/FORTRAN

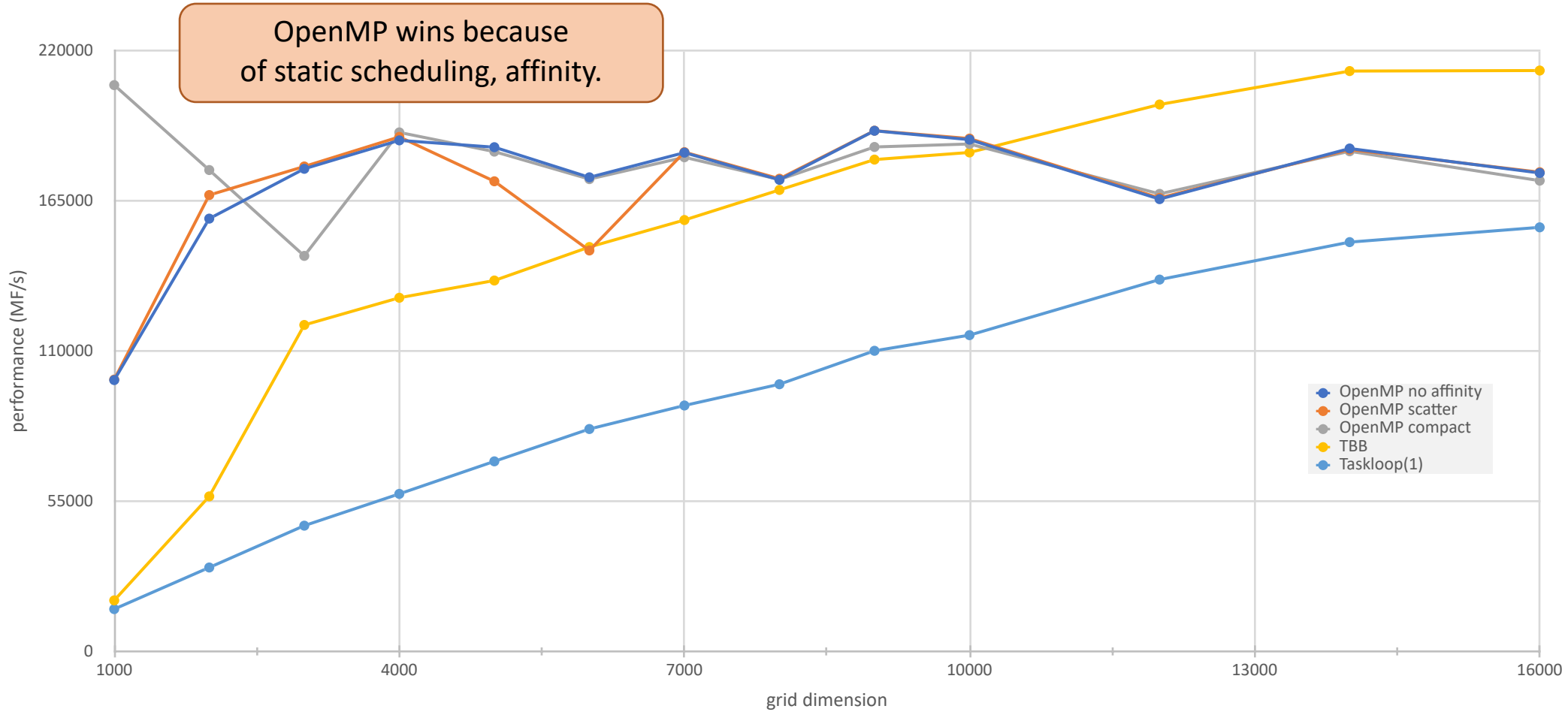
Performance disclaimer

- None of the PRK implementations are tuned for any architecture.
- The only PRK that achieves a large fraction of peak is nstream.
- The only tuning parameter is a blocking factor, which helps with TLB misses when running on a CPU.
- This data is only intended for use in evaluating programming models.
- Details are available upon request but are omitted because *relative performance matters when comparing software*.

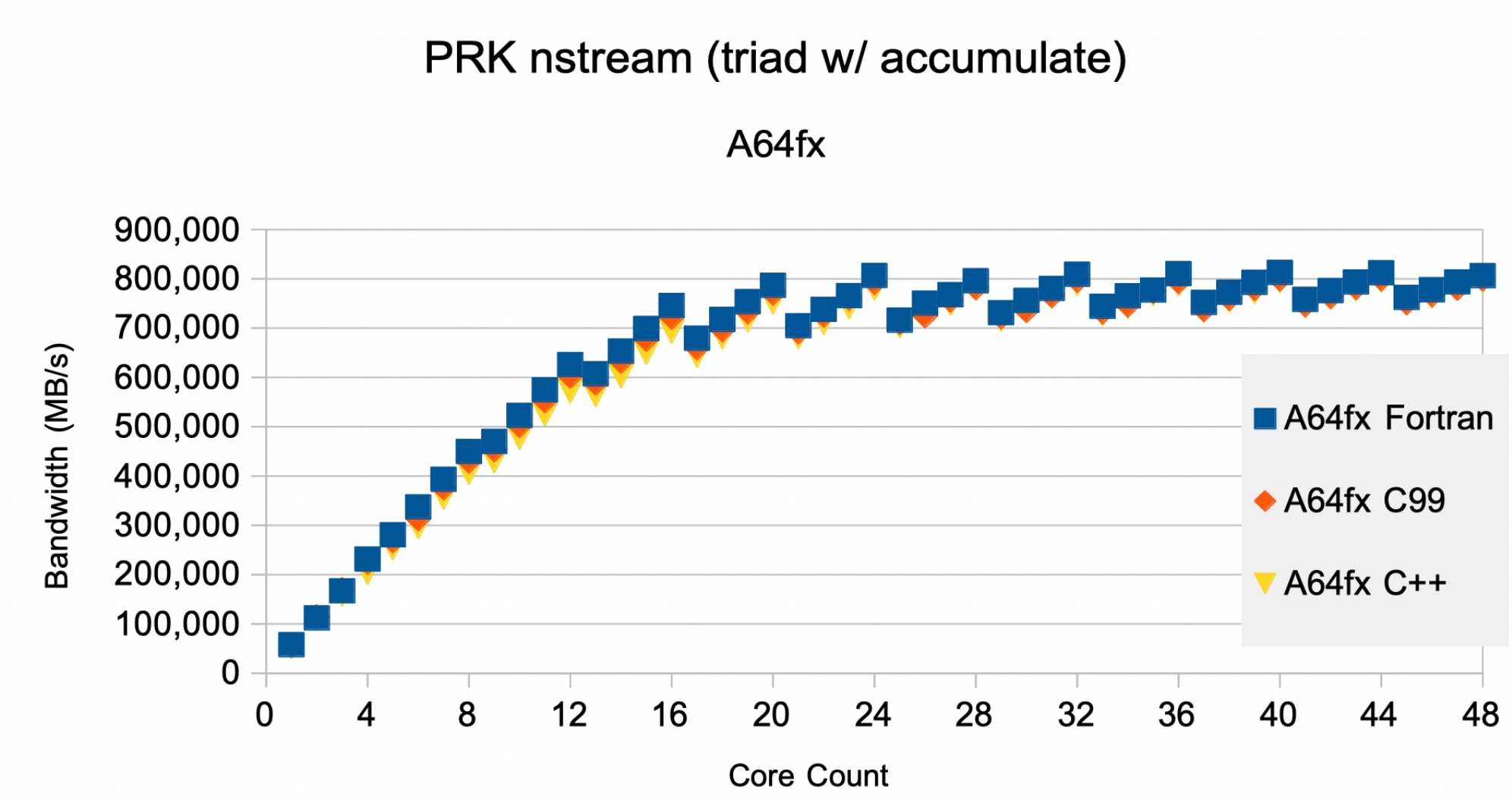
PRK stencil: C++ implementations on KNL



Improved PRK stencil on KNL (MCDRAM flat)

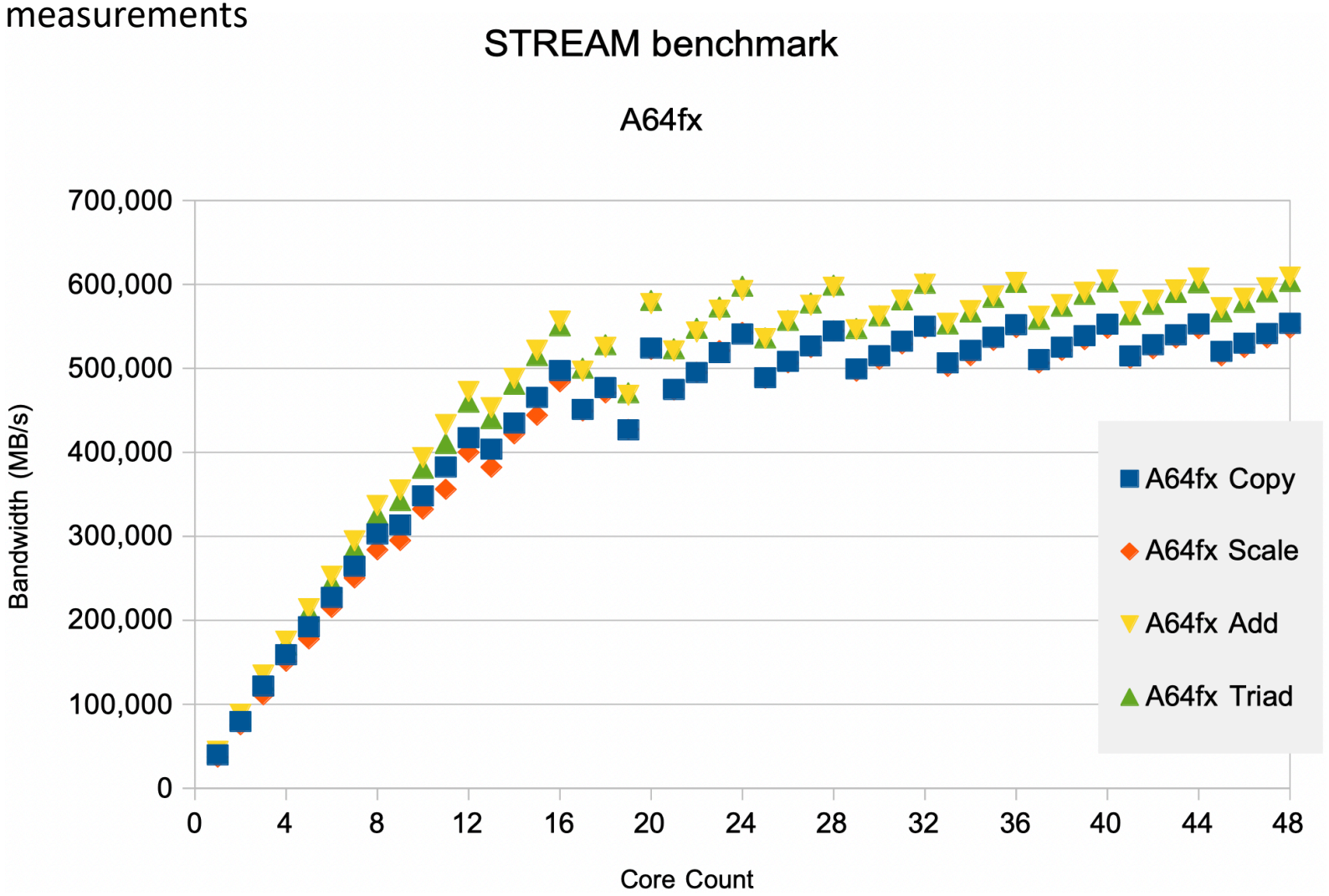


A64fx measurements



*"This research was supported by the NSF MRI award #**1828187**: "MRI: Acquisition of an HPC System for Data-Driven Discovery in Computational Astrophysics, Biology, Chemistry, and Materials Science."*

A64fx measurements

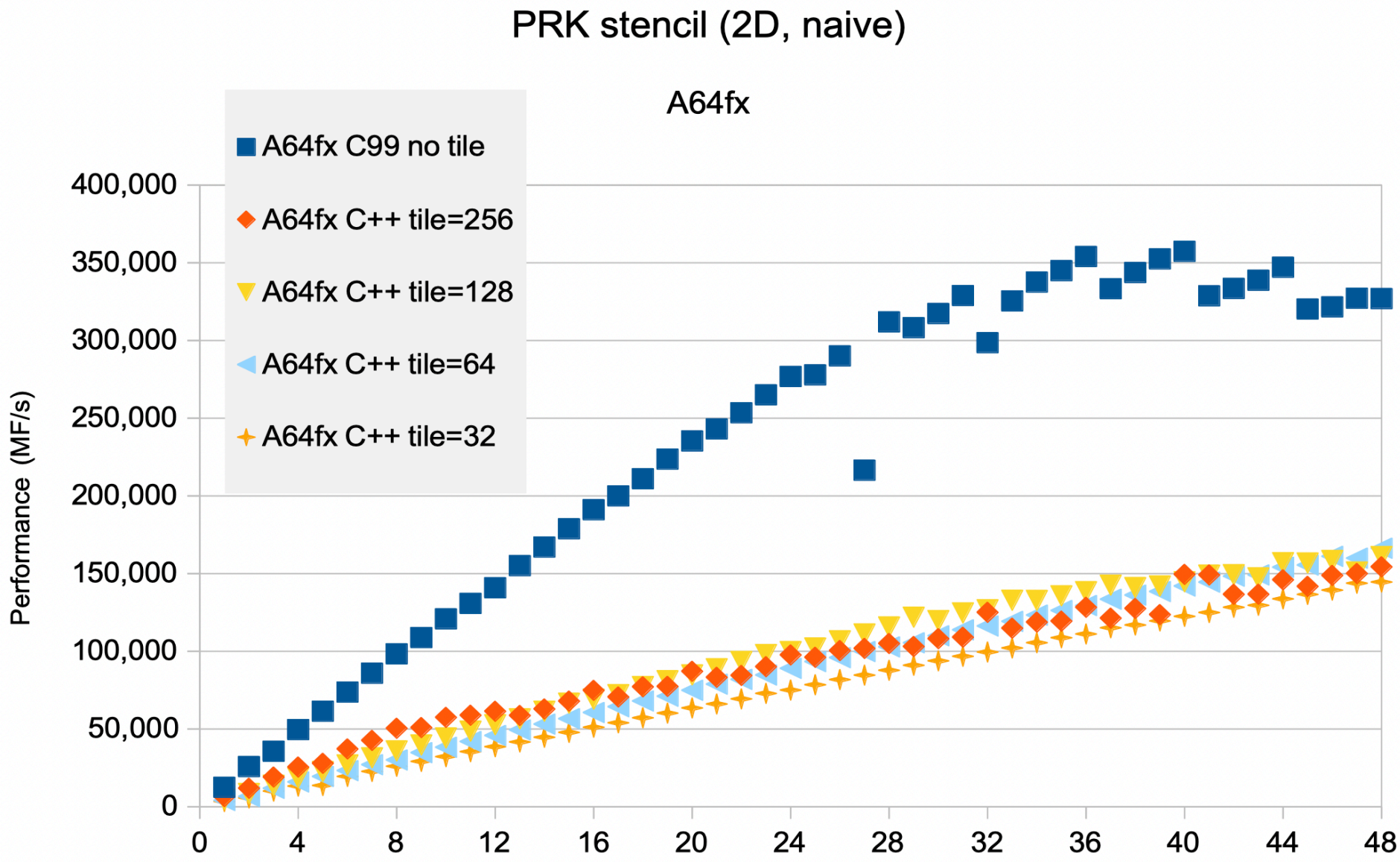


When bandwidth is measured with overwriting instead of accumulation, there is a strong dependence on compiler options and code generation, which isn't the point of a bandwidth benchmark.

https://community.arm.com/developer/research/m/resources/993/download explains the compiler dependencies here.

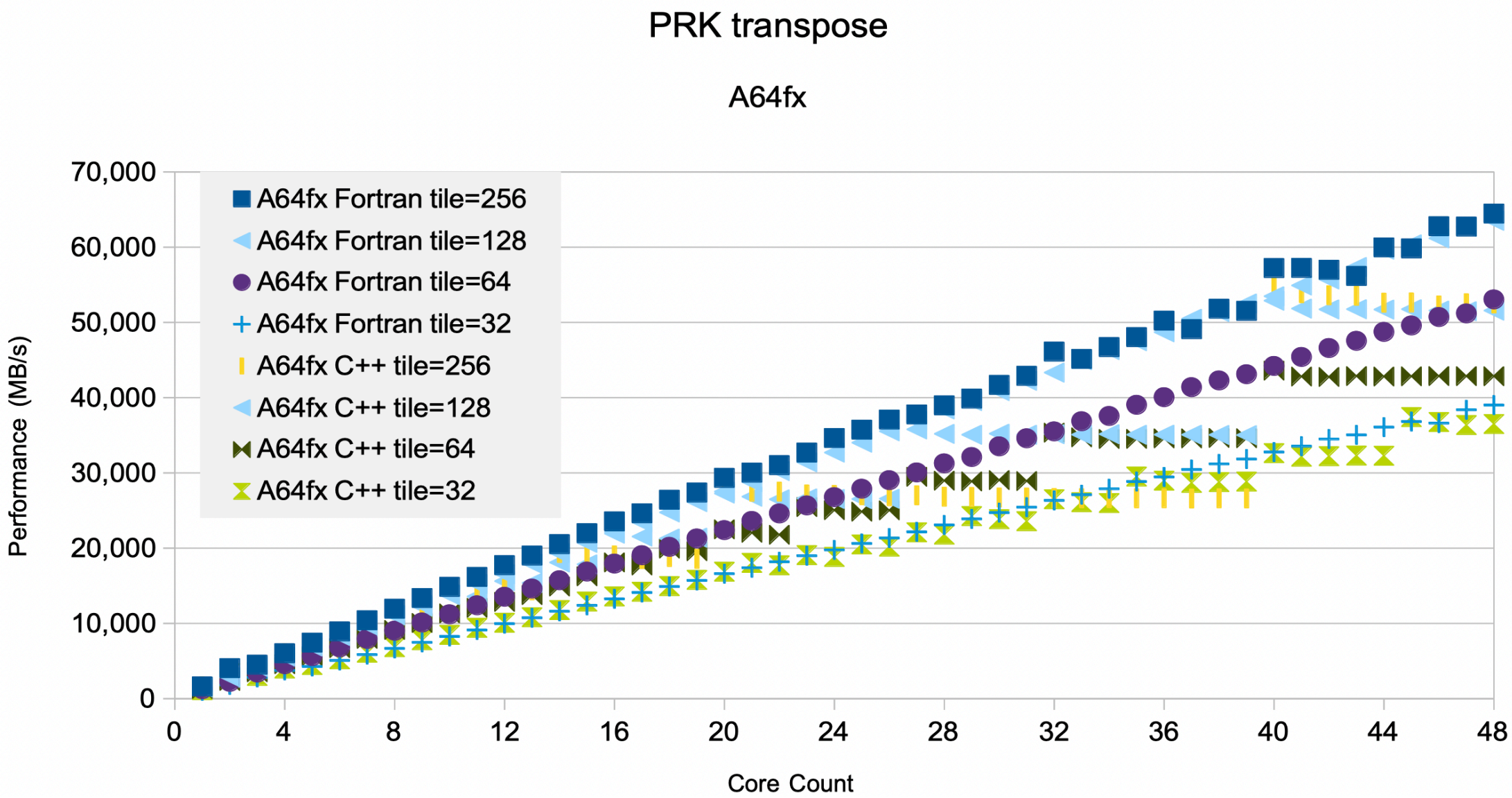
"*This research was supported by the NSF MRI award #**1828187**: "MRI: Acquisition of an HPC System for Data-Driven Discovery in Computational Astrophysics, Biology, Chemistry, and Materials Science.*"

A64fx measurements



"*This research was supported by the NSF MRI award #**1828187**: "MRI: Acquisition of an HPC System for Data-Driven Discovery in Computational Astrophysics, Biology, Chemistry, and Materials Science.*"

A64fx measurements



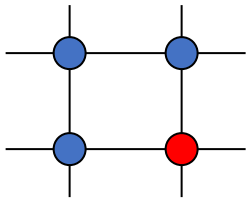
*"This research was supported by the NSF MRI award #**1828187**: "MRI: Acquisition of an HPC System for Data-Driven Discovery in Computational Astrophysics, Biology, Chemistry, and Materials Science."*

Wavefront Parallelism

Look for */p2p*.*

Wavefront Parallelism

```
// sequential C implementation
for (int i=1; i<m; ++i) {
  for (int j=1; j<n; ++j) {
    A[i][j] = A[i-1][j] + A[i][j-1] - A[i-1][j-1];
  }
}
```

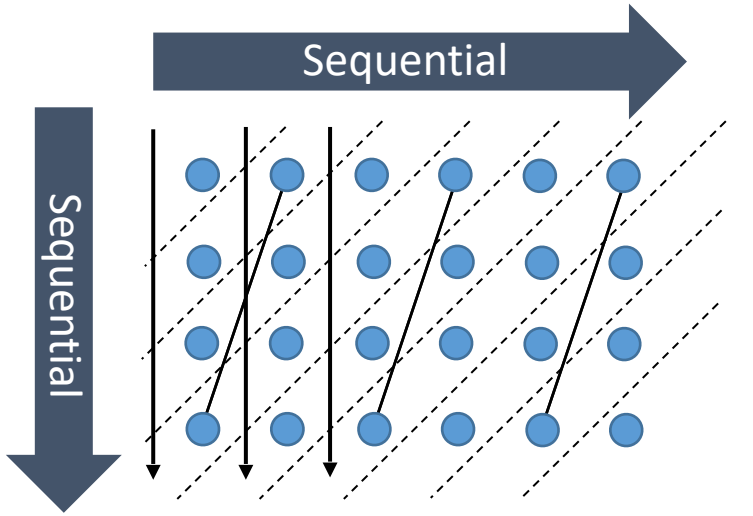


$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$

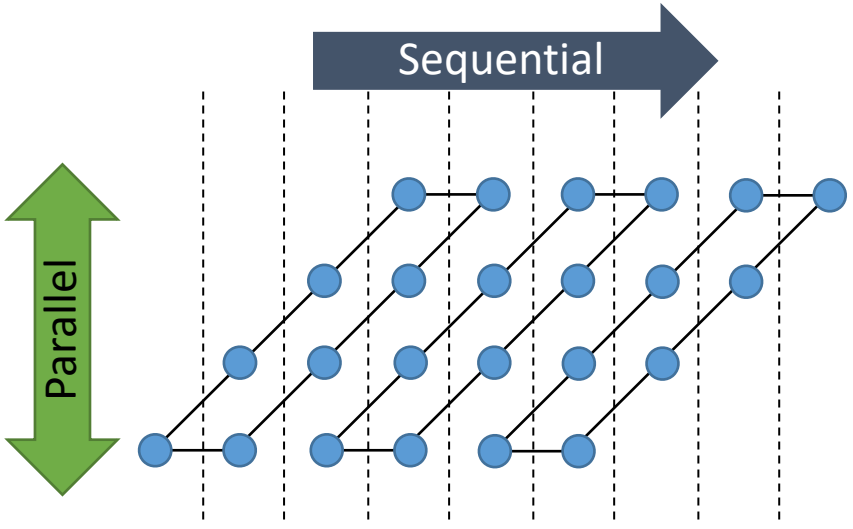
This *pattern* appears in a range of applications:

- Deterministic neutron transport (DOE-NNSA mission science)
- Smith-Waterman/PairHMM (bioinformatics)
- Dynamic programming
- Linear algebra (e.g. NAS LU benchmark)

Changing the iteration space exposes parallelism



I-loop goes to the right,
J-loop goes down

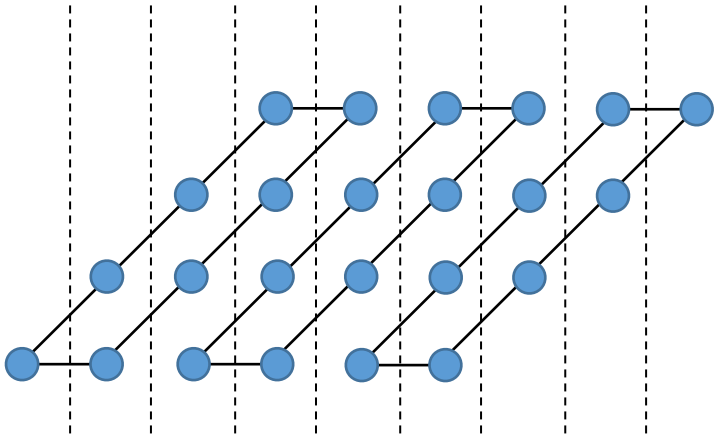


Loop over anti-diagonals
(see next slide)

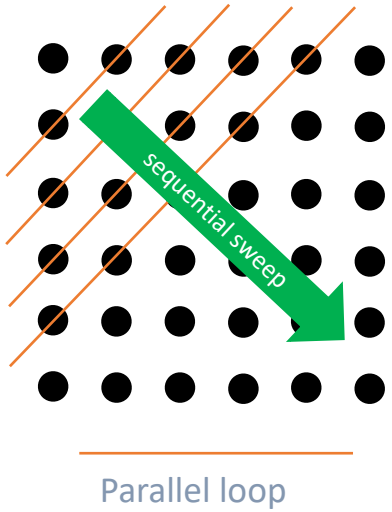
OpenMP inner-loop parallelism

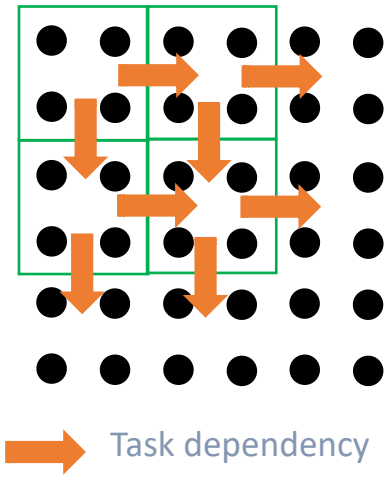
```
// sequential loop
for (int i=2; i<=2*n-2; ++i) {
  int start = max(2,i-n+2);
  int stop  = min(i,n);
  #pragma omp for simd
  for (int j=start; j<=stop; ++j) {
    const int x = i-j+2-1;
    const int y = j-1;
    A[x][y] = A[x-1][y]
            + A[x][y-1]
            - A[x-1][y-1];
  }
  // implicit barrier (required)
}
```

- Very low parallel efficiency once data spills private cache.
- CPU SIMD doesn't work because data access is non-contiguous.



Amortizing synchronization overheads

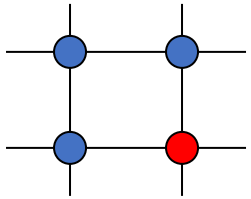




- Sequential execution requires no synchronization.
- Formally, there are $O(n^2)$ element-wise dependencies.
- Antidiagonal implementation uses O(n) barriers to enforce deps.
- Hyperplane amortizes barriers across many antidiagonals: O(n/unroll) barriers.
- Task-based has $O(n^2/block^2)$ dependencies.

OpenMP task-based parallelism

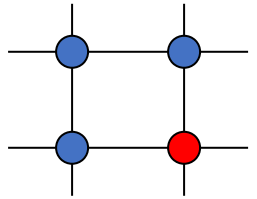
```
#pragma omp parallel
#pragma omp master
for (int i=1; i<m; i+=mc) {
  for (int j=1; j<n; j+=nc) {
    #pragma omp task depend(in:grid[i-mc][j],grid[i][j-nc]) \
                     depend(out:grid[i][j])
    for (int ii=i; ii<std::min(m,i+mc); ii++) {
      for (int jj=j; jj<std::min(n,j+nc); jj++) {
        A[ii][jj] = A[ii-1][jj] + A[ii][jj-1] - A[ii-1][jj-1];
      }
    }
  }
}
#pragma omp taskwait
```



$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$

OpenMP “doacross” parallelism

```
#pragma omp for collapse(2) ordered(2)
for (int i=0; i<ib; i++) {
  for (int j=0; j<jb; j++) {
    #pragma omp ordered depend(sink: i-1,j) depend(sink: i,j-1)
    for (int ii=i; ii<std::min(m,i+mc); ii++) {
      for (int jj=j; jj<std::min(n,j+nc); jj++) {
        A[ii][jj] = A[ii-1][jj] + A[ii][jj-1] - A[ii-1][jj-1];
      }
    }
    #pragma omp depend(source)
  }
}
```



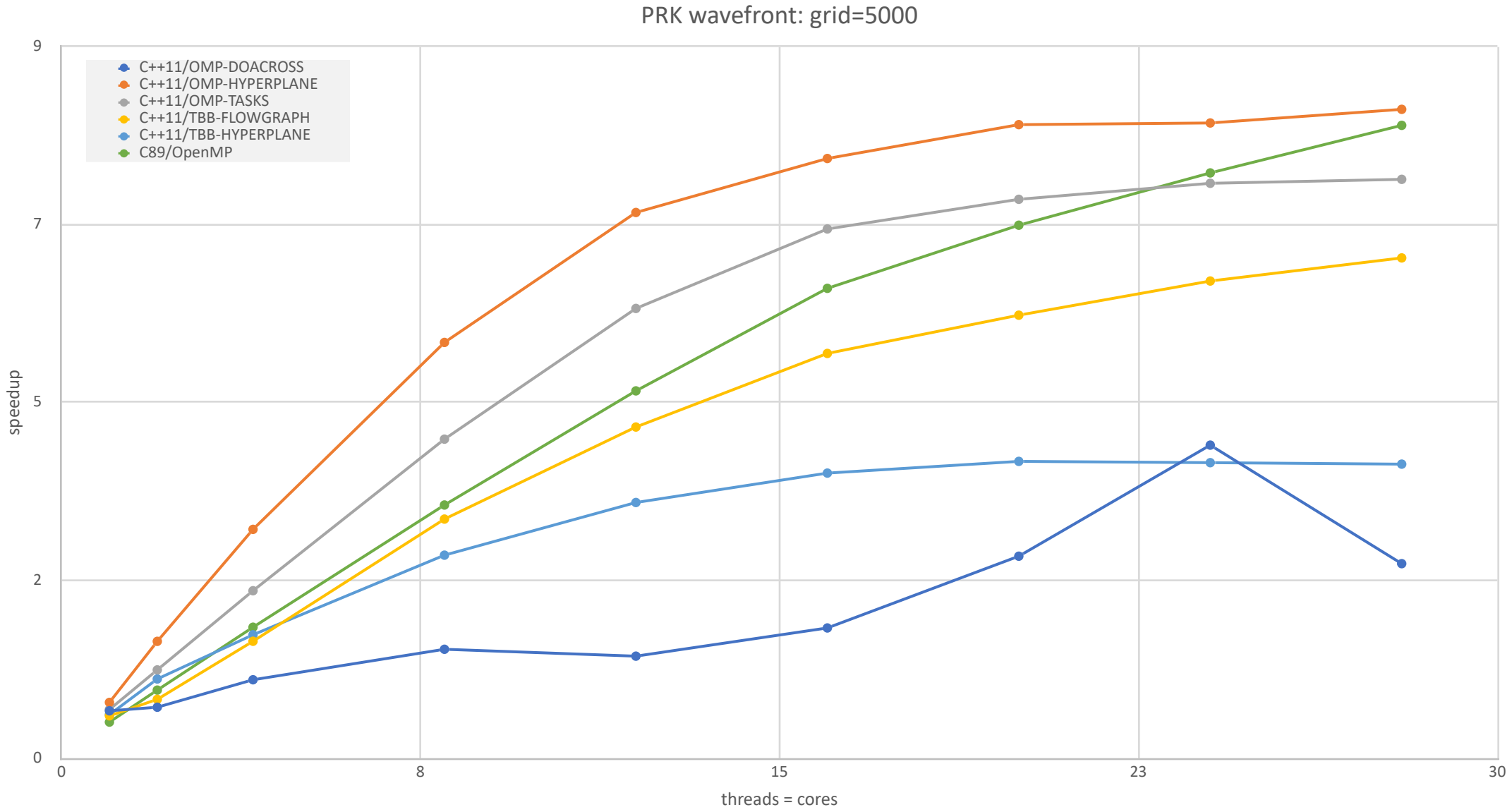
$A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$

OpenMP hyperplane parallelism

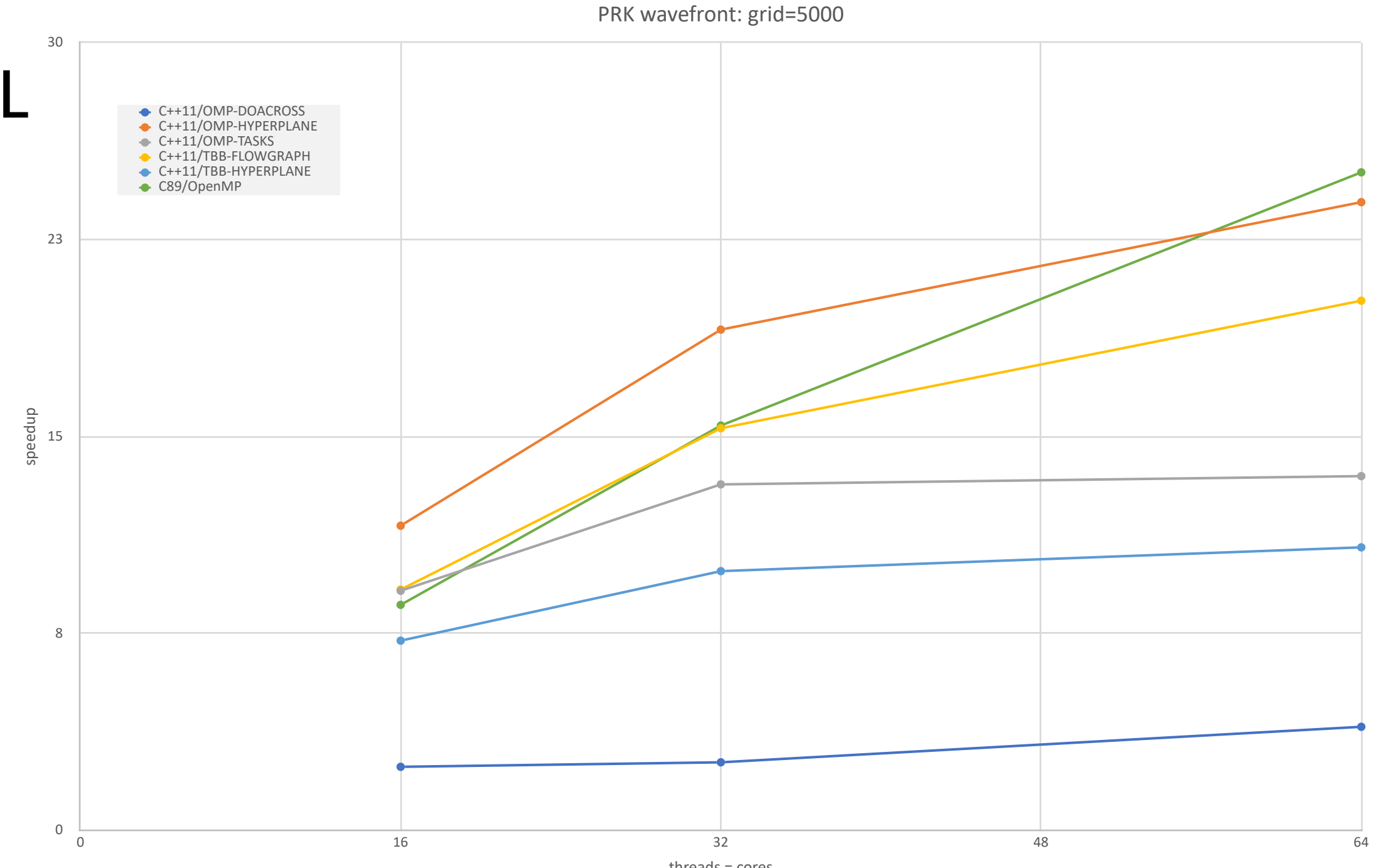
```
#pragma omp parallel
for (int i=2; i<=2*(nb+1)-2; i++) {
  #pragma omp for
  for (int j=std::max(2,i-(nb+1)+2); j<=std::min(i,nb+1); j++) {
    const int ib = nc*(i-j)+1;
    const int jb = nc*(j-2)+1;
    for (int ii=ib; ii<std::min(m,ib+nc); ii++) {
      for (int jj=jb; jj<std::min(n,jb+nc); jj++) {
        A[ii][jj] = A[ii-1][jj] + A[ii][jj-1] - A[ii-1][jj-1];
      }
    }
  }
}
```

This is only implemented for square grids to keep the polyhedral arithmetic simpler.

SKX 8180 (1S)


PRK wavefront: grid=5000
C++11/OMP-DOACROSS
C++11/OMP-HYPERPLANE
C++11/OMP-TASKS
C++11/TBB-FLOWGRAPH
C++11/TBB-HYPERPLANE
C89/OpenMP
speedup
0
2
5
7
9
0
8
15
23
30
threads = cores

KNL


PRK wavefront: grid=5000
C++11/OMP-DOACROSS
C++11/OMP-HYPERPLANE
C++11/OMP-TASKS
C++11/TBB-FLOWGRAPH
C++11/TBB-HYPERPLANE
C89/OpenMP
speedup
30
23
15
8
0
0
16
32
48
64
threads = cores

PRK transpose: CPU vs GPU (integrated)

order=4000, tile=32	MF/s
C++11/OpenMP TARGET tiled	1972
C++11/OpenMP TARGET untiled	8781
C++11/OpenMP host tiled	10351
C++11/OpenMP host untiled	6418
C++11/OpenCL CPU untiled	6110
C++11/OpenCL GPU untiled	11803

PRK stencil: CPU vs GPU (integrated)

order=4000, tile=32, star, radius=4	MF/s
C++11/OpenMP TARGET tiled	16229
C++11/OpenMP TARGET untiled	16565
C++11/OpenMP host tiled	21826
C++11/OpenMP host untiled	11654
C++11/OpenCL CPU untiled	25106
C++11/OpenCL GPU untiled	5692

What if we used the same methods to evaluate non-HPC programming languages?

Language	Nstream	Transpose	Stencil	Synch_p2p	Dgemm
C#	√	√			
Go	√	√			√
Java	√	√	√	√	
Julia	√√	√√	√	√	
Lua	√				
Octave (Matlab)	√√	√√	√√	√	
Python	√√√	√√√	√√√	√	√√o
Ruby	√				
Rust	√	√	√	√	√
Scala	√				

√√ = Traditional (loops) and Pretty (higher-level)
√√√ = Traditional (loops), Numpy, MPI (mpi4py)

Do not assume that Jeff knew what he was doing when he wrote Go, Lua, Rust, … well anything frankly.

PRK nstream language showdown

Language	MB/s
C89	31,906
Fortran	33,221
C11	34,529
C++11	33,460
Rust	21,576
Java	32,264
Go	24,546
Julia loops	34,193
Julia pretty	34,645
Numpy	17,229
Octave colon	9,804
Lua	*273*
Python	*262*
Octave loops	*4*

Language	MB/s
C89	27,073
Fortran pretty	27,369
Fortran loops	27,448
C11	30,014
C++11	27,593
C++11 vector	27,317
C++11 valarray	27,219
C++11 range-for	27,515
C++11 for_each	23,951
Rust	20,018
Java	27,693
Go	21,534
Julia loops	27,544
Julia pretty	27,500
Numpy	11,568

1Mi (left) and 64Mi (right) elements, GCC 9, Ubuntu 20.04, Hades Canyon NUC

PRK transpose language showdown

Language	MB/s	Comments
C89	1,590	
Fortran pretty	1,646	Two passes: $B+=A^T$; A+=1. Not blocked.
Fortran loops	3,605	
C11	1,664	
C++11	1,546	
C++11 vector	2,640	
C++11 valarray	2,609	
C++11 range-for	1,510	Block range ctor in block loops
C++11 for_each	1,178	Not blocked
Rust	1,432	
Java	3,843	
Julia loops	1,257	
Julia pretty	1,499	
Numpy	2,008	

8Ki rank matrix, GCC 9, Ubuntu 20.04, Hades Canyon NUC

MPI: Python vs C

Nstream

procs (cores)	mpi4py+numpy	MPI1+C89
1	11480	29996
2	14046	31729
4	13685	31057

Transpose

procs (cores)	mpi4py+numpy	MPI1+C89
1	1443	1599
2	2557	3002
4	4081	4322

Same details as before…

Preliminary analysis

- I am still in shock at how well Java does (h/t Yijian).
- Julia is the real deal, if you use it correctly (h/t Kiran).
- Numpy results sensitive to seemingly unrelated settings (OpenMP).
- Neither Rust nor Go threatens C11 on performance.
- C vs C++ vs Fortran can’t be settled with toy codes and I need to read some assembly and hardware counters to say anything useful.
- mpi4py is awesome but requires old MPI dogs to learn new tricks.

Summary

- The PRK project is a research project for understanding:
  - Programming model semantics and syntax, e.g. SYCL vs Kokkos
  - Programming language differences, e.g. Fortran vs Julia, C++ vs Python
  - Toolchain implementation quality, e.g. GCC vs LLVM
  - Communication semantics and implementation details, e.g. MPI vs SHMEM
  - Anything else where quick-and-easy answers are sufficient
- The PRK project is not particularly useful for understanding absolute performance or hardware characteristics *unless you tune the code appropriately.*

Conclusions

- The Parallel Research Kernels have proven to be an effective experimental apparatus for researchers at Intel and elsewhere to understand how computers work.
- The new results (gathered yesterday) did not meet any of Torsten's rules for good performance analysis! Do not reuse any of this data for any serious purpose.
- Consult the experts (e.g. Lisandro) but also encourage new contributors (e.g. Yijian).

References

- R. F. Van der Wijngaart, A. Kayi, J. R. Hammond, G. Jost, T. St. John, S. Sridharan, T. G. Mattson, J. Abercrombie, and J. Nelson. ISC 2016. *Comparing runtime systems with exascale ambitions using the Parallel Research Kernels.*
- E. Georganas, R. F. Van der Wijngaart and T. G. Mattson. IPDPS 2016. *Design and Implementation of a Parallel Research Kernel for Assessing Dynamic Load-Balancing Capabilities*.
- R. F. Van der Wijngaart, S. Sridharan, A. Kayi, G. Jost, J. Hammond, T. Mattson, and J. Nelson. PGAS 2015. *Using the Parallel Research Kernels to study PGAS models.*
- R. F. Van der Wijngaart and T. G. Mattson. HPEC 2014. *The Parallel Research Kernels.*

The End