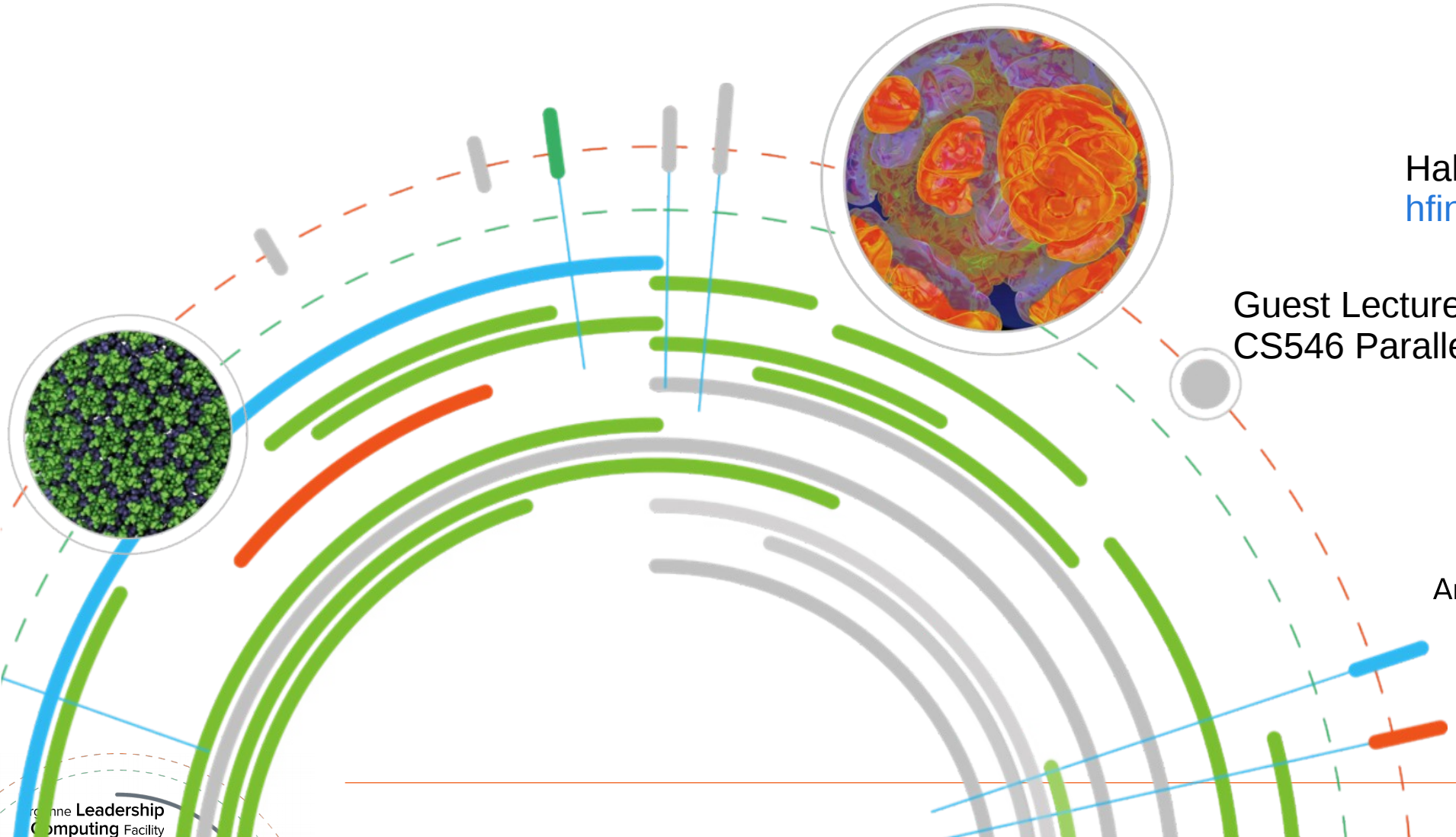
OpenMP Overview



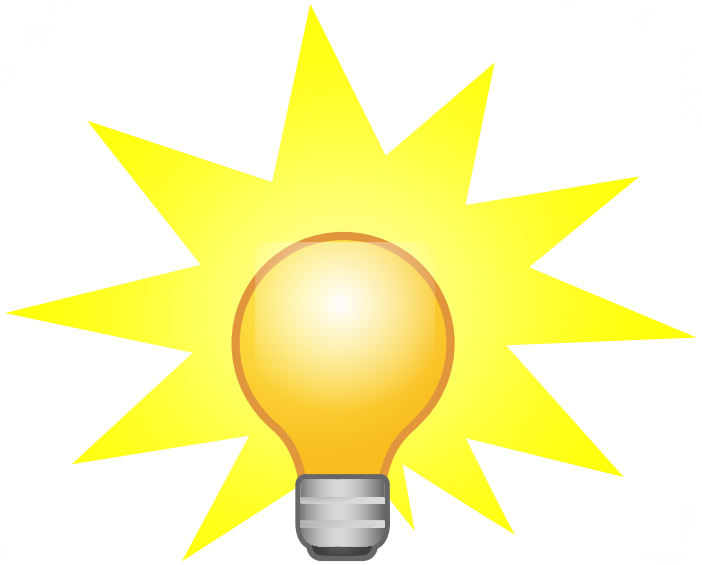
Hal Finkel
hfinkel@anl.gov

Guest Lecture – IIT – 2019-09-24:
CS546 Parallel and Distributed Processing

Argonne **Leadership Computing** Facility

Argonne NATIONAL LABORATORY

Some Background...



Types of parallelism

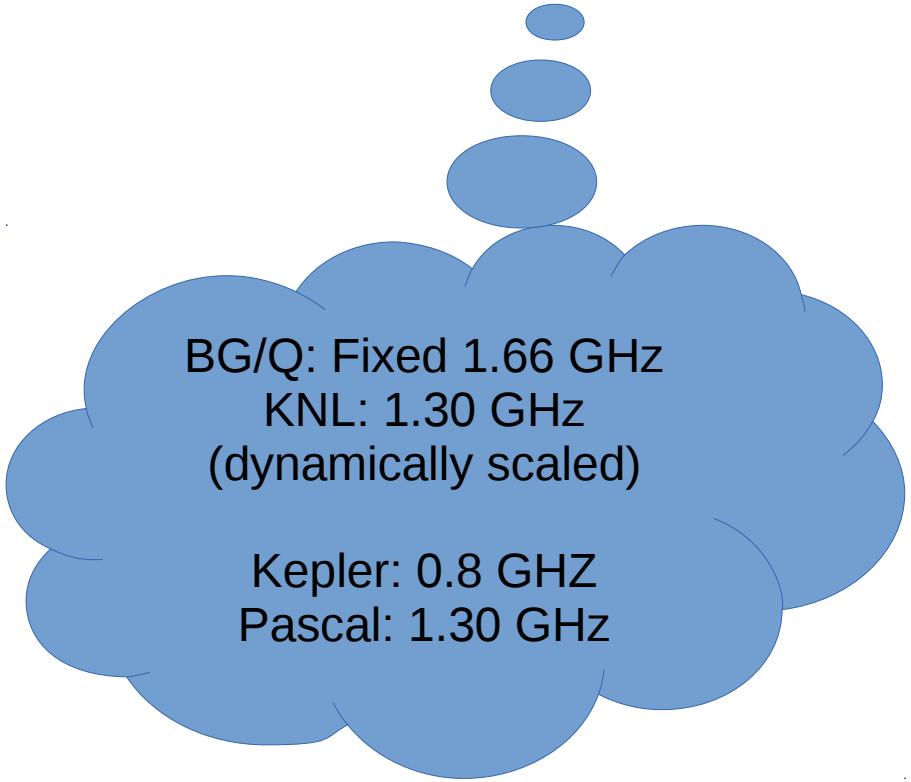
- Parallelism across nodes (using MPI, etc.)
- Parallelism across sockets within a node [Not applicable to the BG/Q, KNL, etc.]
- Parallelism across cores within each socket
- Parallelism across pipelines within each core (i.e. instruction-level parallelism)
- Parallelism across vector lanes within each pipeline (i.e. SIMD)
- Using instructions that perform multiple operations simultaneously (e.g. FMA)

Hardware threads tie in here too!

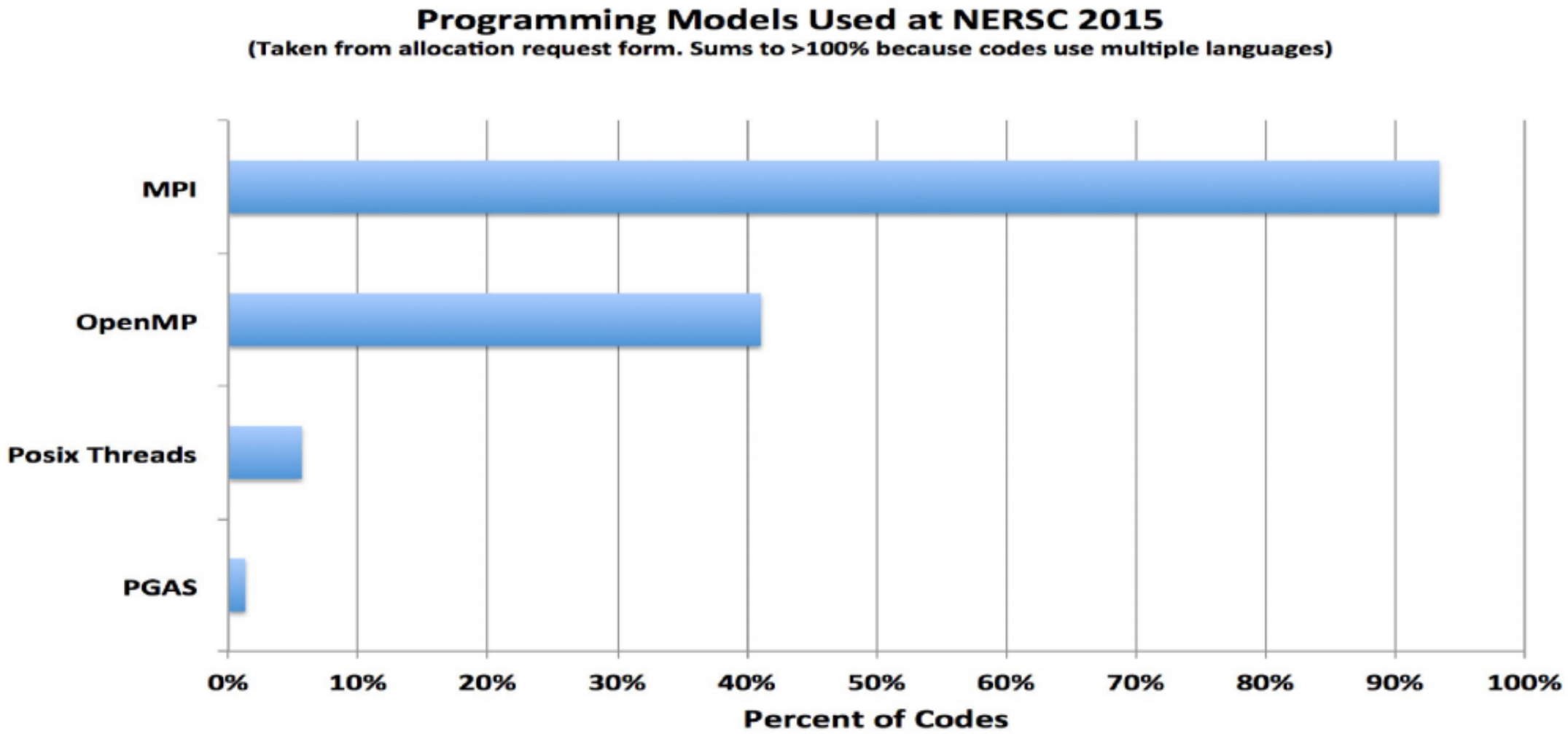
How fast can you go...

The speed at which you can compute is bounded by:

(the clock rate of the cores) x (the amount of parallelism you can exploit)



How do we express parallelism?

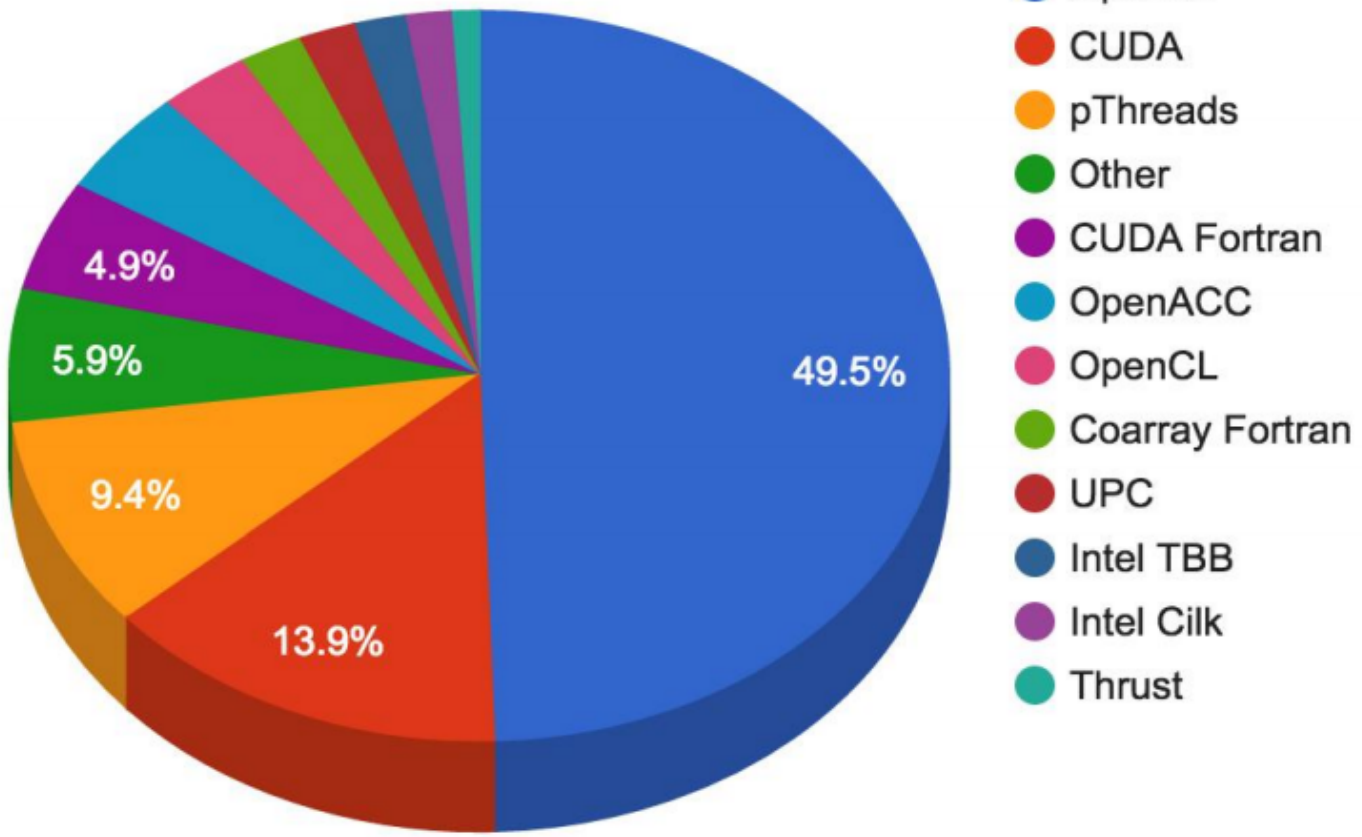


Courtesy of Yun (Helen) He, Alice Koniges, et. al., (NERSC) at OpenMPCon'2015

http://llvm-hpc2-workshop.github.io/slides/Tian.pdf

How do we express parallelism - MPI+X?

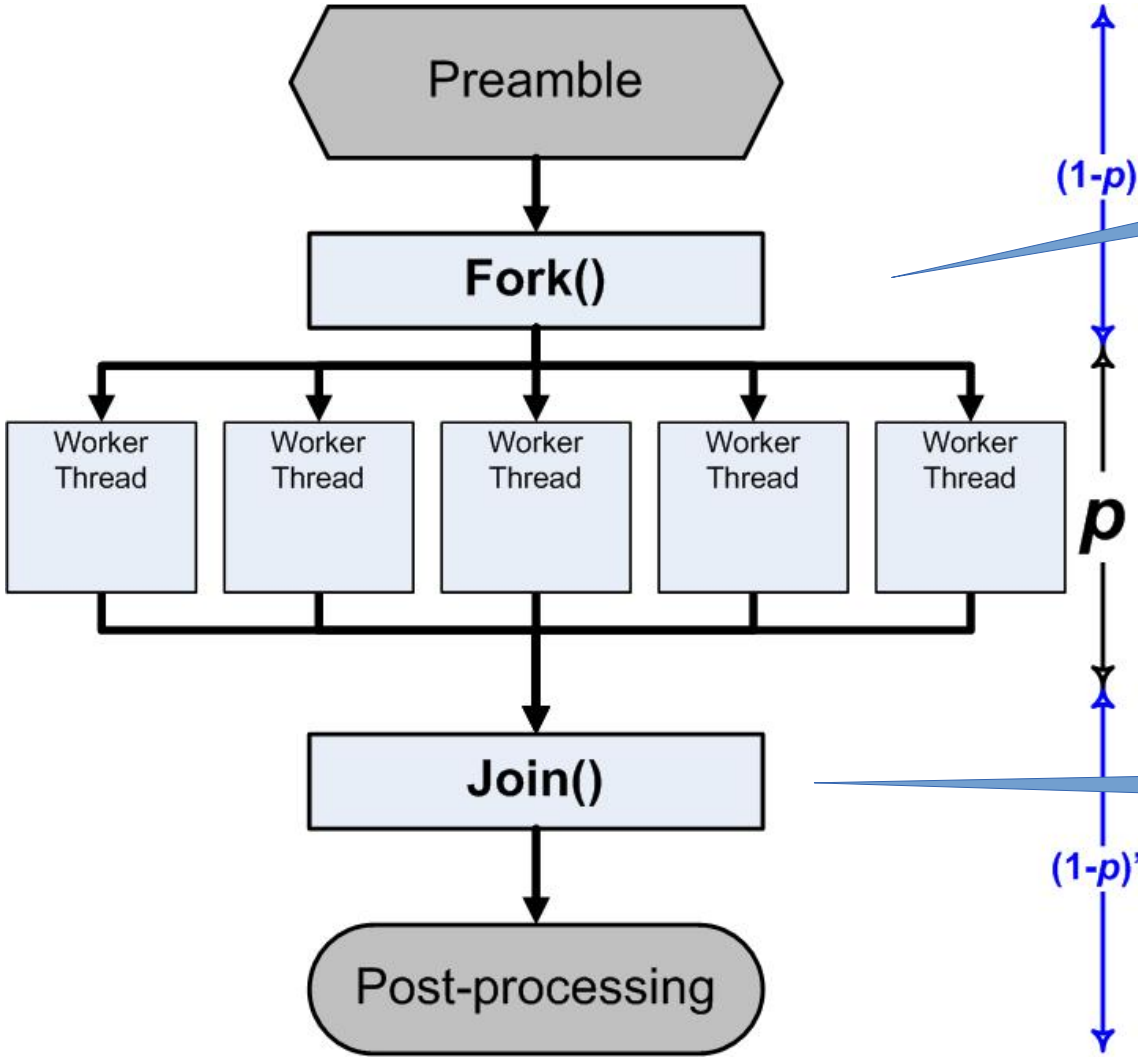
✓ OpenMP is about 50%, out of all choices of X



Courtesy of Yun (Helen) He, Alice Koniges, et. al., (NERSC) at OpenMPCon'2015

http://llvm-hpc2-workshop.github.io/slides/Tian.pdf

Coarse Grained vs. Fine Grained Parallelism



This is expensive
(many thousands of cycles)

Amdahl's law says the speedup is limied to: 1/(1 – p). So if 5% of the program remains serial, then the speedup from parallelization is limited to 20x.

https://en.wikipedia.org/wiki/Amdahl%27s_law

This is expensive too!

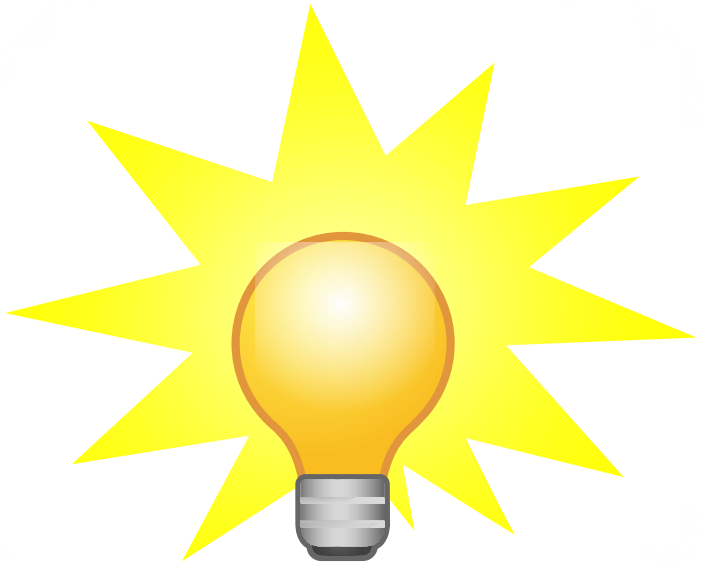
https://blogs.msdn.microsoft.com/ddperf/2009/04/29/parallel-scalability-isnt-childs-play-part-2-amdahls-law-vs-gunthers-law/

Compiling

Basic optimization flags...

- -O3 – Generally aggressive optimizations (try this first)
- -g – Always include debugging symbols (**really, always**! - when your run crashes at scale after running for hours, you want the core file to be useful)
- -fopenmp – Enable OpenMP (the pragmas will be ignored without this)
- -ffast-math (clang, gcc, etc.) – Enable “fast” math optimizations (most people don't need strict IEEE floating-point semantics).

OpenMP Basics...



OpenMP Basics

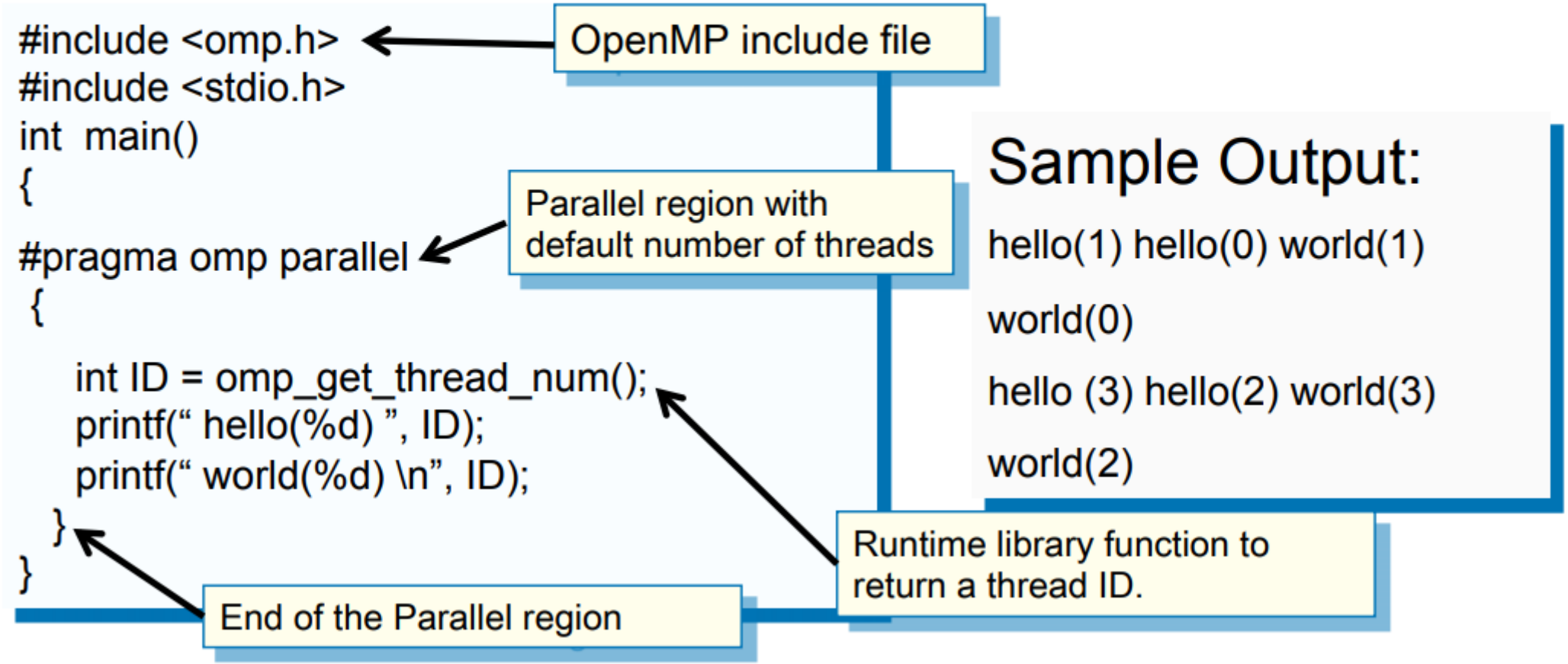
- Write a program that prints “hello world”.

```
#include<stdio.h>
int main()
{




    int ID = 0;

    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);

}
```


http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

OpenMP Basics

- Write a multithreaded program where each thread prints "hello world".

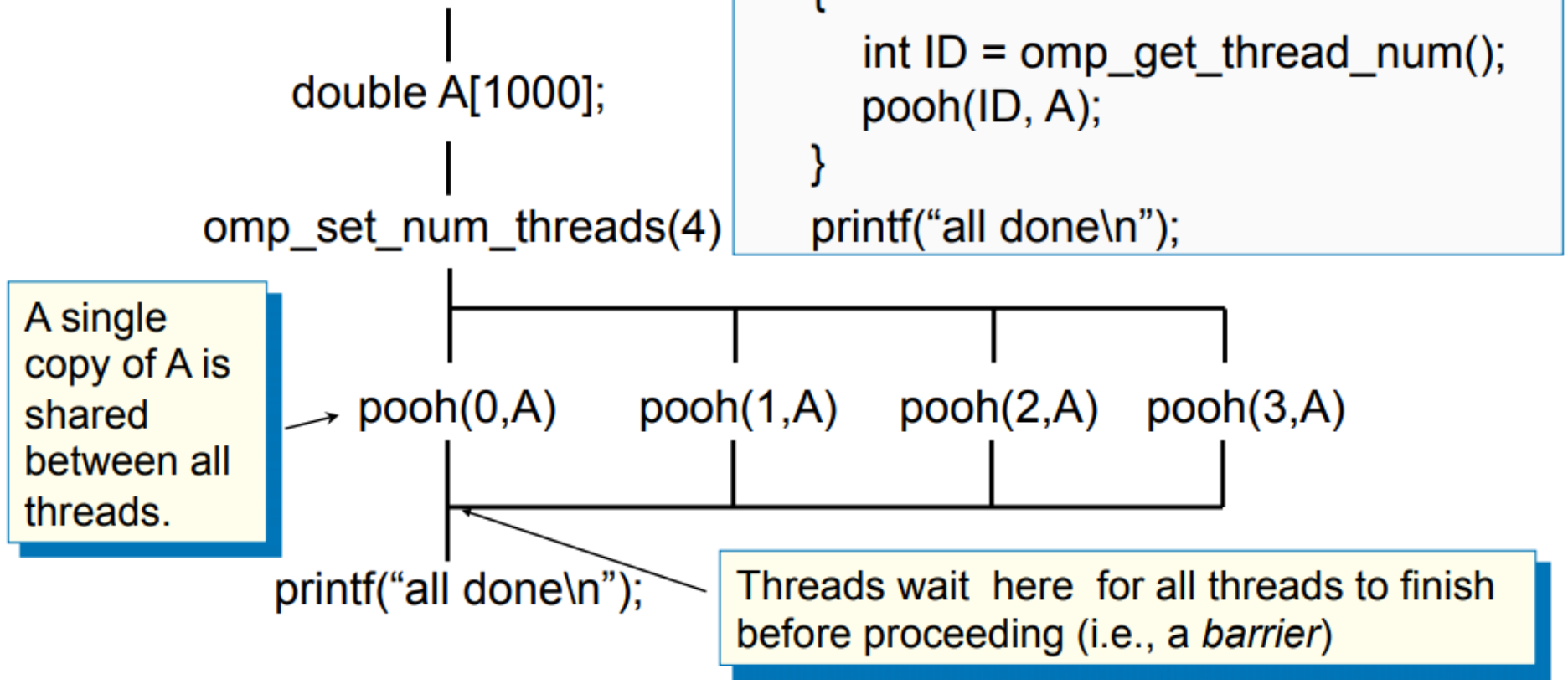


http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

OpenMP Basics

- Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

- Mutual exclusion: Only one thread at a time can enter a **critical** region.

Threads wait their turn – only one at a time calls consume()

```
float  res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
      for(i=id;i<niters;i+=nthrds){
           B =  big_job(i);
#pragma omp critical
           res += consume (B);
     }
}
```


http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

- The loop worksharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
}
```

The variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause

Loop construct name:

- C/C++: for
- Fortran: do

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

OpenMP Basics (Syntax)

- Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]…]

  – Example

  #pragma omp parallel num_threads(4)

- Function prototypes and types in the file:

  #include <omp.h>

  use omp_lib

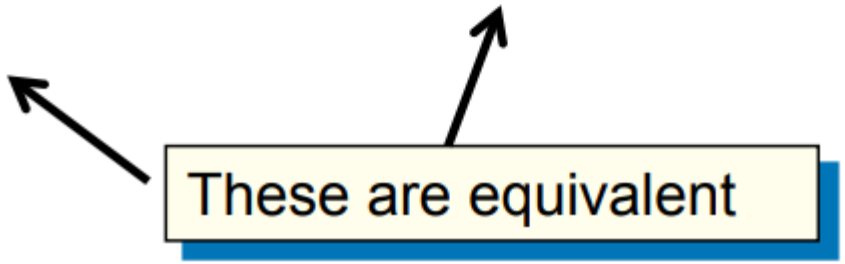
- Most OpenMP* constructs apply to a “structured block”.
  – Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  – It’s OK to have an exit() within the structured block.

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```
double res[MAX]; int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```

```
double res[MAX]; int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
```



http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead
AUTO	When the runtime can "learn" from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

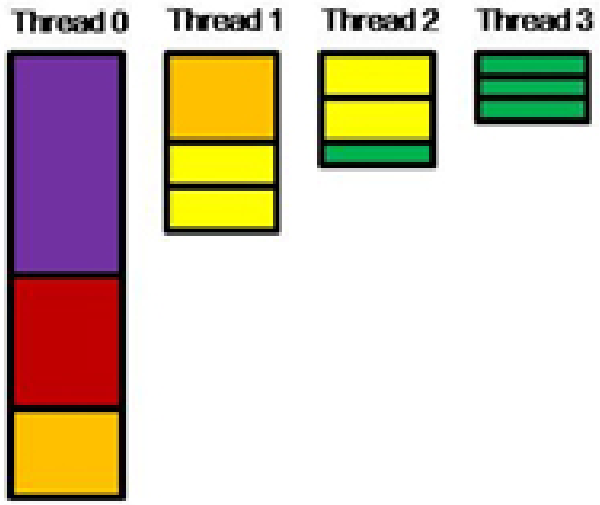
Most work at runtime : complex scheduling logic used at run-time

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

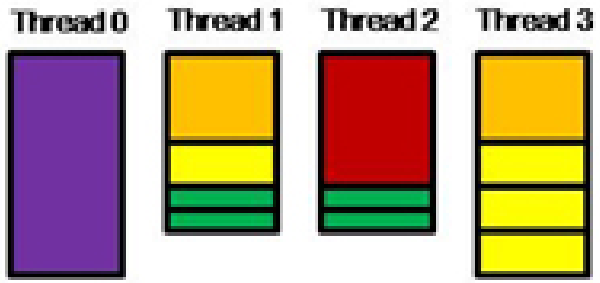
schedule(dynamic) can be your friend...

```
#pragma omp parallel for schedule(dynamic)
  for (i = 0; i < n; i++) {
    unknown_amount_of_work(i);
  }
```

You can use schedule(dynamic, <n>) to distribute in chunks of size n.



(a) Unbalanced assignment of tasks to threads



(b) Balanced assignment of tasks to threads

https://software.intel.com/en-us/articles/load-balance-and-parallel-performance

- **For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the collapse clause:**

```
#pragma omp parallel for collapse(2)
for (int i=0; i<N; i++) {
  for (int j=0; j<M; j++) {
          .....
  }
}
```

Number of loops to be parallelized, counting from the outside

- Will form a single loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult.

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

- One can selectively change storage attributes for constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE

All the clauses on this page apply to the OpenMP construct NOT to the entire region.

- The final value of a private variable inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - LASTPRIVATE
- The default attributes can be overridden with:
  - DEFAULT (PRIVATE | SHARED | NONE)

DEFAULT(PRIVATE) *is Fortran only*

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

- Variables initialized from a shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```

Each thread gets its own copy of incr with an initial value of 0

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

- How do we handle this case?

```
double  ave=0.0, A[MAX];    int i;
 for (i=0;i< MAX; i++) {
       ave + = A[i];
 }
 ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "reduction".
- Support for reduction operations is included in most parallel programming environments.

- OpenMP reduction clause:

  reduction (op : list)

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
       ave + = A[i];
 }
 ave = ave/MAX;
```


http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

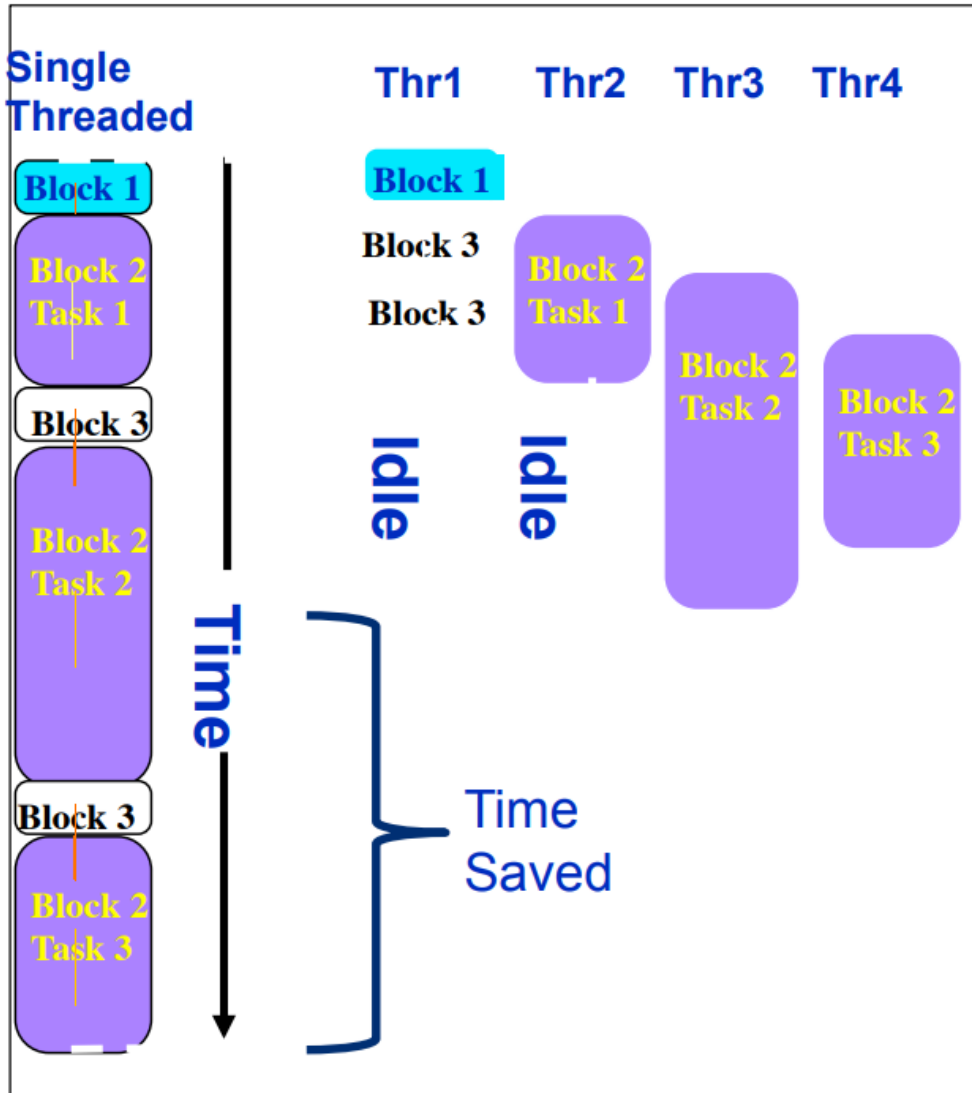
C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
     node * p = head;
     while (p) {  //block 2
     #pragma omp task
        process(p);

     }
  }
}
```



http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

OpenMP Basics

- Tasks are guaranteed to be complete at thread barriers:

  #pragma omp barrier

  – applies to all tasks generated in the current parallel region up to the barrier

- … or task barriers

  #pragma omp taskwait

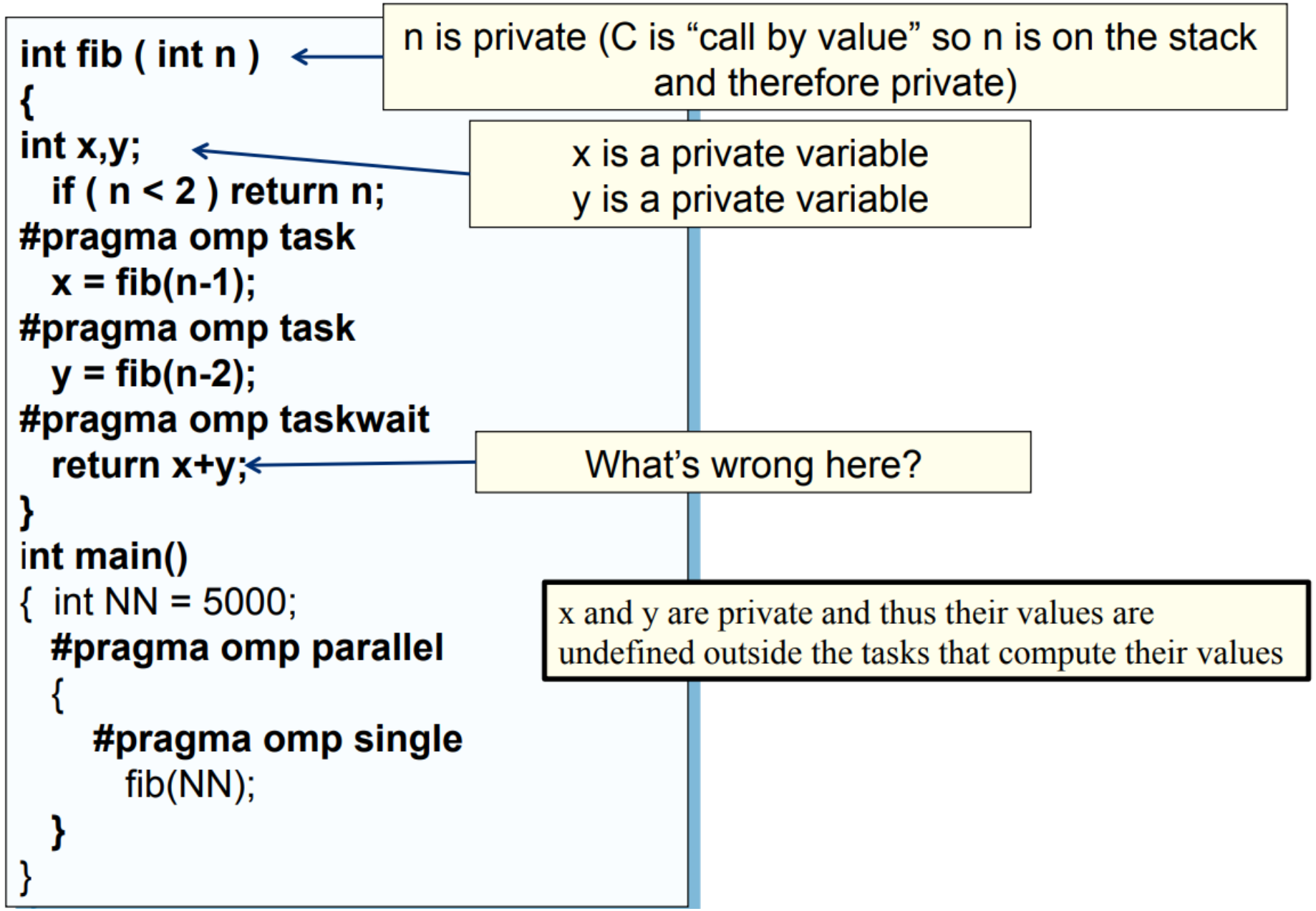
  – wait until all tasks generated in the current task have completed. Applies only to “sibling” tasks, not “descendants”

- … or at the end of a taskgroup region

  #pragma omp taskgroup

  – wait until all tasks created within the taskgroup have completed; Applies to “descendants” (and “siblings”)

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf



http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

```
int fib ( int n )
{
int x,y;
  if ( n < 2 ) return n;
#pragma omp task shared (x)
  x = fib(n-1);
#pragma omp task shared(y)
  y = fib(n-2);
#pragma omp taskwait
  return x+y
}
Int main()
{  int NN = 5000;
  #pragma omp parallel
  {
     #pragma omp single
       fib(NN);
  }
}
```

Solution: make x and y shared so they have well defined values that are still available after the tasks complete

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
      #pragma omp task
         process(e);
}
```

What's wrong here?

Possible data race!
Shared variable e
updated by multiple tasks

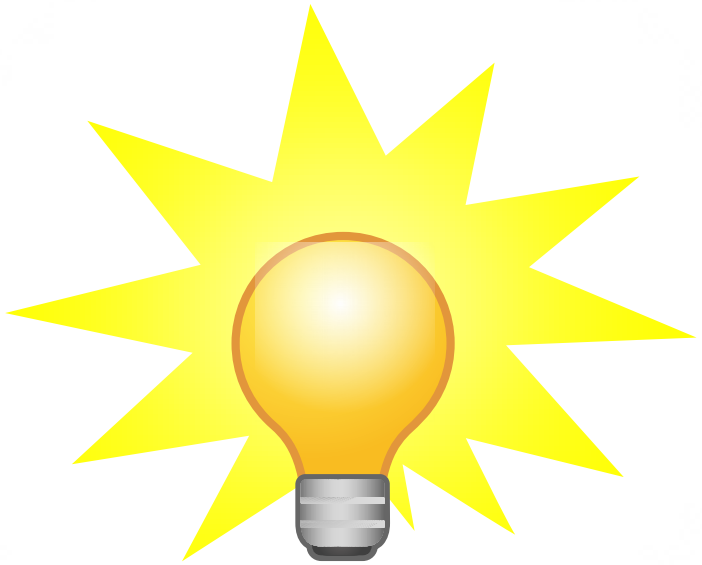
http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
      #pragma omp task firstprivate(e)
         process(e);
}
```

Solutions: Make “e” firstprivate so each task has its own, well-defined private copy of e

http://extremecomputingtraining.anl.gov/files/2015/08/Supinski_Mattson_omp_hands_on.pdf

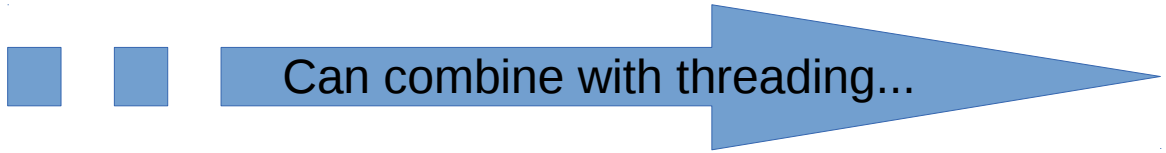
OpenMP and Heterogeneous Parallelism...



#pragma omp simd

Starting with OpenMP 4.0, OpenMP also supports explicit vectorization...

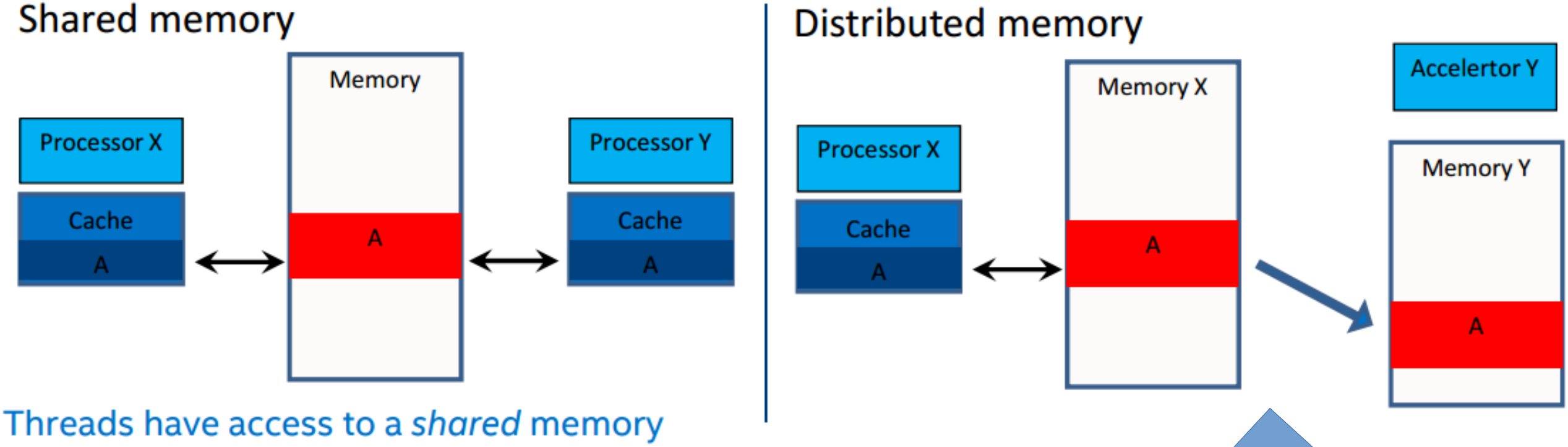
```
char foo(char *A, int n) {
  int i;
  char x = 0;
#pragma omp simd reduction(+:x)
  for (i=0; i<n; i++){
    x = x + A[i];
  }
  return x;
}
```




```
char foo(char *A, int n) {
  int i;
  char x = 0;
#pragma omp parallel for simd reduction(+:x)
  for (i=0; i<n; i++){
    x = x + A[i];
  }
  return x;
}
```

https://software.intel.com/en-us/articles/enabling-simd-in-program-using-openmp40

OpenMP Evolving Toward Accelerators



http://llvm-hpc2-workshop.github.io/slides/Tian.pdf

OpenMP Accelerator Support – An Example (SAXPY)

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y









  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

  free(x); free(y); return 0;
}
```

http://llvm-hpc2-workshop.github.io/slides/Wong.pdf

OpenMP Accelerator Support – An Example (SAXPY)

```
int main(int argc, const char* argv[]) {
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
```


```
#pragma omp distribute
  for (int i = 0; i < n; i += num_blocks){
```


```
#pragma omp parallel for
    for (int j = i; j < i + num_blocks; j++) {
```


```
      y[j] = a*x[j] + y[j];
  } }
} free(x); free(y); return 0; }
```

Memory transfer if necessary.

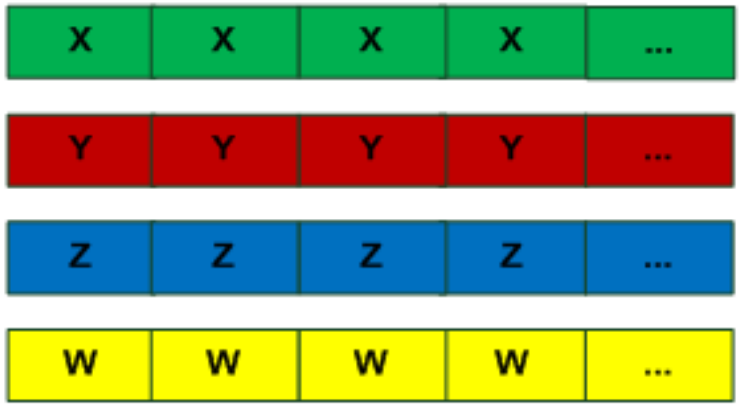
Traditional CPU-targeted OpenMP might only need this directive!

http://llvm-hpc2-workshop.github.io/slides/Wong.pdf

AOS vs. SOA

Easy to vectorize; uses lots of prefetching streams!

Structure of Arrays




```
struct Particles {
  float *x;
  float *y;
  float *z;
  float *w;
};
```

Array of Structures

https://software.intel.com/en-us/articles/ticker-tape-part-2

```
struct Particle {
  float x;
  float y;
  float z;
  float w;
};

struct Particle *Particles;
```

Better cache locality; fewer prefetcher streams
with scatter/gather support, maybe vectorization is not so bad!

Motivation

```
!! !$acc loop gang
!$omp teams distribute

!! !$acc loop gang worker
!$omp teams distribute parallel do

!! !$acc loop gang vector
!$omp teams distribute simd

!! !$acc loop gang worker vector
!$omp teams distribute parallel do simd

!! !$acc loop worker
!$omp parallel do

!! !$acc loop vector
!$omp simd

!! !$acc loop worker vector
!$omp parallel do simd
```

```
!$acc loop gang worker
do i=1,N
   !$acc loop independent
   do j=1,N
      ...
   end do
end do
```

- Initially proposed by NVidia, ORNL and LBL
- Education purpose
- Portable code with understand of non optimal performance

Compiler Backend Decision for Parallelization and Vectorization

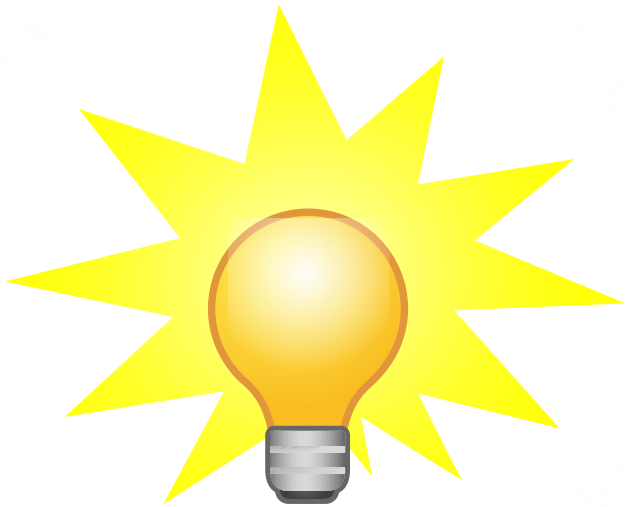
```
!$omp target teams distribute parallel do
 do i=1,N
   !$omp concurrent
   do j=1,N
      ...
   end do
end do
```

(Slide from Xinmin Tian (Intel))

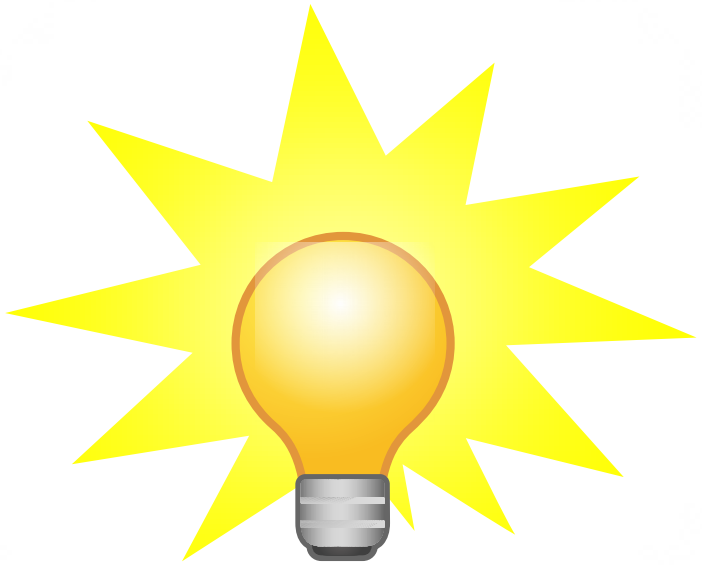
MKL, cuBLAS, ESSL, etc.

Vendors provide optimized math libraries for each system (BLAS for linear algebra, FFTs, and more).

- MKL on Intel systems, ESSL on IBM systems, cuBLAS (and others) for NVIDIA GPUs
- For FFTs, there is often an optional FFTW-compatible interface.



C++ Parallel-Algorithms Libraries...



C++ Parallel-Algorithms Libraries

We used to see only coarse-grained OpenMP, but this is changing…

- We're seeing even greater adoption of OpenMP, but…
- Many applications are not using OpenMP directly. Abstraction libraries are gaining in popularity.

Often uses OpenMP and/or other compiler directives under the hood.

Use of C++ Lambdas.

…TBB and Thrust.

- RAJA (https://github.com/LLNL/RAJA)

```
RAJA::ReduceSum<reduce_policy, double> piSum(0.0);

RAJA::forall<execute_policy>(begin, numBins, [=](int i) {
  double x = (double(i) + 0.5) / numBins;
  piSum += 4.0 / (1.0 + x * x);
});
```

- Kokkos (https://github.com/kokkos)

```
int sum = 0;
// The KOKKOS_LAMBDA macro replaces
// the capture-by-value clause [=].
// It also handles any other syntax
// needed for CUDA.
Kokkos::parallel_reduce (n, KOKKOS_LAMBDA (const int i,
                                           int& lsum) {
    lsum += i*i;
  }, sum);
```

C++ Parallel-Algorithms Libraries

And starting with C++17, the standard library has parallel algorithms too...

Table 2 — Table of parallel algorithms

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

[*Note:* Not all algorithms in the Standard Library have counterparts in Table 2. — *end note*]

```
// For example:
std::sort(std::execution::par_unseq, vec.begin(), vec.end()); // parallel and vectorized
```

(Compiler) Optimizations for OpenMP Code

OpenMP is already an abstraction layer. Why can't programmers just write the code optimally?

- Because what is optimal is different on different architectures.
- Because programmers use abstraction layers and may not be able to write the optimal code directly:

```
in library1:
void foo() {
  std::for_each(std::execution::par_unseq, vec1.begin(), vec1.end(), ...);
}

in library2:
void bar() {
  std::for_each(std::execution::par_unseq, vec2.begin(), vec2.end(), ...);
}

foo(); bar();
```

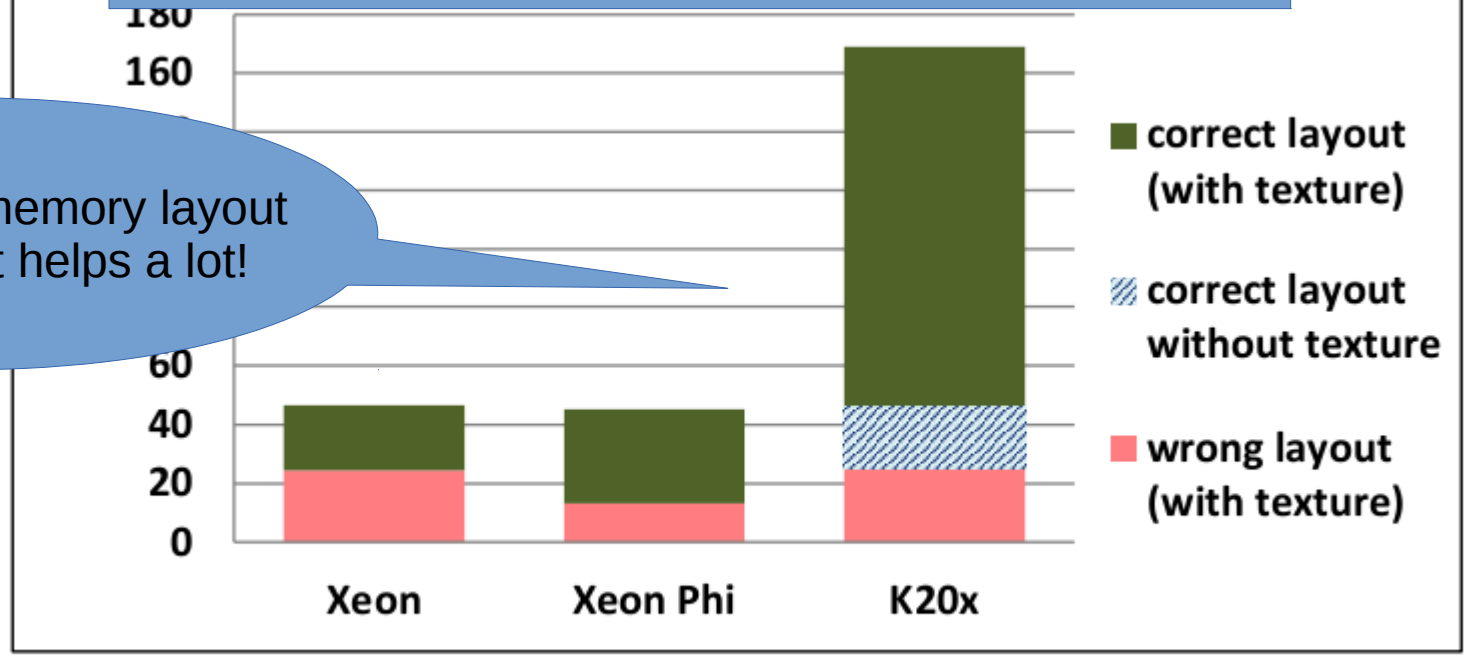
What About Memory?

It is really hard for compilers to change memory layouts and generally determine what memory is needed where. The Kokkos C++ library has memory placement and layout policies:

View<const double ***, Layout, Space , MemoryTraits<RandomAccess>> name (...);

Constant random-access data might be put into texture memory on a GPU, for example.

Using the right memory layout and placement helps a lot!



- **Large loss in performance with wrong layout**

https://trilinos.org/oldsite/events/trilinos_user_group_2013/presentations/2013-11-TUG-Kokkos-Tutorial.pdf

(Missing) Optimizations for Parallel Programs

Or, “Why parallel loops might slow down your code”

(Featuring work by Johannes Doerfert, see our IWOMP 2018 paper)

Problem 1: variable capturing

Input program:

```
int y = 1337;

#pragma omp parallel for

for (int i = 0; i < N; i++)
  g(y, i);
g(y, y);
```

Optimal program:

```
#pragma omp parallel for

for (int i = 0; i < N; i++)
  g(1337, i);
g(1337, 1337);
```

Clang output:

```
int y = 1337;
call fork_parallel(fn, &y);
g(y, y);
```

GCC output:

```
int y = 1337;
call fork_parallel(fn, &y);
g(1337, 1337);
```

Solution 1: variable privatization

Input program:

```
int y = 1337;

#pragma omp parallel for

for (int i = 0; i < N; i++)
  g(y, i);
g(y, y);
```

Optimized program:

```
int y = 1337; y_p = y;

#pragma omp parallel for

for (int i = 0; i < N; i++)
  g(y_p, i);
g(1337, 1337);
```

Clang output:

```
int y = 1337;
call fork_parallel(fn, &y);
g(y, y);
```

Clang output:

```
int y_p = 1337;
call fork_parallel(fn, &y_p);
g(1337, 1337);
```

Problem 2: (implicit) barriers

```
void copy(float* dst, float* src, int N) {
 #pragma omp parallel for
  for(int i = 0; i < N; i++)
    dst[i] = src[i];
}

void compute_step_factor(int nelr, float* vars,
                         float* areas, float* sf) {
#pragma omp parallel for
  for (int blk = 0; blk < nelr / block_length; ++blk) {
      ...
```

Problem 2: (implicit) barriers (con’t)

```
for (int i = 0; i < iterations; i++) {
  copy(old_vars, vars, nelr * NVAR);
  compute_step_factor(nelr, vars, areas, sf);
  for (int j = 0; j < RK; j++) {
    compute_flux(nelr, ese, normals, vars, fluxes, ff_vars,
                 ff_m_x, ff_m_y, ff_m_z, ff_dnergy);
    time_step(j, nelr, old_vars, vars, sf, fluxes);
  }
}
```

Problem 2: (implicit) barriers (con’t)

```
for (int i = 0; i < iterations; i++) {
  #pragma omp parallel for              // copy
  for (...) { /* write old_vars, read vars */ }
  #pragma omp parallel for              // compute_step_factor
  for (...) { /* write sf, read vars & area */ }
  for (int j = 0; j < RK; j++) {
    #pragma omp parallel for            // compute_flux
    for (...) { /* write fluxes, read vars & ... */ }
    ...
```

Solution 2: region expansion & barrier elimination

```
#pragma omp parallel
for (int i = 0; i < iterations; i++) {
  #pragma omp for nowait          // copy
  for (...) { /* write old_vars, read vars */ }
  #pragma omp for   nowait        // compute_step_factor
  for (...) { /* write sf, read vars & area */ }
  for (int j = 0; j < RK; j++) {
    #pragma omp for               // compute_flux
    for (...) { /* write fluxes, read vars & ... */ }
  ...
```

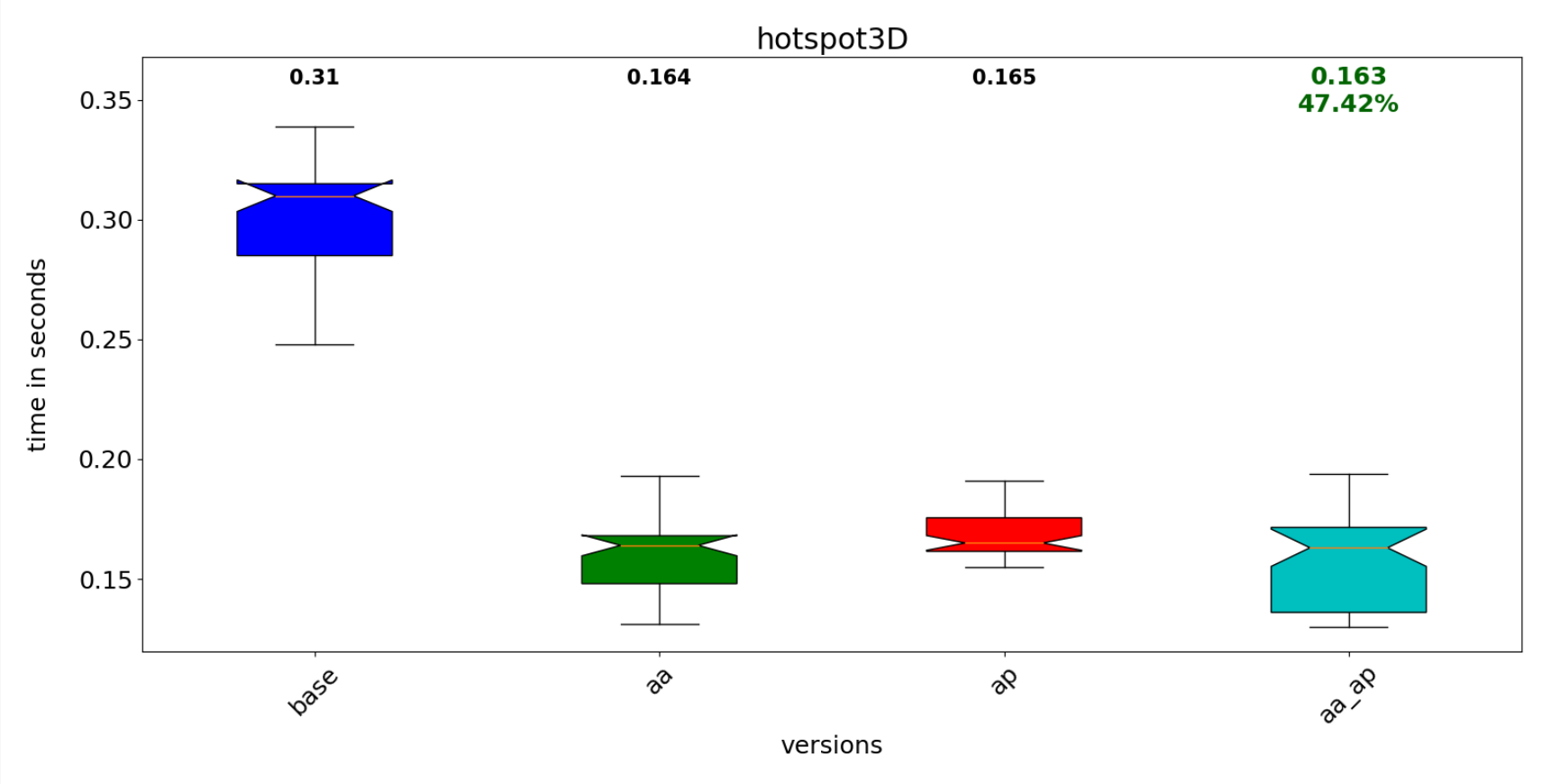
Example 1: Rodinia - hotspot3D

```
#pragma omp parallel
{
  int count = 0;
  float *tIn = In, *tOut = Out;
#pragma omp master
  printf("%d threads running \n", omp_get_num_threads ());
  do {
    int z;
#pragma omp for
    for (z = 0; z < nz; z++) {
      int y;
      for (y = 0; y < ny; y++) {
        int x;
        for (x = 0; x < nx; x++) {
          int c, w, e, n, s, b, t;
          c = x + y * nx + z * nx * ny;
          w = (x == 0) ? c : c - 1;
          e = (x == nx - 1) ? c : c + 1;
          n = (y == 0) ? c : c - nx;
          s = (y == ny - 1) ? c : c + nx;
          b = (z == 0) ? c : c - nx * ny;
          t = (z == nz - 1) ? c : c + nx * ny;
          tOut[c] = cc * tIn[c] + cw * tIn[w] + ce * tIn[e] +
                    cs * tIn[s] + cn * tIn[n] + cb * tIn[b] +
                    ct * tIn[t] + (dt/Cap) * pIn[c] + ct * a;
        }
      }
    }
    float *t = tIn, tIn = tOut;
    tOut = t;
  } while (++count < numiter);
}
```

Example 1:

Rodinia - hotspot3D

./3D 512 8 100 ../data/hotspot3D/power_512x8 ../data/hotspot3D/temp_512x8



Intel core i9, 10 cores, 20 threads, 51 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization

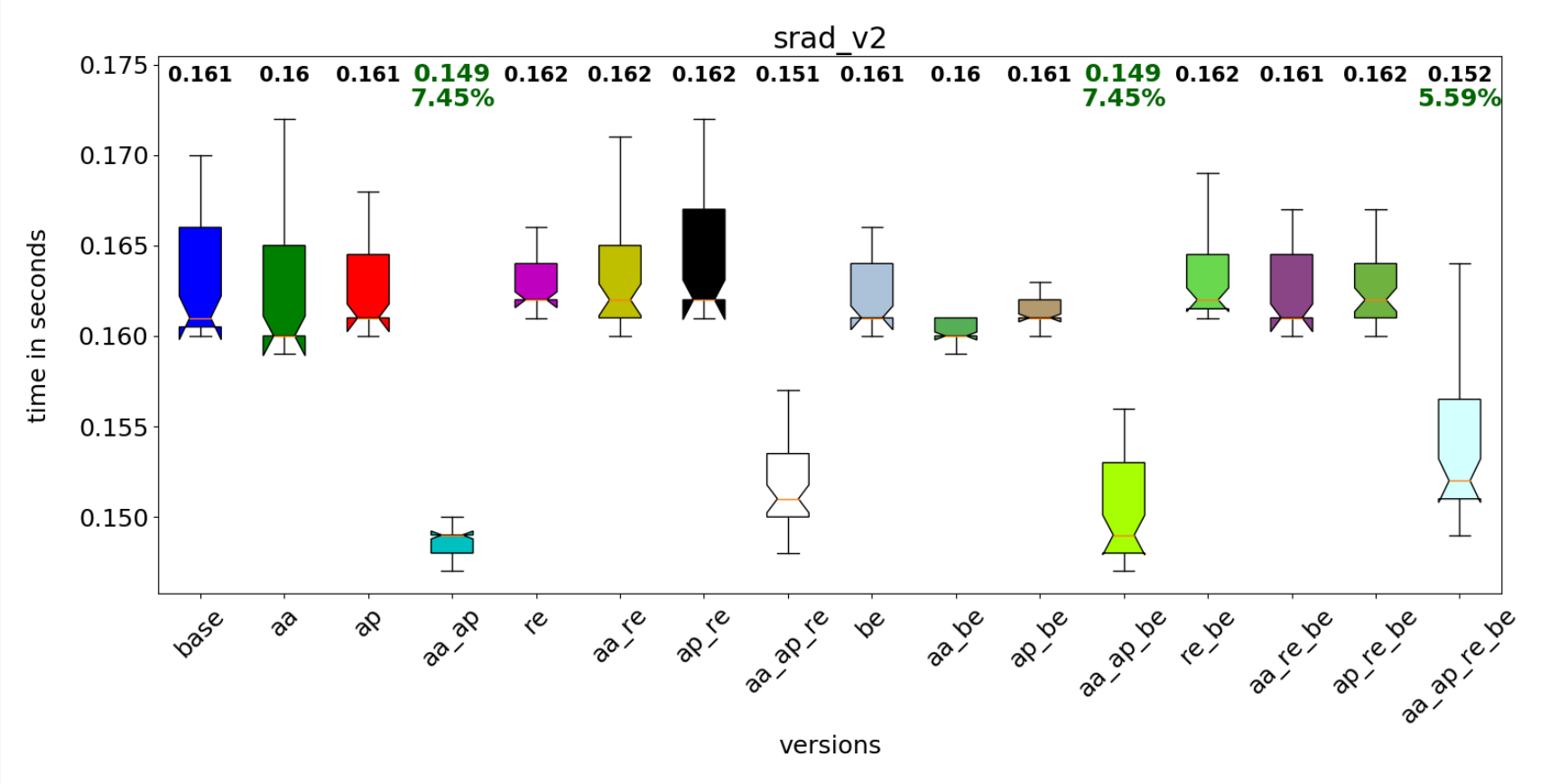
Example 2: Rodinia - srad_v2

```
#pragma omp parallel for shared(J, dN, dS, dW, dE, c, rows,    \
  cols, iN, iS, jW, jE) private(j, k, Jc, G2, L, num, den, qsqr)
for (int i = 0; i < rows; i++) {
  ...
}
#pragma omp parallel for shared(J, c, rows, cols, lambda)
                         private(i, j, k, D, cS, cN, cW, cE)
for (int i = 0; i < rows; i++) {
  ...
}
```

Example 2:

Rodinia - srad_v2

./srad 2048 2048 0 127 0 127 20 0.5 20



Intel core i9, 10 cores, 20 threads, 51 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 3: Rodinia - cfd

```
for (int i = 0; i < iterations; i++) {
  copy(old_vars, vars, nelr * NVAR);
  compute_step_factor(nelr, vars, areas, sf);
  for (int j = 0; j < RK; j++) {
    compute_flux(nelr, ese, normals, vars, fluxes, ff_vars,
                 ff_m_x, ff_m_y, ff_m_z, ff_dnergy);
    time_step(j, nelr, old_vars, vars, sf, fluxes);
  }
}
```

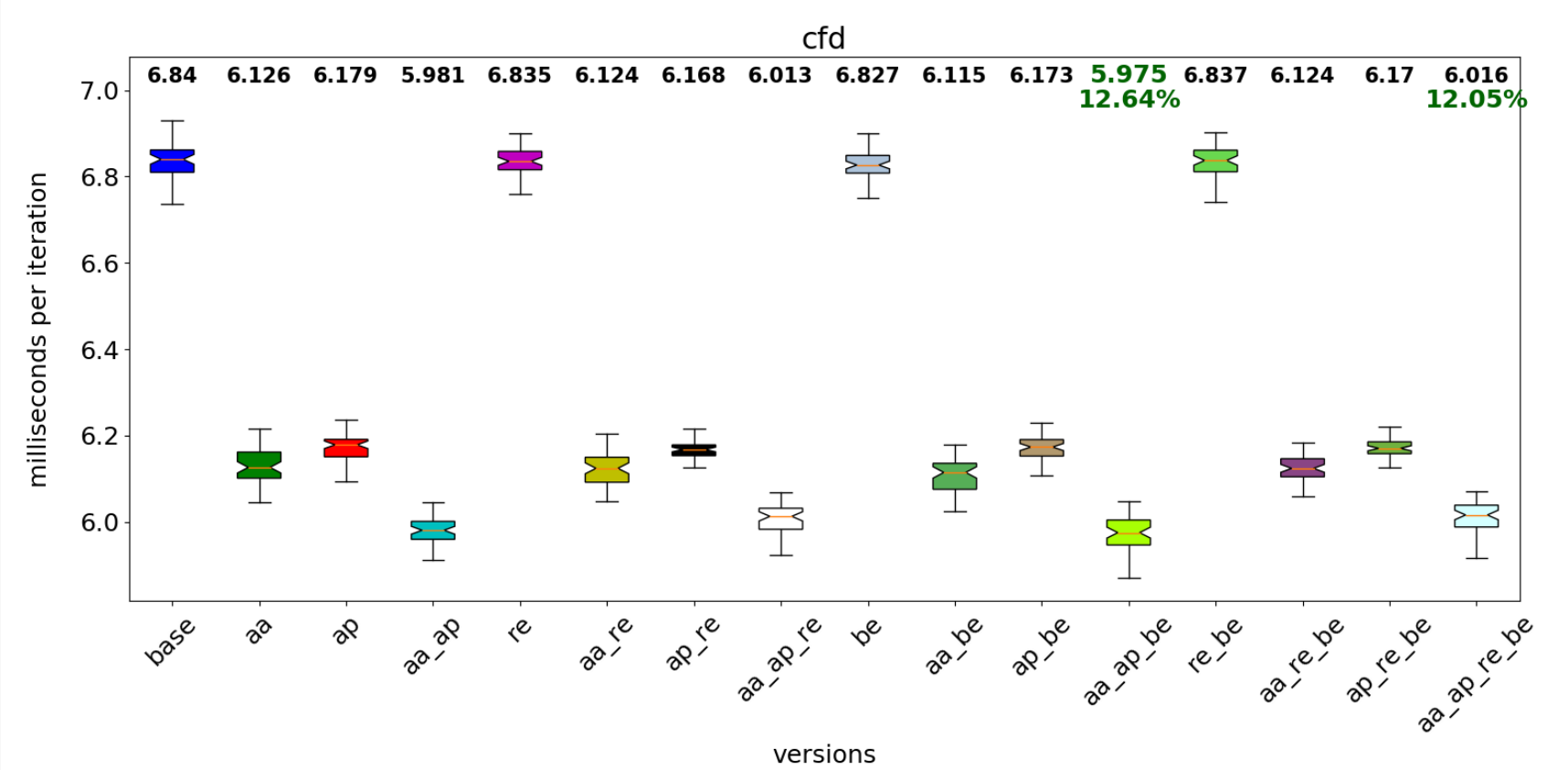
Example 3: Rodinia - cfd

```
#pragma omp parallel
for (int i = 0; i < iterations; i++) {
  #pragma omp for nowait                    // copy
  for (...) { /* write old_vars, read vars */ }
  #pragma omp for nowait                    // compute_step_factor
  for (...) { /* write sf, read vars & area */ }
  for (int j = 0; j < RK; j++) {
    #pragma omp for                         // compute_flux
    for (...) { /* write fluxes, read vars & ... */ }
  ...
```

Example 3:

Rodinia - cfd

cfd fvcorr.domn.193K



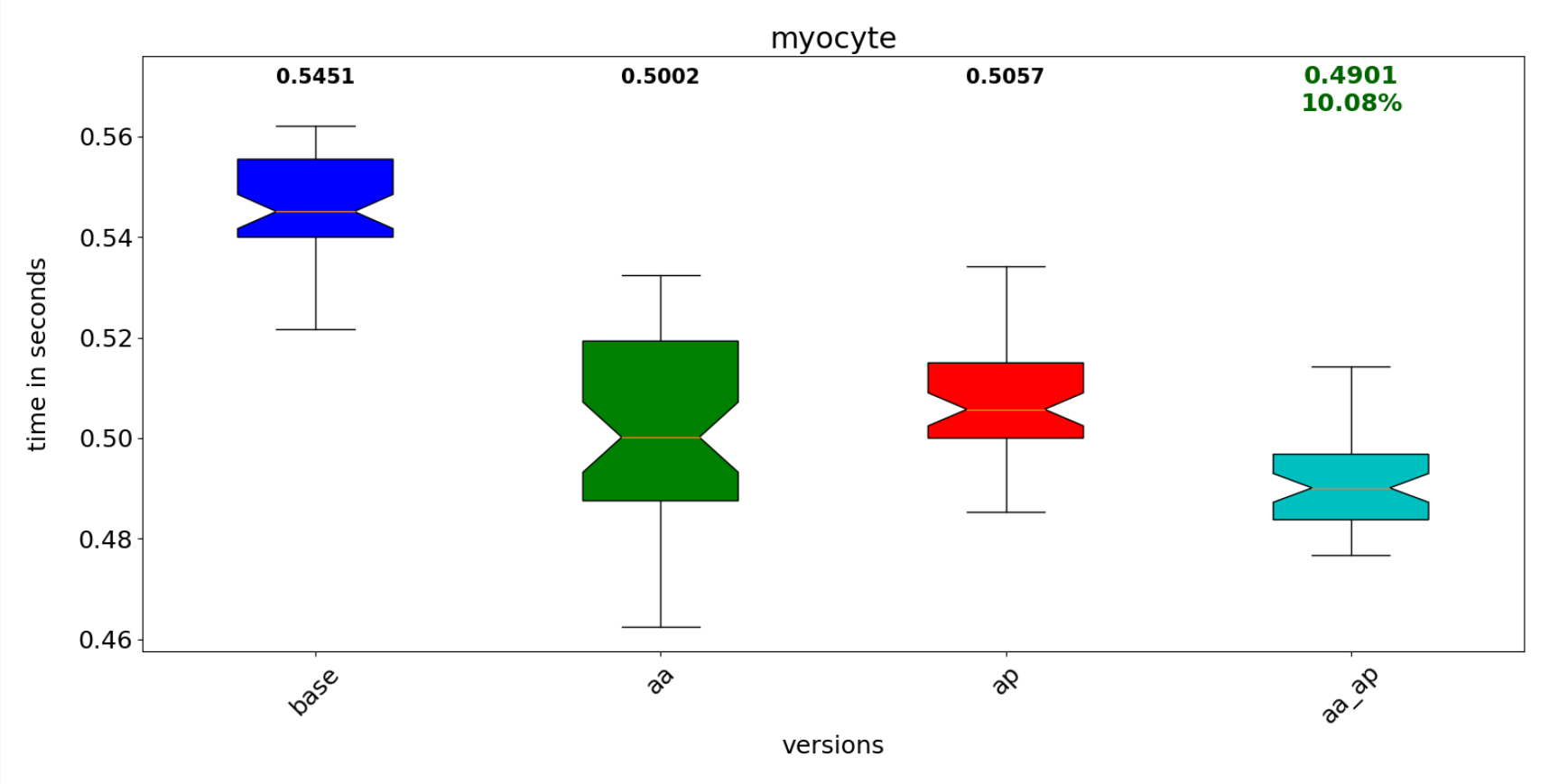
Intel core i9, 10 cores, 20 threads, 51 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 4:

Rodinia - myocyte

./myocyte 100 100 0 8



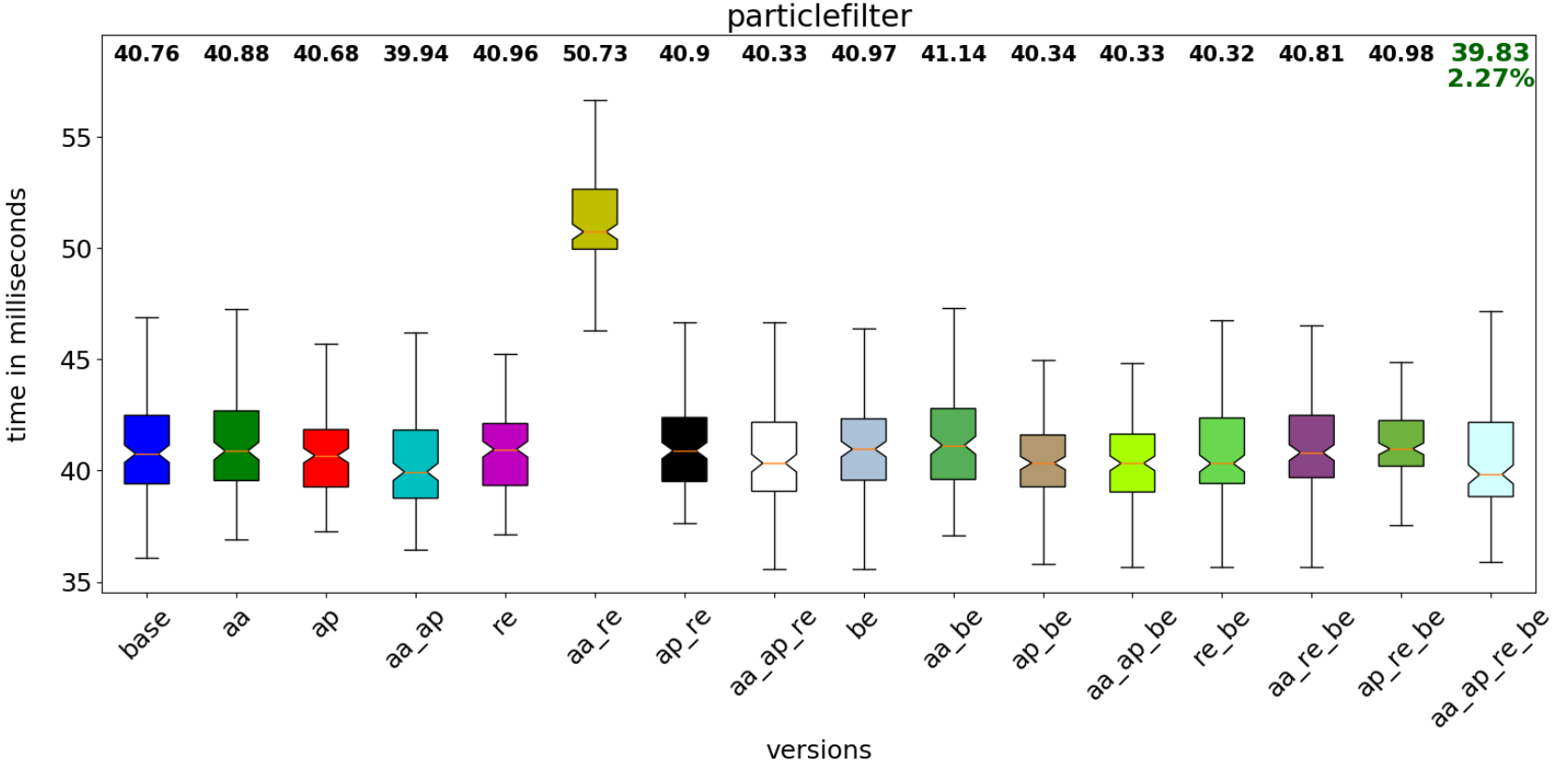
Intel core i9, 10 cores, 20 threads, 51 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 5:

Rodinia - particlefilter

./particlefilter -x 128 -y 128 -z 10 -np 10000

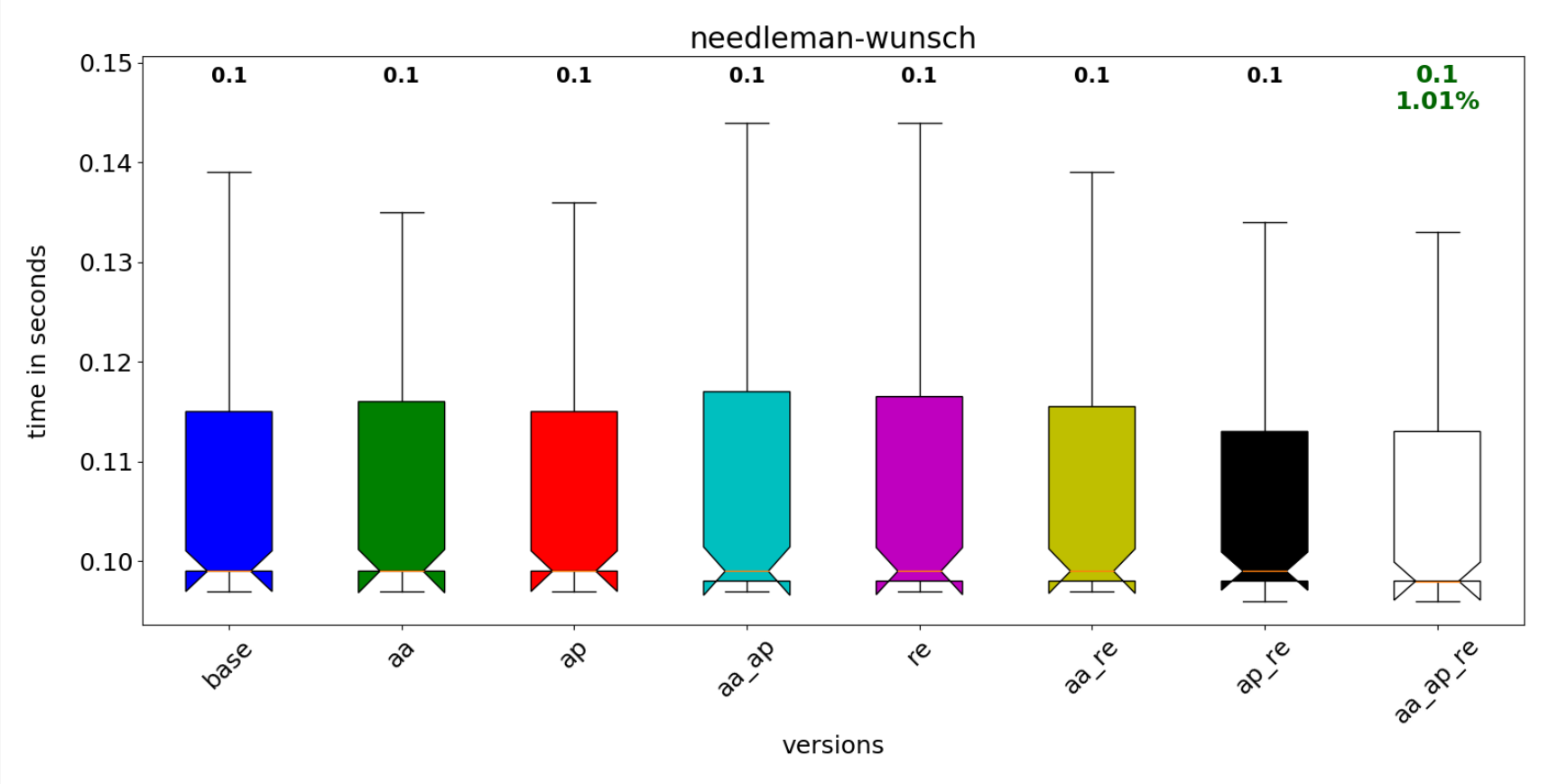


Intel core i9, 10 cores, 20 threads, 151 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 6: Rodinia - needleman-wunsch

./nw 8192 10 8



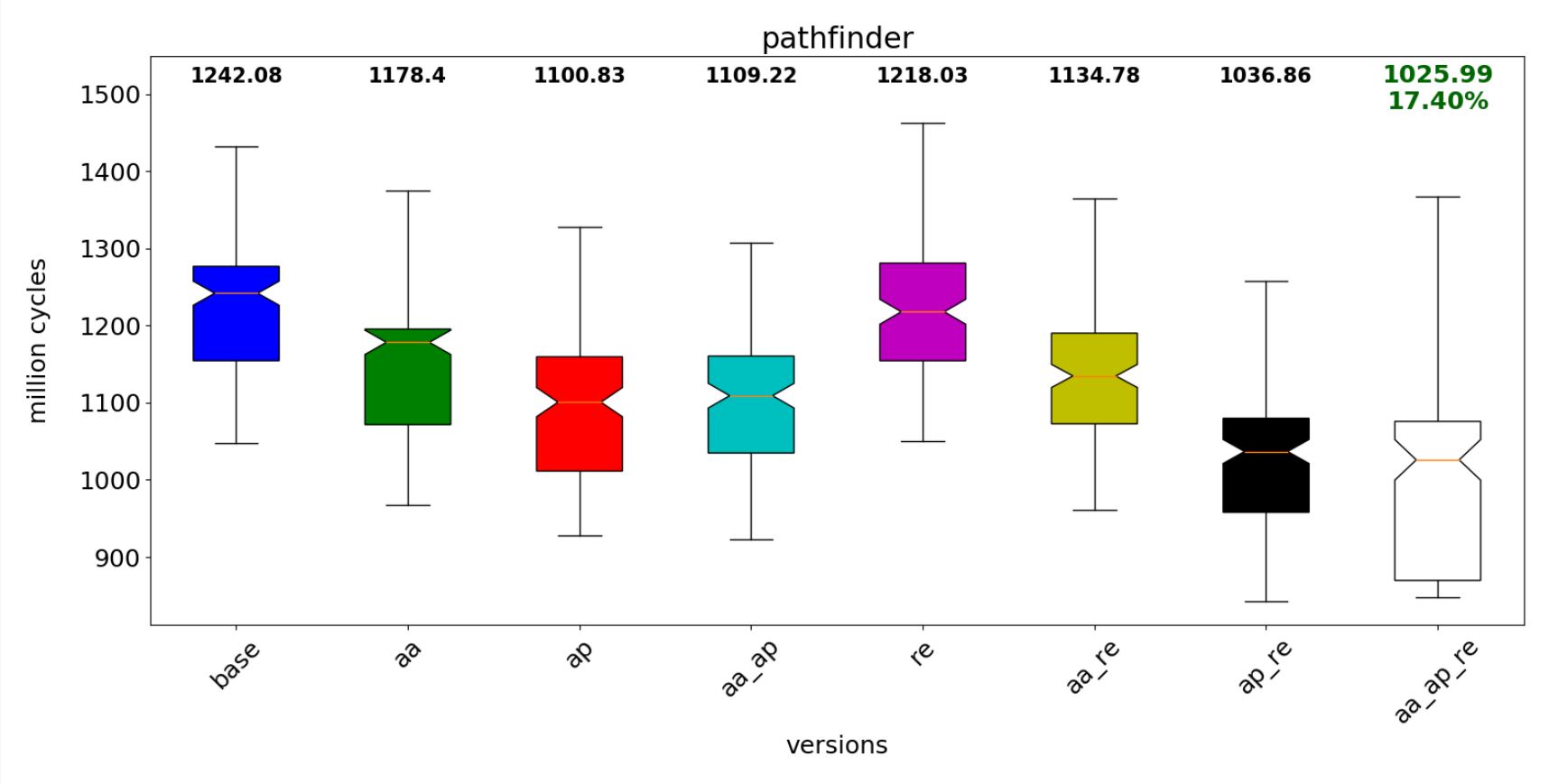
Intel core i9, 10 cores, 20 threads, 151 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

Example 7:

Rodinia - pathfinder

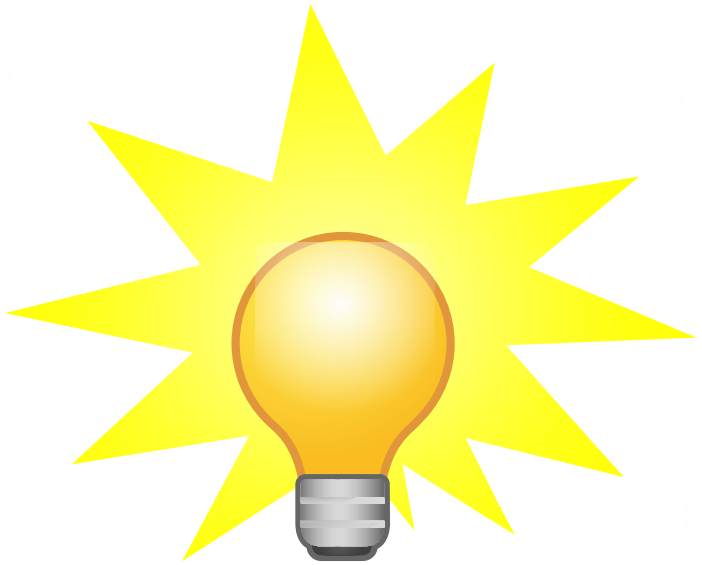
./pathfinder 40000 40000



Intel core i9, 10 cores, 20 threads, 151 runs, with and without

- aa => alias attribute propagation
- ap => argument privatization
- re => region expansion
- be => barrier elimination

What LANL (+MIT, et al.) Has Been Working On...

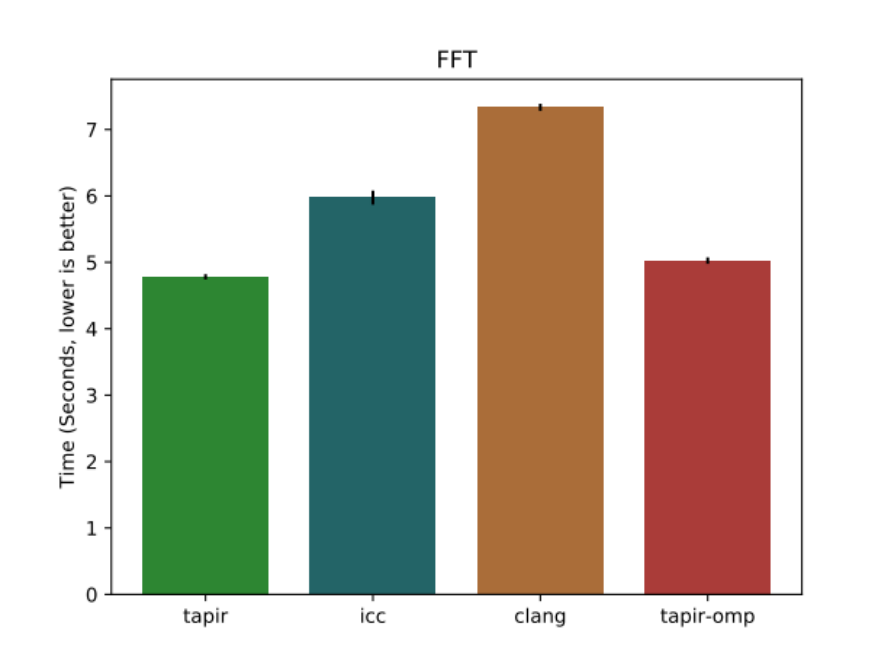


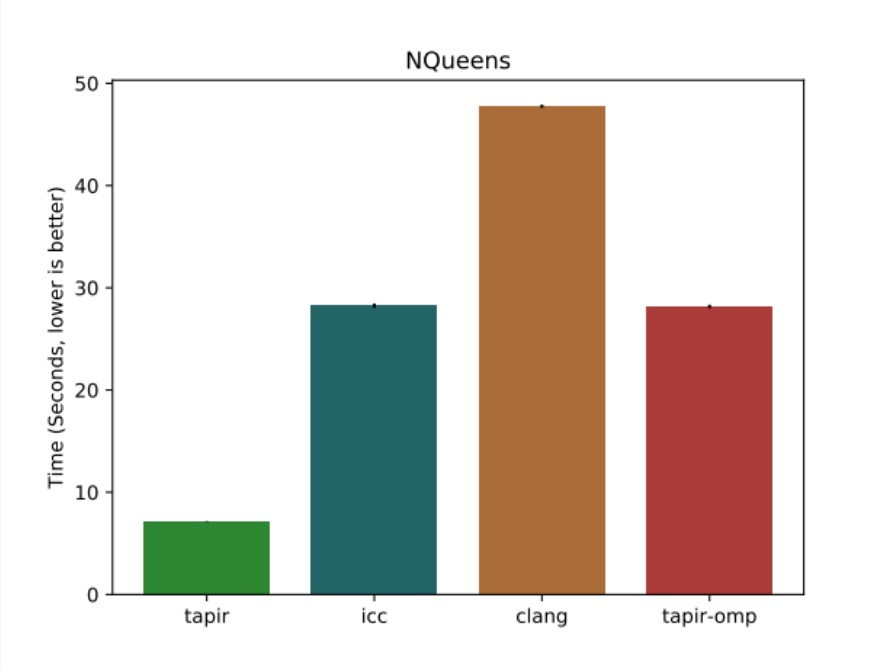
OpenMP Tasks → Tapir

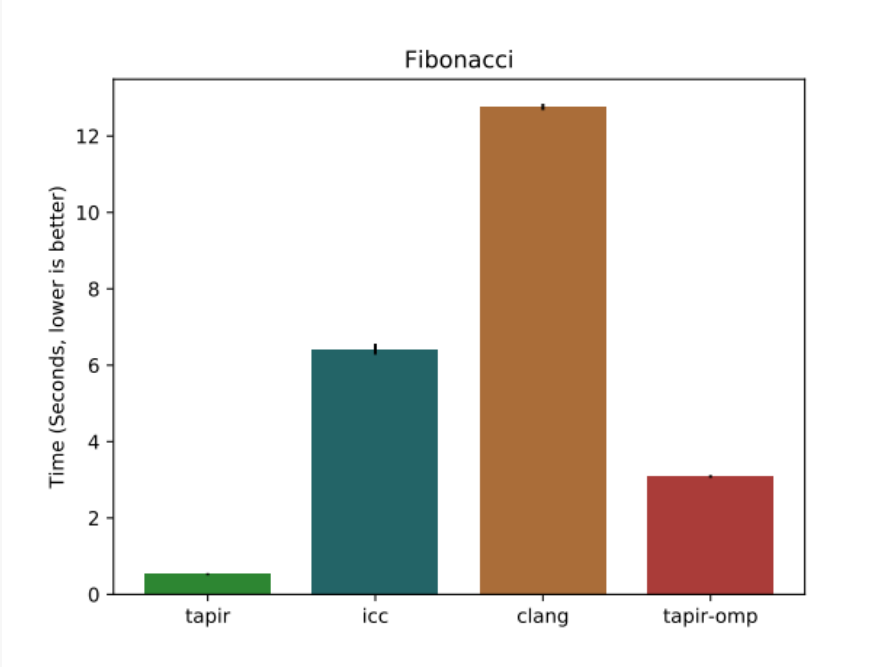
```
int fib(int n){

...
    #pragma omp task
      x = fib(n-1);
    #pragma omp task
      y = fib(n-2);
    #pragma omp taskwait
...
}
```

```
...
if.end:
  detach label %det.achd, label %det.cont

det.achd:
  %2 = load i32, i32* %n.addr, align 4
  %sub = sub nsw i32 %2, 1
  %call = call i32 @fib(i32 %sub)
  store i32 %call, i32* %x, align 4
  reattach label %det.cont

det.cont:
  detach label %det.achd1, label %det.cont4

det.achd1:
  %3 = load i32, i32* %n.addr, align 4
  %sub2 = sub nsw i32 %3, 2
  %call3 = call i32 @fib(i32 %sub2)
  store i32 %call3, i32* %y, align 4
  reattach label %det.cont4

det.cont4:
  sync label %sync.continue
...
```

OpenMP Task Results







Some final advice...

Don't guess! Profile! Your performance bottlenecks might be very different on different systems.

And don't be afraid to ask questions...

Any questions?

➔ ALCF is supported by DOE/SC under contract DE-AC02-06CH11357

Acknowledgments

- The LLVM community (including our many contributing vendors)
- ALCF, ANL, and DOE
- ALCF is supported by DOE/SC under contract DE-AC02-06CH11357

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.