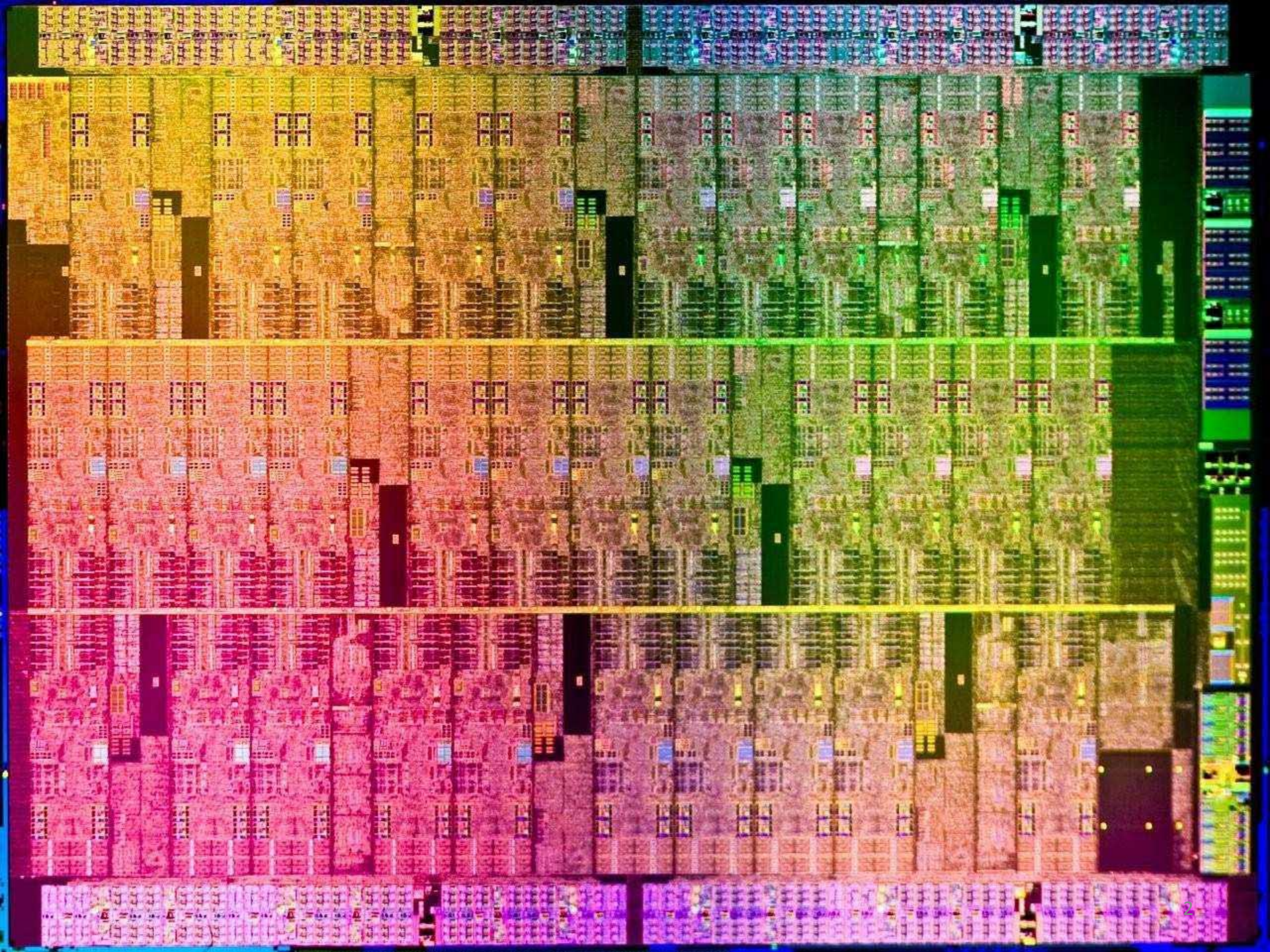
Intro to MIC performance

Vasily Volkov

March 6, 2012



Agenda

- Three programming models for MIC
- Hitting best throughput numbers
- OpenMP thread affinity settings
- Reproducing high performance in MKL
- Performance of cache coherence mechanism
- Beating Intel OpenMP barrier implementation

Part I

Basics of MIC programming

Starting up

I. SSH to MIC systems at:

- **knight.millennium.berkeley.edu**
  - 32 cores @ 1.2 GHz
  - Used throughout this presentation
- **ferry.millennium.berkeley.edu**
  - 30 active cores @ 1.05 GHz

II. Set up compiler environment:

source /opt/intel/composerxe_mic/bin/compilervars.sh intel64

Three ways to program MIC

Native mode

- SSH to MIC and use it as any multiprocessor machine

(Heterogeneous) explicit copy model

- Mark up the code that runs on MIC using #pragma
- Resembles OpenMP and CUDA
- Specify which arrays to transfer to MIC and back

(Heterogeneous) implicit copy model

- Mark up calls to MIC using keywords
- All transfers to MIC are handled automatically

Computing π in MIC native mode

```
#include <stdio.h>
main()
{
    double sum = 0, h = 1./1024/1024;
    for( double x = h/2; x < 1; x += h )
        sum += 4/(1+x*x);
    printf( "pi = %.15f\n", sum*h );//15 accurate digits
}
```

At host command line:

icc -mmic pi.cpp

sudo scp -i /opt/intel/mic/id_rsa a.out root@192.168.1.100:

sudo ssh -i /opt/intel/mic/id_rsa root@192.168.1.100

At MIC command line: **./a.out**

Ask support@millennium for sudo access

Computing π in MIC offload mode

```
#include <stdio.h>
main()
{
    double sum = 0, h = 1./1024/1024;
    #pragma offload target(mic)
    {
        for( double x = h/2; x < 1; x += h )
            sum += 4/(1+x*x);
    }
    printf( "pi = %.15f\n", sum*h );
}
```

Host command line:

icc -offload-build pi.cpp

./a.out

Note: if MIC is unavailable, everything runs on host

Computing π in MIC Cilk+ mode

```
#include <stdio.h>
_Cilk_shared double compute_pi( double h )
{
  double sum = 0;
  for( double x = h/2; x < 1; x += h ) sum += 4/(1+x*x);
  return sum*h;
}
main()
{
    double pi = _Cilk_offload compute_pi( 1./(1<<20) );
    printf( "pi = %.15f\n", pi );
}
```

At host command line:

icc -offload-build pi.cpp

./a.out

Explicit copy model

```
#include <stdio.h>
__declspec(target(mic)) double compute( double *x, int n )
{
    double sum = 0;
    for( int i = 0; i < n; i++ ) sum += 4/(1+x[i]*x[i]);
    return sum/n;
}
main()
{
    int n = 1<<20;
    double *x = (double *)malloc( n*sizeof(double) ), pi;
    for( int i = 0; i < n; i++ ) x[i] = (i+0.5)/n;
    #pragma offload target(mic) in(x:length(n))
        pi = compute( x, n );
    printf( "pi = %.15f\n", pi );
}
```

To transfer an array to MIC use in(…)

Other options: out(…), inout(…), nocopy(…)

Implicit copy model

```
#include <stdio.h>
_Cilk_shared double compute( double _Cilk_shared *x, int n )
{
    double sum = 0;
    for( int i = 0; i < n; i++ ) sum += 4/(1+x[i]*x[i]);
    return sum/n;
}
main()
{
    int n = 1<<20;
    double _Cilk_shared *x = (double _Cilk_shared*)
                _Offload_shared_malloc( n*sizeof(double) );
    for( int i = 0; i < n; i++ ) x[i] = (i+0.5)/n;
    double pi = _Cilk_offload compute( x, n );
    printf( "pi = %.15f\n", pi );
}
```

Automatic transfers if using **_Offload_shared_malloc** and **_Cilk_offload**

Recommended Reading

https://mic-dev.intel.com

- Need Intel contact to get access
- Of interest: architecture specs, forum

/opt/intel/composerxe_mic/Documentation/en_US/mic/mic_compiler_guide.pdf

- Programming models, language extensions

/opt/intel/composerxe_mic/Documentation/en_US/Release_Notes_mic.pdf

- Section 4.4.21: Using large pages in DMA buffers
- Section 4.4.22: Using large pages in malloc

A few more system tools

System info on MIC: **sudo micinfo**

Software stack is still not very stable…

- To reboot MIC only: **sudo micstart -r**
- To reboot entire system: **sudo reboot**

Part II

Measuring offload overhead

Offload overhead in pragma model

```
#include <stdio.h>
#include <mkl.h>
main( )
{
    for( int i = 0; i < 100; )
    {
        double t = dsecnd(); //accurate timer from MKL
        #pragma offload target(mic)
        {
            i++;
        }
        t = dsecnd() - t;
        printf( "%g ms\n", t*1e3 );
    }
}
```

Compile this **and all following codes** using:

icc -offload-build -O3 -mkl bench.cpp

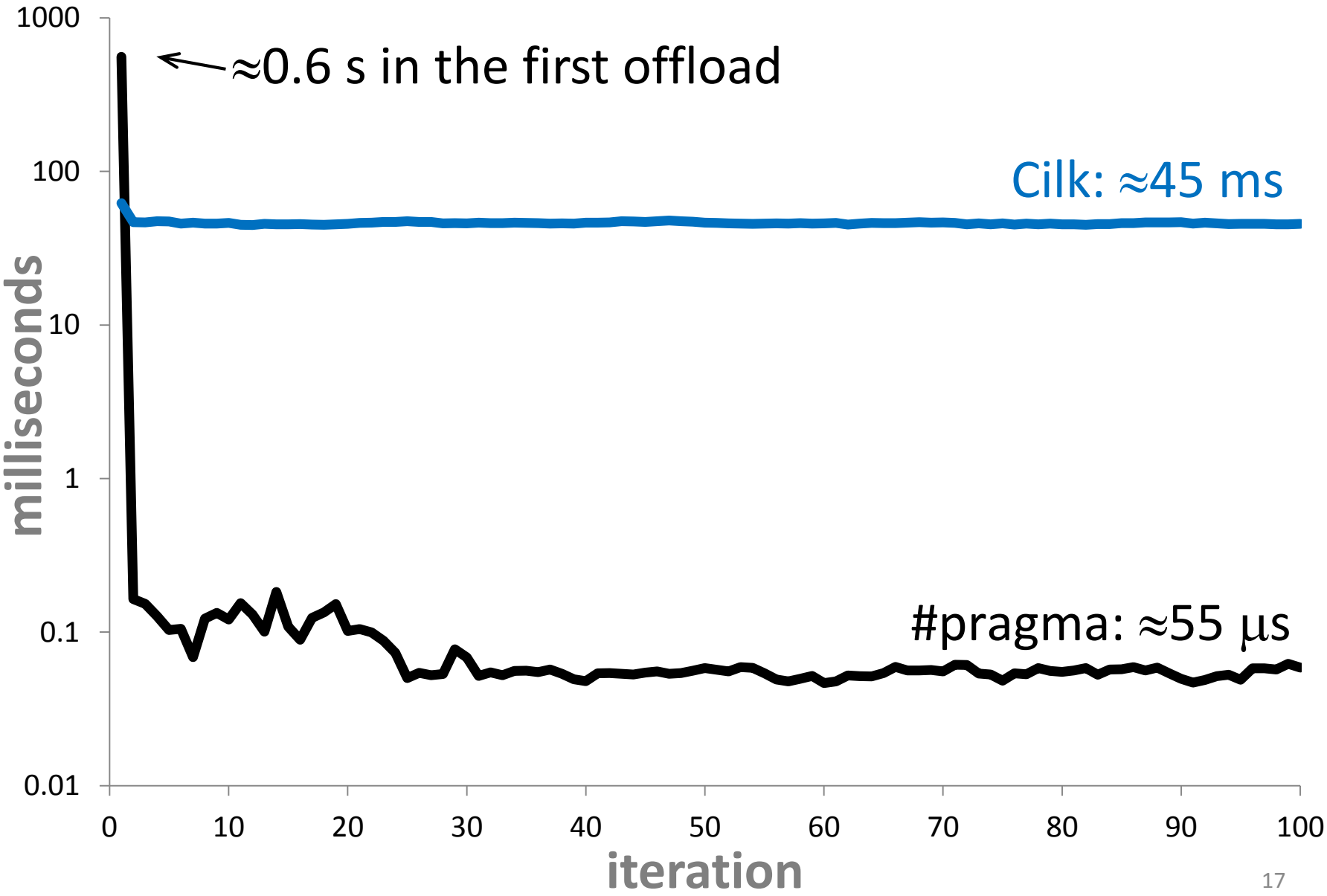
Offload overhead in Cilk+ model

```
#include <stdio.h>
#include <mkl.h>

_Cilk_shared int foo( int i ) { return i+1; }

main( )
{
    for( int i = 0; i < 100; )
    {
        double t = dsecnd();
        i = _Cilk_offload foo( i );
        t = dsecnd() - t;
        printf( "%g ms\n", t*1e3 );
    }
}
```

Compile this **and all following codes** using:

icc -offload-build -O3 -mkl bench.cpp

Offload overhead



Large overheads in Cilk+ are going to stay

Kevin Davis (Intel):

> Development does not consider the _Cilk_shared behavior [40+ ms overhead] to be a defect…
>
> **https://mic-dev.intel.com/node/1508#comment-3161**

Part III

Arithmetic peak

Getting the arithmetic peak (1/2)

```
#ifdef __MIC__ //can't use intrinsics if not on MIC
#include <micvec.h>
__declspec(target(mic)) void foo( float &r, int niterations )
{
    F32vec16 a( 100.f ), c( 1.f ), x( 0.9f );
    for( int i = 0; i < niterations; i += 8 )
    {
        a = a*x + c; a = a*x + c; a = a*x + c; a = a*x + c;
        a = a*x + c; a = a*x + c; a = a*x + c; a = a*x + c;
    }
    r = reduce_add( a );
}
#else
__declspec(target(mic)) void foo( float &r, int niterations ){}
#endif
```

Getting the arithmetic peak (2/2)

```
#include <stdio.h>
#include <mkl.h>
main( )
{
    for( int i = 0; i < 20; i++ )
    {
        int niterations = (100+i)<<20;
        float t, r;
        #pragma offload target(mic)
        {
            double t0 = dsecnd();
            foo( r, niterations );
            t = dsecnd() - t0;
        }
        printf( "%g Gflop/s, %g\n", 32e-9*niterations/t, r );
    }
}
```

Arithmetic throughput results

Result: 9.5 Gflop/s = **25% of 1-core peak**

Why: MAD back-to-back latency is 4 cycles

- **Need 4 MADs in the flight to get the peak**

Issues from same thread only every other cycle

- **Need at least 2 threads to get the peak**

NOPs?!

Disassemble: **icc -offload-build -O3 -mkl -S bench.cpp**

```
..B2.3:                         # Preds ..B2.1 ..B2.3 Latency 29
        vmadd213ps %v1, %v0, %v2                                #8.19 c1
        addl       $8, %eax                                     #6.38 c1
        cmpl       %esi, %eax                                   #6.25 c3
        vmadd213ps %v1, %v0, %v2                                #8.32 c5
        nop                                                     #8.45 c7
        vmadd213ps %v1, %v0, %v2                                #8.45 c9
        nop                                                     #8.58 c11
        vmadd213ps %v1, %v0, %v2                                #8.58 c13
        nop                                                     #9.19 c15
        vmadd213ps %v1, %v0, %v2                                #9.19 c17
        nop                                                     #9.32 c19
        vmadd213ps %v1, %v0, %v2                                #9.32 c21
        nop                                                     #9.45 c23
        vmadd213ps %v1, %v0, %v2                                #9.45 c25
        nop                                                     #9.58 c27
        vmadd213ps %v1, %v0, %v2                                #9.58 c29
        jl         ..B2.3       # Prob 82%                     #6.25 c29
```

Can’t get >50% of peak with this code (NOPs don’t pair)

Why NOPs?

Most hazards on MIC stall entire core, not a thread:

- Instruction cache miss
- L2 cache miss (but not prefetch)
- Full prefetch queue
- Vector register dependency
- Cache bank conflicts between U and V pipe
- Misaligned access
- Forwarding from store queue to load unit

Use NOP/DELAY to pause a thread to avoid core stall

- Compiler tends to add NOPs automatically
- It can be disabled: -offload-copts="-mCG_lrb_num_threads=4”
- (unofficial option, may be dropped in the future)

Two MADs in flight: 49% peak (1 core)

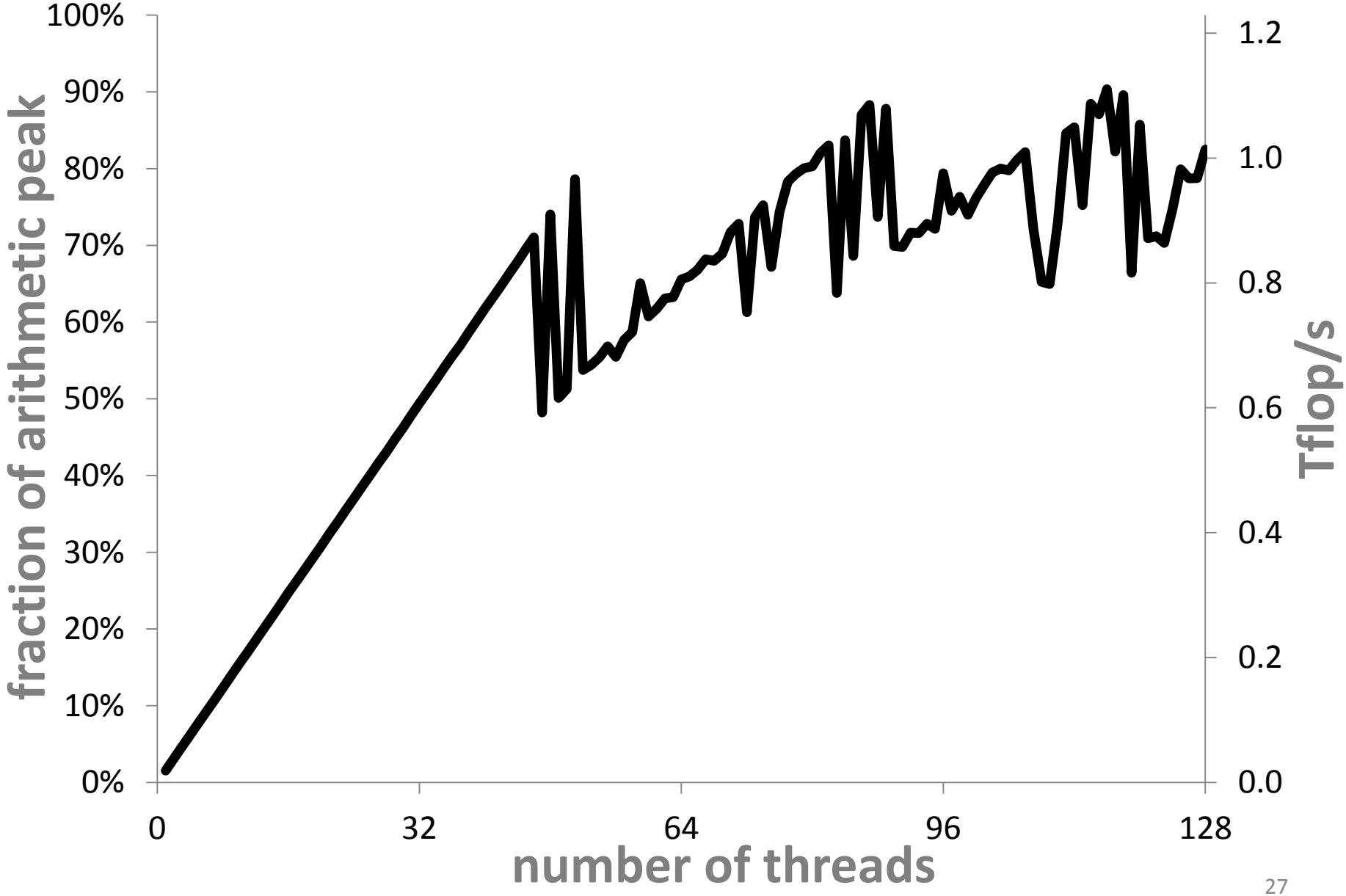
```
#ifdef __MIC__ //can't use intrinsics if not on MIC
#include <micvec.h>
__declspec(target(mic)) void foo( float &r, int niterations )
{
    F32vec16 a( 100.f ), b( 200.f ), c( 1.f ), x( 0.9f );
    for( int i = 0; i < niterations; i += 8 )
    {
        a = a*x + c; b = b*x + c; a = a*x + c; b = b*x + c;
        a = a*x + c; b = b*x + c; a = a*x + c; b = b*x + c;
    }
    r = reduce_add(a+b);
}
#else
__declspec(target(mic)) void foo(float &r, int niterations){}
#endif
```

No NOP problem here as no back-to-back dependencies

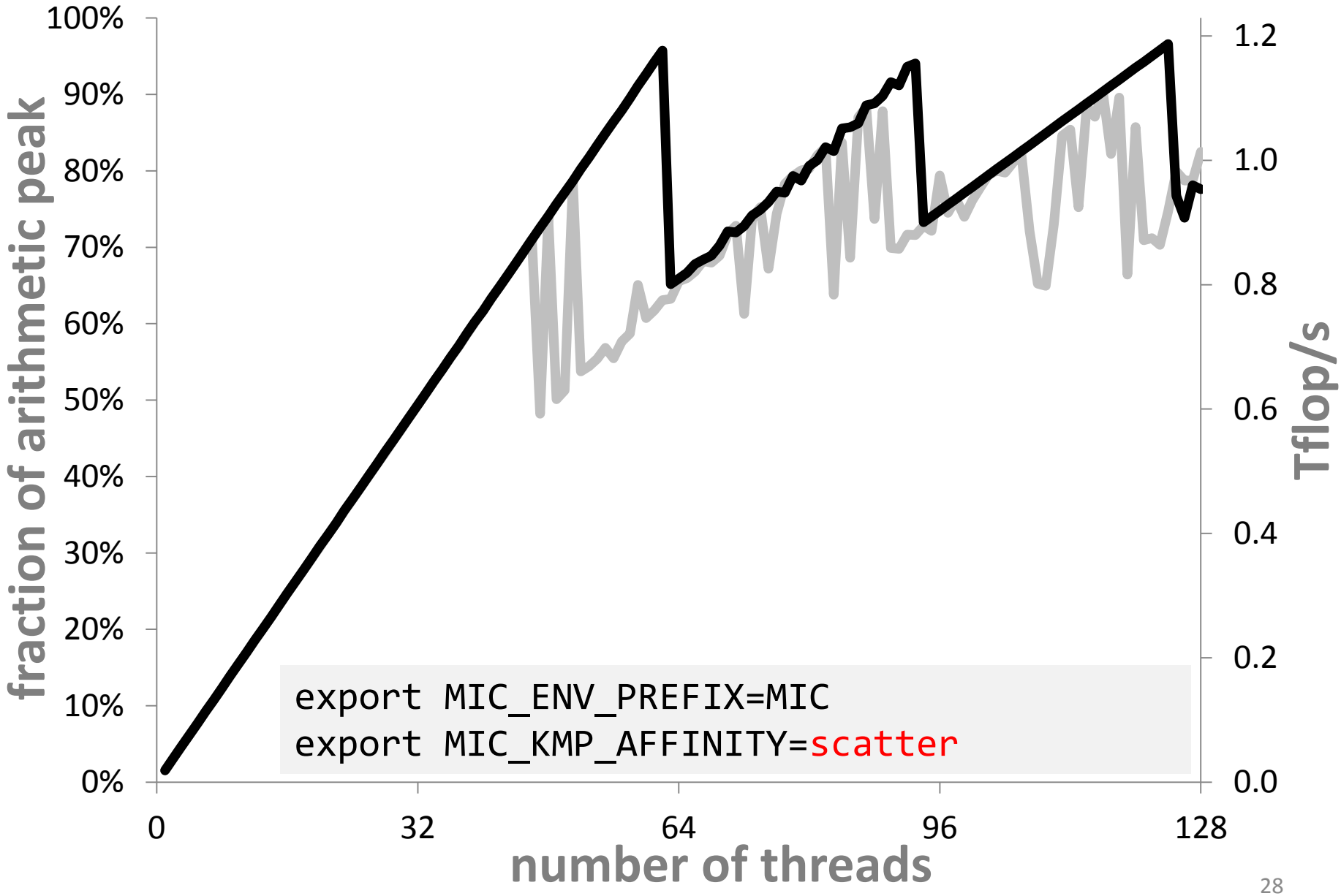
Run many threads using OpenMP

```
#include <stdio.h>
#include <omp.h>
#include <mkl.h>
main()
{
  float Gflops, r[1000];
  for( int i = 0; i < 128; i++ )
  {
    int niterations = 100<<20, nthreads = ((i*79)%128)+1;
    #pragma offload target(mic)
    for( int k = 0; k < 3; k++ )//first two runs are warm up
    {
      double t = dsecnd();
      #pragma omp parallel num_threads(nthreads)
        foo( r[omp_get_thread_num()], niterations );
      Gflops = 32e-9*niterations*nthreads/(dsecnd()-t);
    }
    printf( "%d threads, %g Gflop/s, %g\n", nthreads, Gflops, r[0]);
  }
}
```

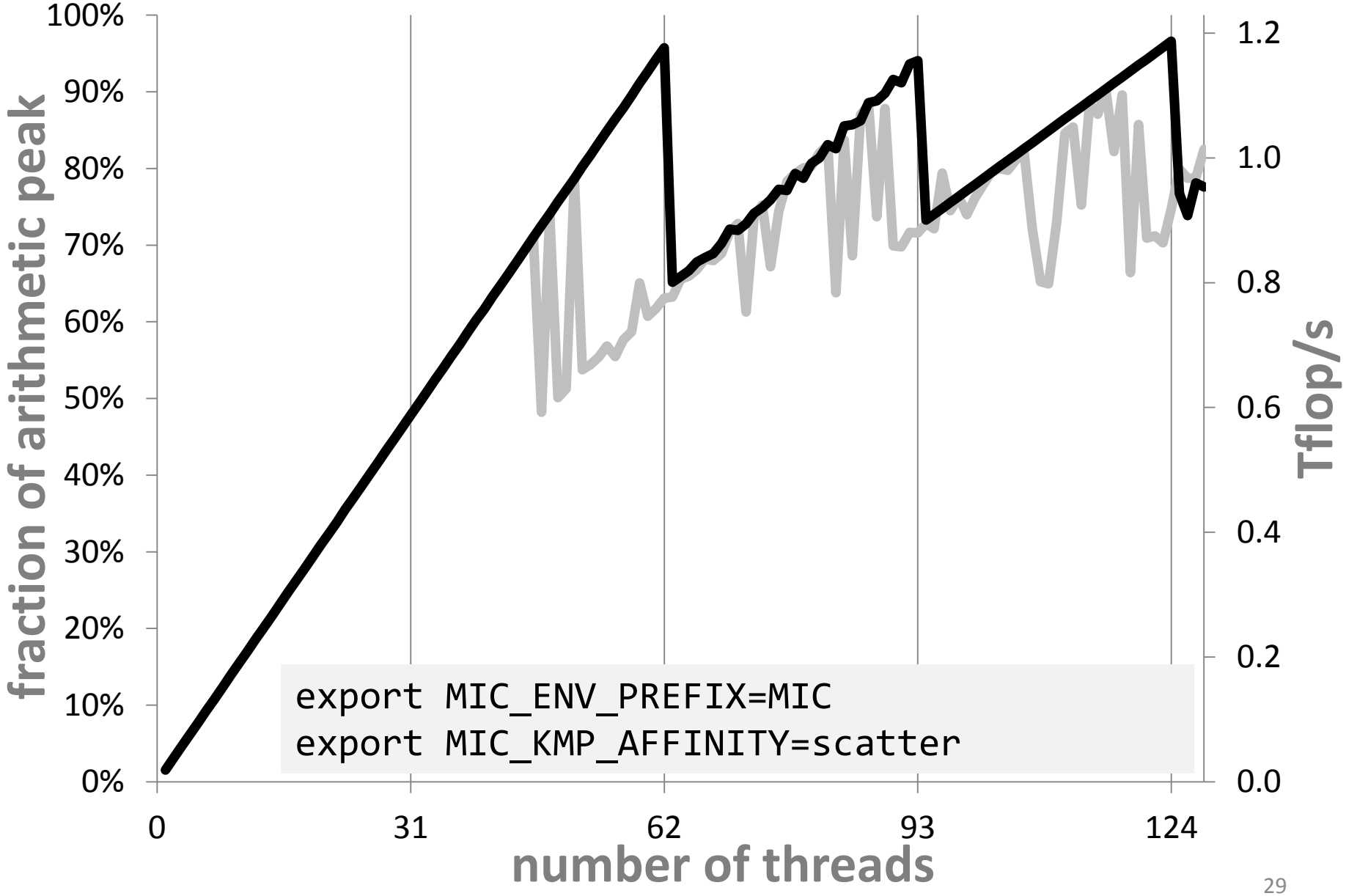
How throughput grows with #threads



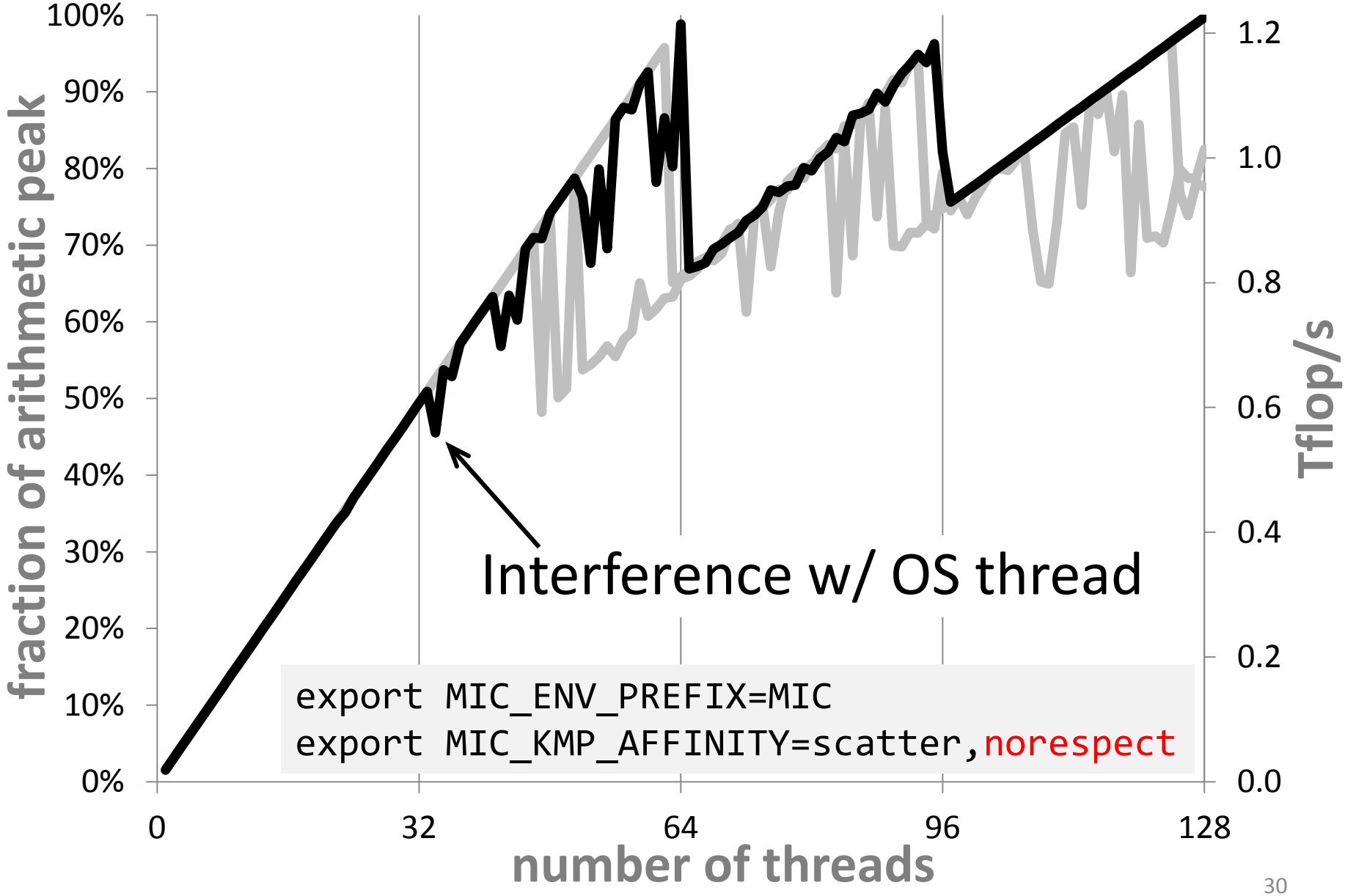
Enforce round-robin thread assignment



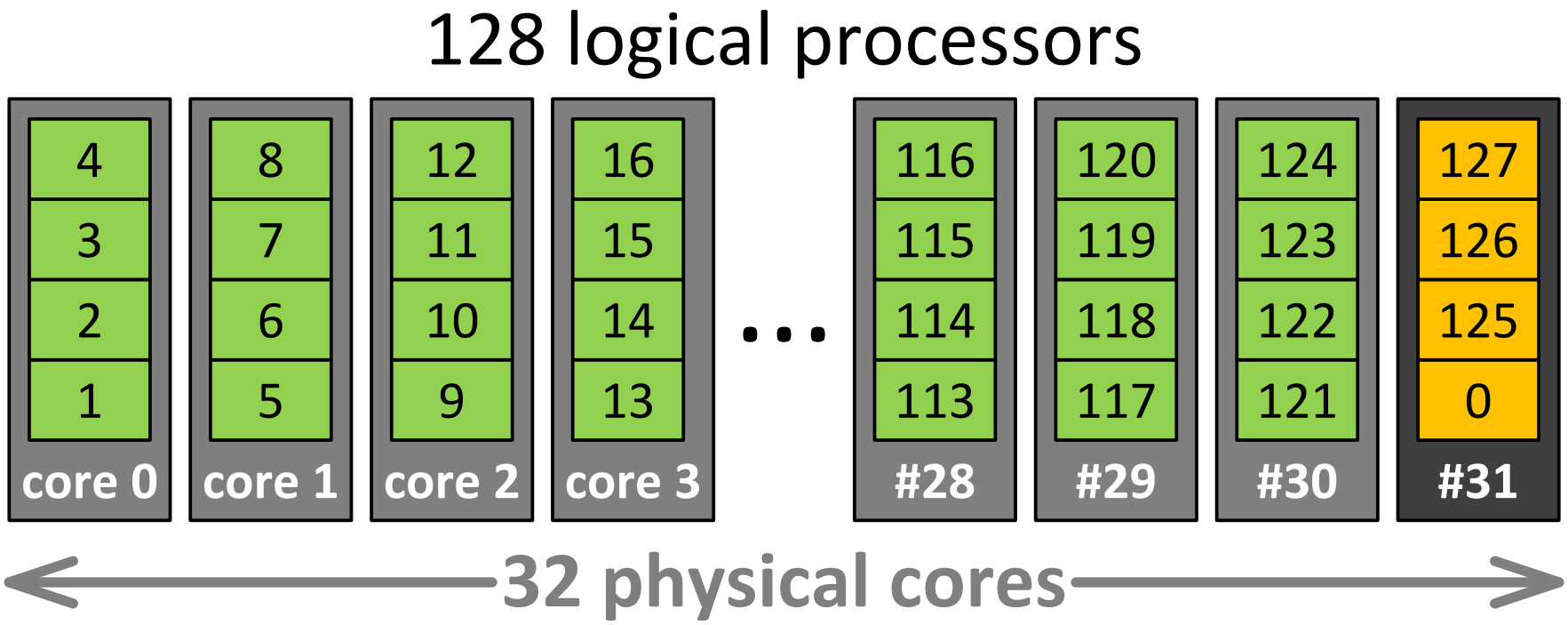
One core is reserved in offload mode



Enable the reserved core



Common KMP_AFFINITY settings



`norespect:` enables core #31 (invisible otherwise)
`compact:` assign in the order 1,2,3,4,…,124,0,125,…
`scatter:` round-robin order – 1,5,9,…,121,2,6,…
`explicit,proclist=[…]:` any user-defined order
`granularity=fine:` disable thread migration within core

Figuring out thread affinity

export MIC_KMP_AFFINITY=...,verbose

- Reports mapping of OpenMP threads to OS threads to hardware thread contexts

sched_getcpu() / #include <sched.h>

- Returns current OS thread

```
#define cpuid_getcpu() ({unsigned int eax,ebx;__asm__\
("cpuid":"=a"(eax),"=b"(ebx):"0"(1):"cx","dx");ebx>>24;})
```

- cpuid_getcpu()/4 = physical core
- cpuid_getcpu()%4 = local thread context

Part IV

Memory peak

Prefetching on MIC

Chip pin bandwidth: 256 pins@1.5GHz = 96 GB/s

- Need 30 KB in flight to hide 300 cycle latency
- This is over 200 B/thread if using 128 threads
- Cache line size is 64 B

So, multithreading alone is not enough

- But entire core stalls on an L2 miss anyway

Must use software prefetching (no HW prefetch)

- Inserted by compiler or manually
- #pragma noprefetch disables compiler prefetching
- Page fault if prefetching invalid pages
  - **Allocate extra trailing space**

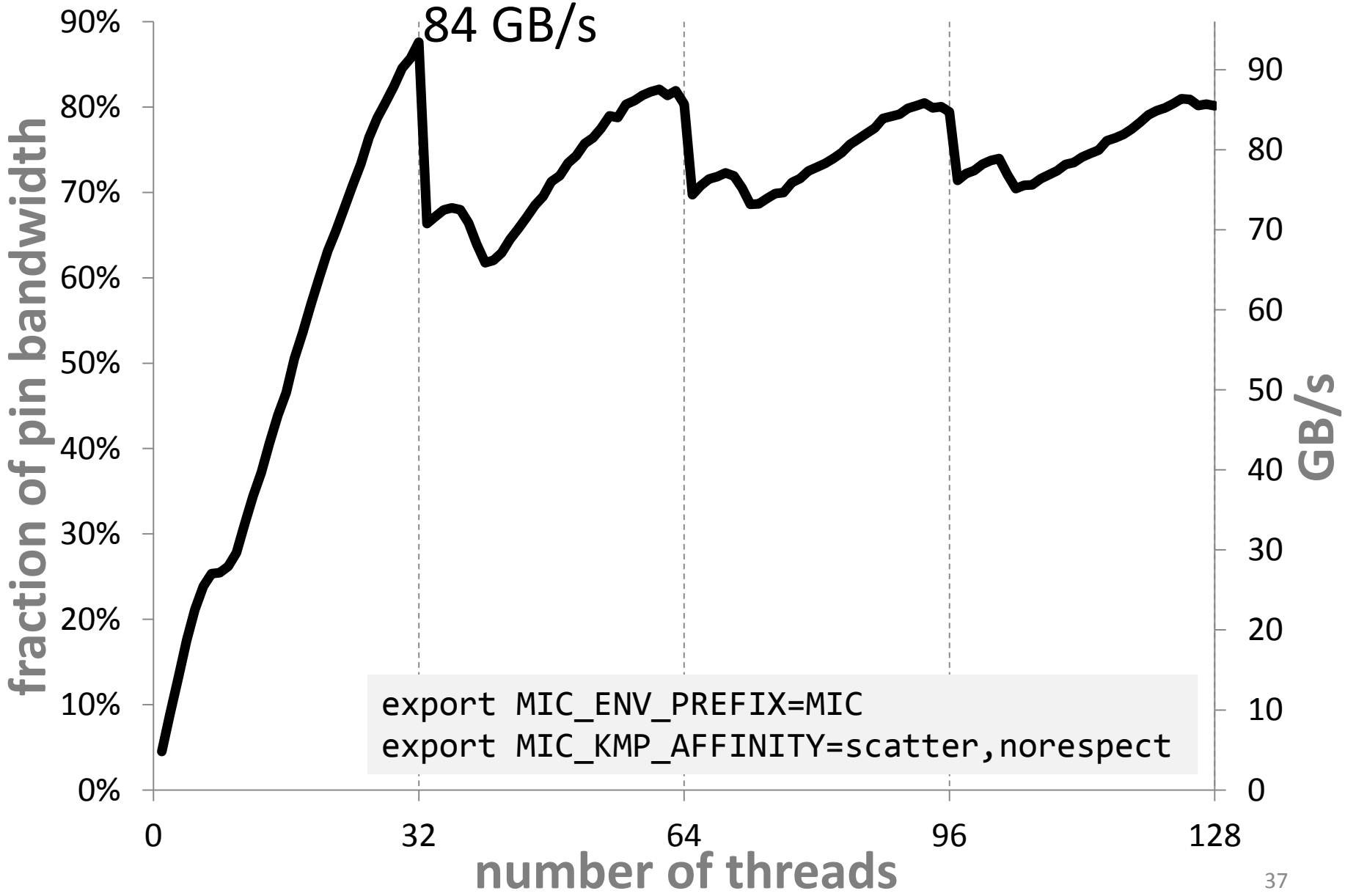
Manual prefetching example (1/2)

```
#include <immintrin.h>
__declspec(target(mic)) void foo( float &r, float *p, int n, int reps )
{
#ifdef __MIC__
   __m512 a = _mm512_set_1to16_ps( 0 );
   for( int j = 0; j < reps; j++ )
#pragma noprefetch
      for( int i = 0; i < n; i += 16 )
      {
         a = _mm512_add_ps( a, _mm512_loadd( p+i,
             _MM_FULLUPC_NONE, _MM_BROADCAST32_NONE, _MM_HINT_NONE ));
         _mm_vprefetch1( p+i+16*16, _MM_PFHINT_NONE );
         _mm_vprefetch2( p+i+32*16, _MM_PFHINT_NONE );
      }
   r = _mm512_reduce_add_ps( a );
#endif
}
```

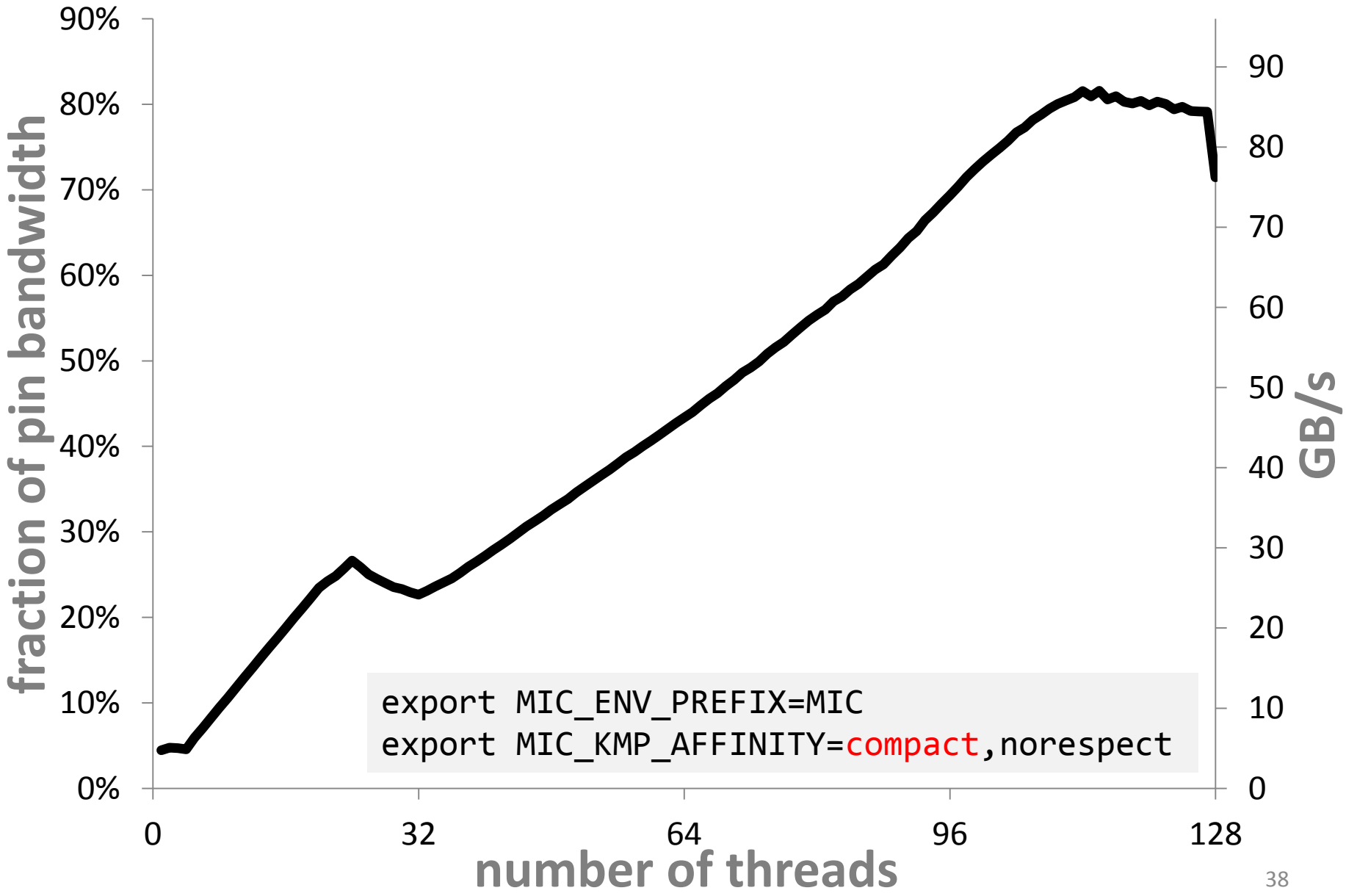
Manual prefetching example (2/2)

```
#include <stdio.h>
#include <omp.h>
#include <mkl.h>
main( )
{
  int n = 2<<18, nthreads_max = 128, reps = 32;
  float *A = (float*)malloc((nthreads_max*n+32*16)*sizeof(float)), r[128];
  for( int nthreads = 1; nthreads <= nthreads_max; nthreads++ )
  {
    double t;
    #pragma offload target(mic) in(A:length(nthreads*n+32*16) align(64))
    for( int k = 0; k < 3; k++ )
    {
      t = dsecnd();
      #pragma omp parallel num_threads(nthreads)
        foo( r[omp_get_thread_num()], A+n*omp_get_thread_num(), n, reps );
      t = dsecnd()-t;
    }
    double GBs = 1e-9*sizeof(float)*reps*n*nthreads / t;
    printf( "%d threads, %.3f GB/s, %g\n", nthreads, GBs, r[0] );
  }
}
```

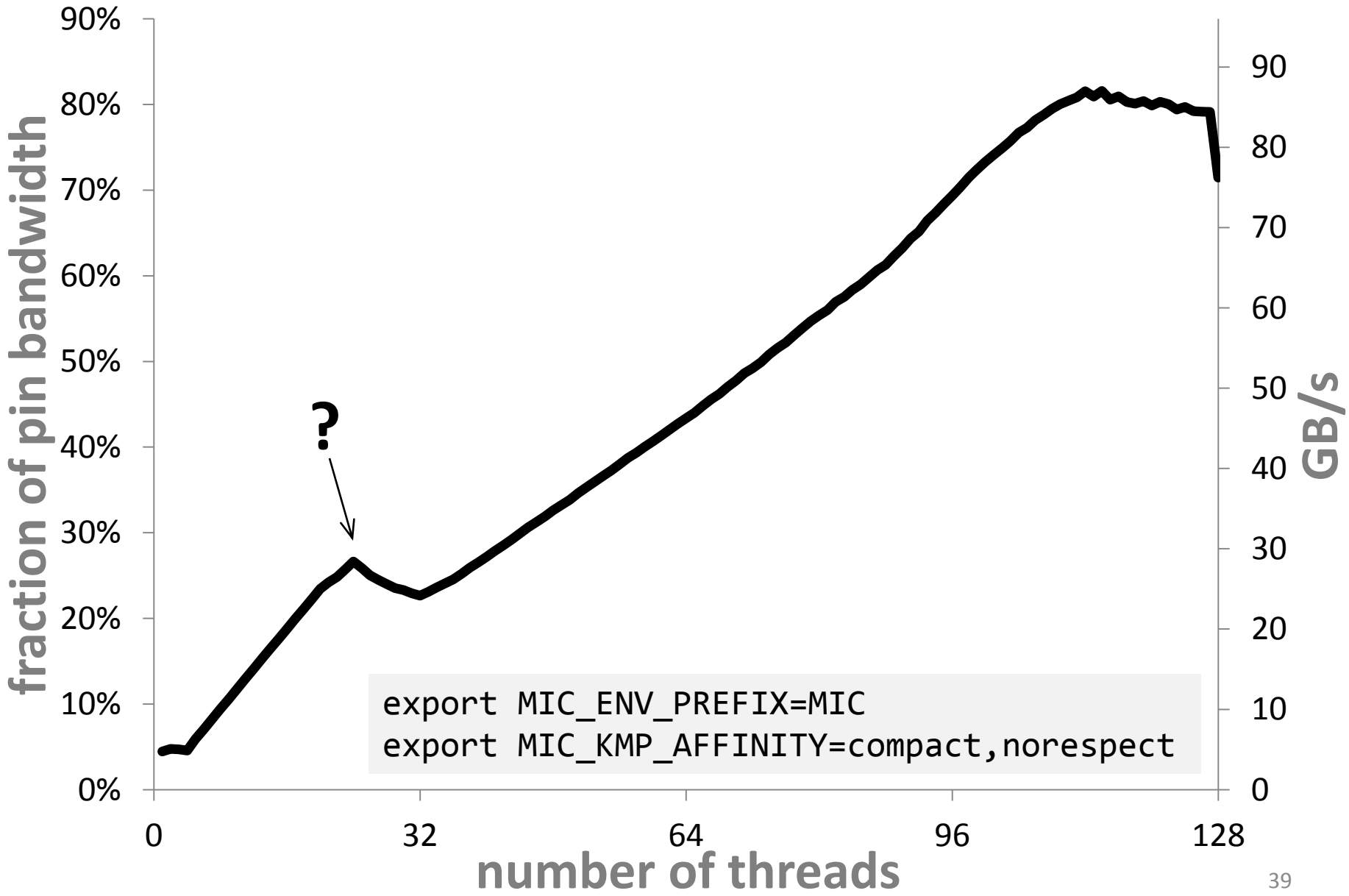
All cores, 1 thread per core: 88% of peak



But not if many threads, a few cores

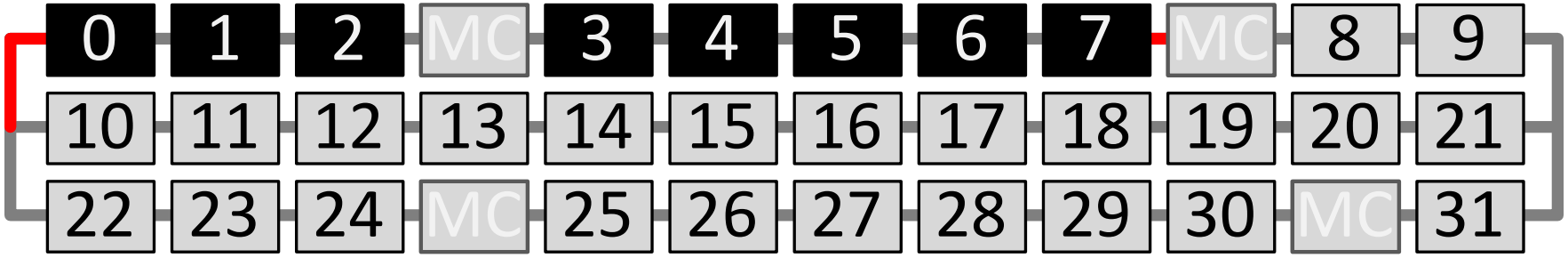


How would you explain this feature?

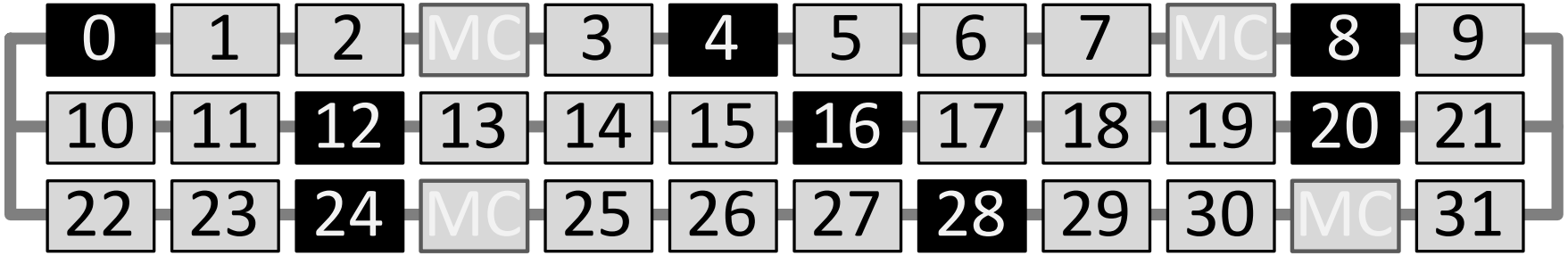


Hypothesis: contention on ring

When assigning threads linearly, most memory and coherency traffic has to go via **2 links**:



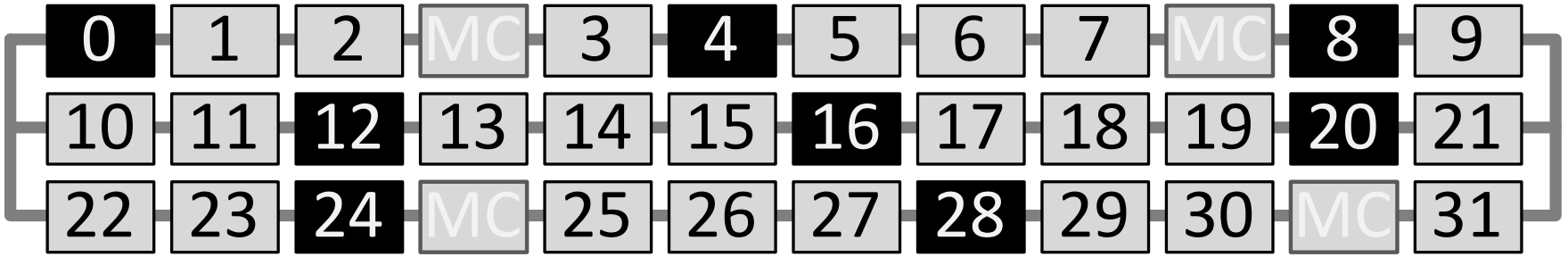
Solution: scatter threads more evenly around the die



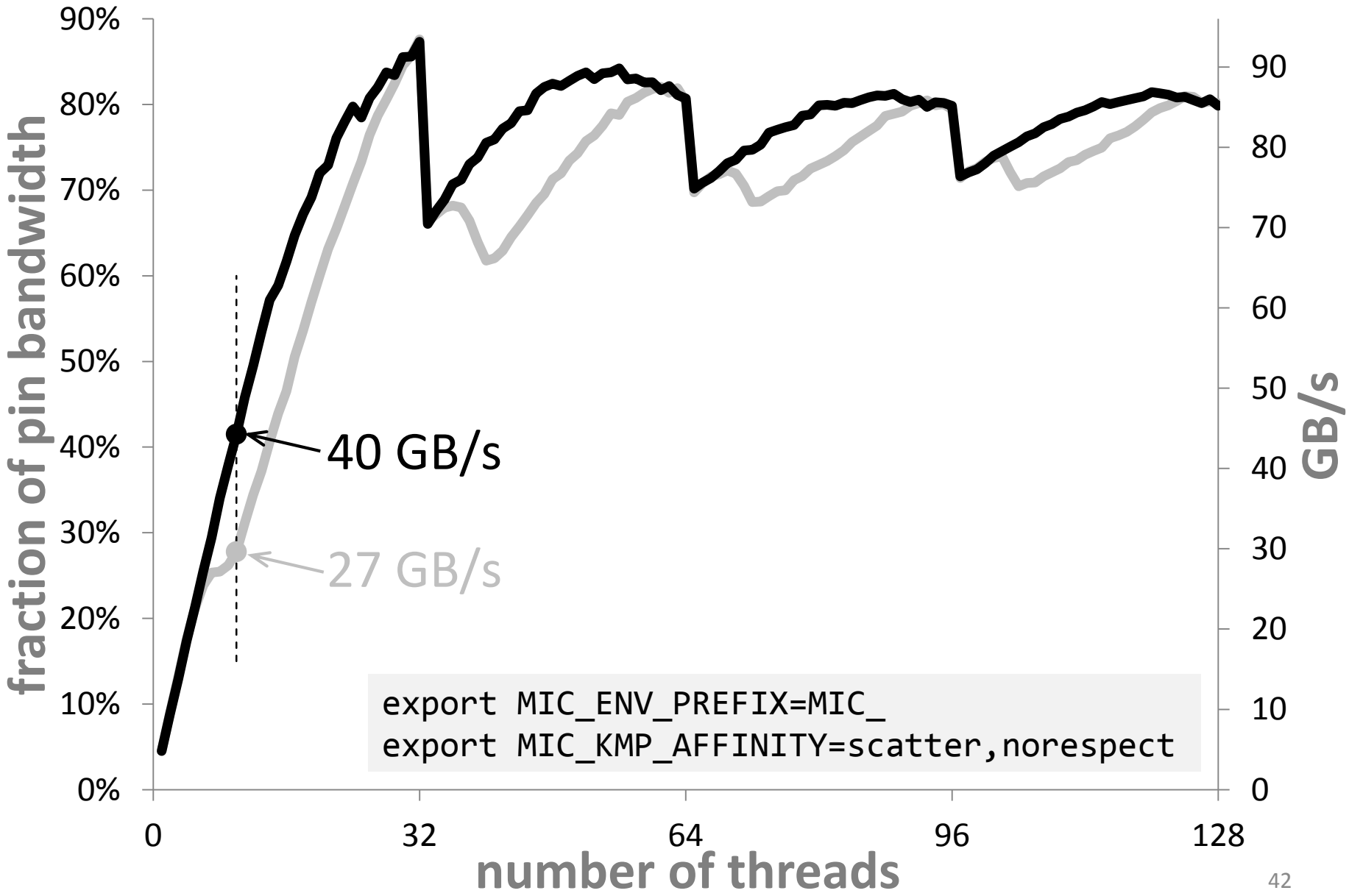
Affinity settings to resolve the contention

```
export MIC_ENV_PREFIX=MIC
```

export MIC_KMP_AFFINITY=explicit,norespect,proclist=[1,65,33,
97,17,81,49,113,9,73,41,105,25,89,57,121,5,69,37,101,21,85,53,
117,13,77,45,109,29,93,61,0,2,66,34,98,18,82,50,114,10,74,42,
106,26,90,58,122,6,70,38,102,22,86,54,118,14,78,46,110,30,94,
62,125,3,67,35,99,19,83,51,115,11,75,43,107,27,91,59,123,7,71,
39,103,23,87,55,119,15,79,47,111,31,95,63,126,4,68,36,100,20,
84,52,116,12,76,44,108,28,92,60,124,8,72,40,104,24,88,56,120,
16,80,48,112,32,96,64,127]



Result: up to 1.5x speedup



Part V

MKL SGEMM

Recommended settings for MKL SGEMM

MIC supports 4KB (default), 64KB and 2MB pages

Must use 2MB pages for MKL, compact affinity:

```
export MIC_ENV_PREFIX=MIC
export MIC_USE_2MB_BUFFERS=1K
export MIC_KMP_AFFINITY=compact
```

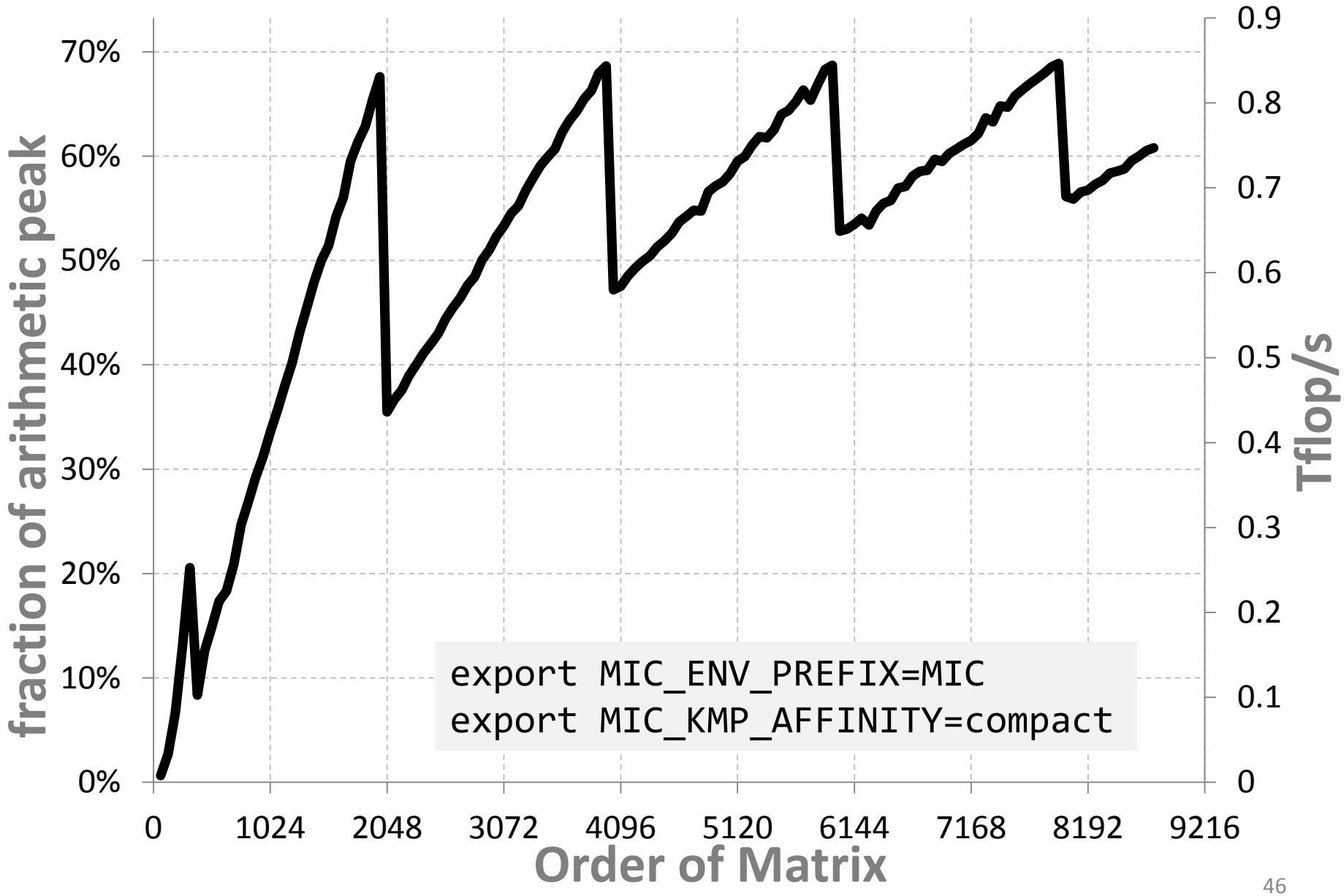
Also:

- 64B alignment
- size multiple of 16 (but 64 is slightly better)
- padding

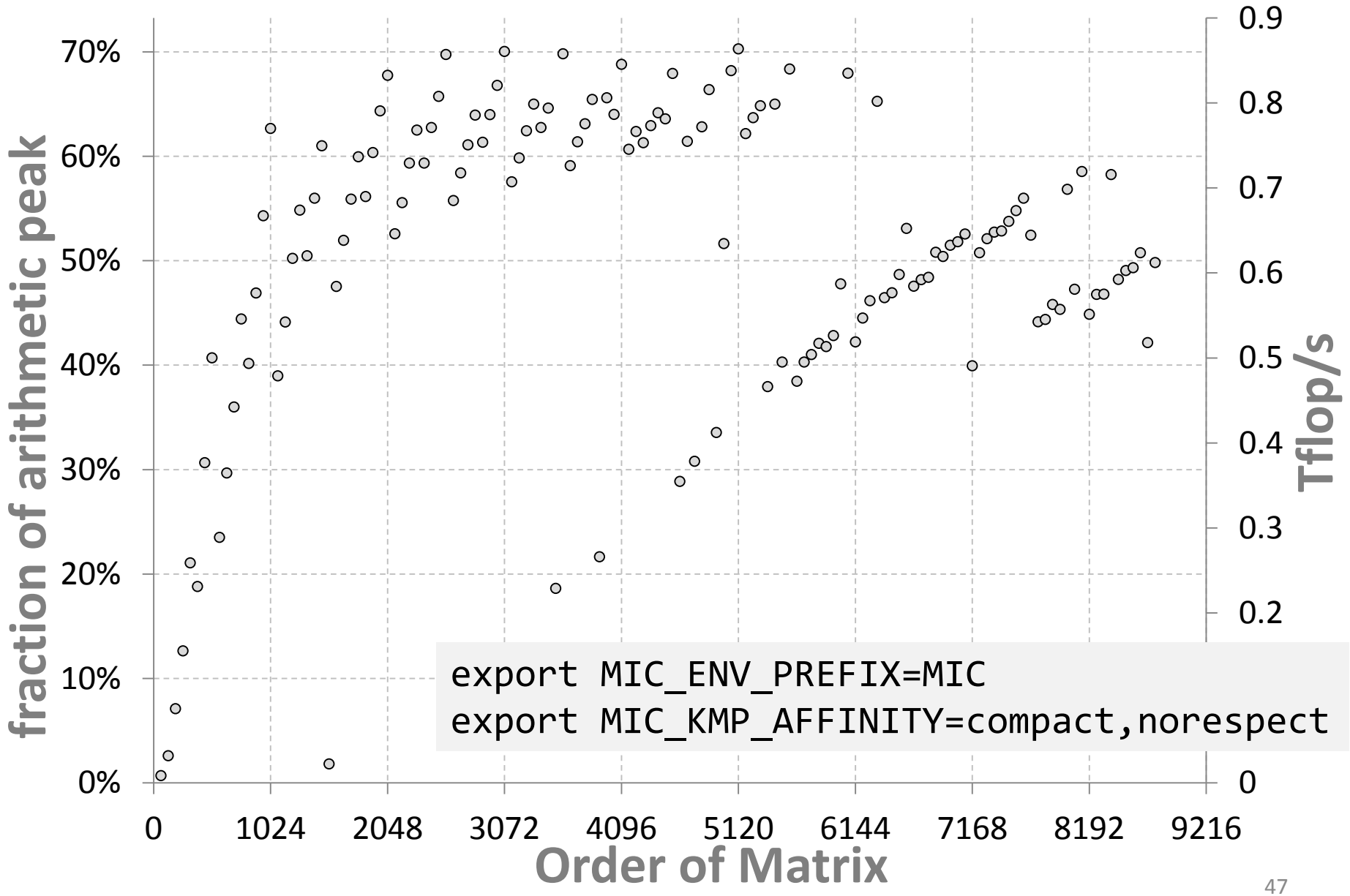
Benchmarking SGEMM

```
#include <stdio.h>
#include <mkl.h>
#include <mkl_blas.h>
main( )
{
  int nmax = 8768;
  float *pool = (float*)malloc( 3*nmax*(nmax|64)*sizeof(float) );
  for( int n = 64; n <= nmax; n += 64 )//size multiple of 64
  {
    int lda = n|64;//a simple padding
    double t;
    for( int j = 0; j < 3*n*lda; j++ ) pool[j] = drand48( )*2-1;
    #pragma offload target(mic) inout(pool:length(3*n*lda) align(64))
    {
      float alpha=1, beta=1, *A=pool, *B=pool+n*lda, *C=pool+2*n*lda;
      sgemm( "N","N", &n,&n,&n, &alpha, A, &lda, B, &lda, &beta, C, &lda );
      t = dsecnd();
      sgemm( "N","N", &n,&n,&n, &alpha, A, &lda, B, &lda, &beta, C, &lda );
      t = dsecnd() - t;
    }
    printf( "%d %g Gflop/s \n", n, 2e-9*n*n*n/t );
  }
}
```

Performance of SGEMM on 31 cores



Performance of SGEMM on 32 cores



Benchmarking SGEMM in native mode

```
#include <stdio.h>
#include <mkl.h>
#include <mkl_blas.h>
#include <sys/mman.h> // for allocating 2 MB pages
main( )
{
  int nmax = 8768;
  float *pool = (float*)mmap( NULL, 3*nmax*(nmax|64)*sizeof(float),
 PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE | MAP_HUGETLB, -1, 0 );
  for( int n = 64; n <= nmax; n += 64 )
  {
    int lda = n|64;
    double t;
    for( int j = 0; j < 3*n*lda; j++ ) pool[j] = drand48( )*2-1;
    float alpha=1, beta=1, *A=pool, *B=pool+n*lda, *C=pool+2*n*lda;
    sgemm( "N","N", &n,&n,&n, &alpha, A, &lda, B, &lda, &beta, C, &lda );
    t = dsecnd();
    sgemm( "N","N", &n,&n,&n, &alpha, A, &lda, B, &lda, &beta, C, &lda );
    t = dsecnd() - t;
    printf( "%d %g Gflop/s\n", n, 2e-9*n*n*n/t );
  }
}
```

Running benchmark in native mode

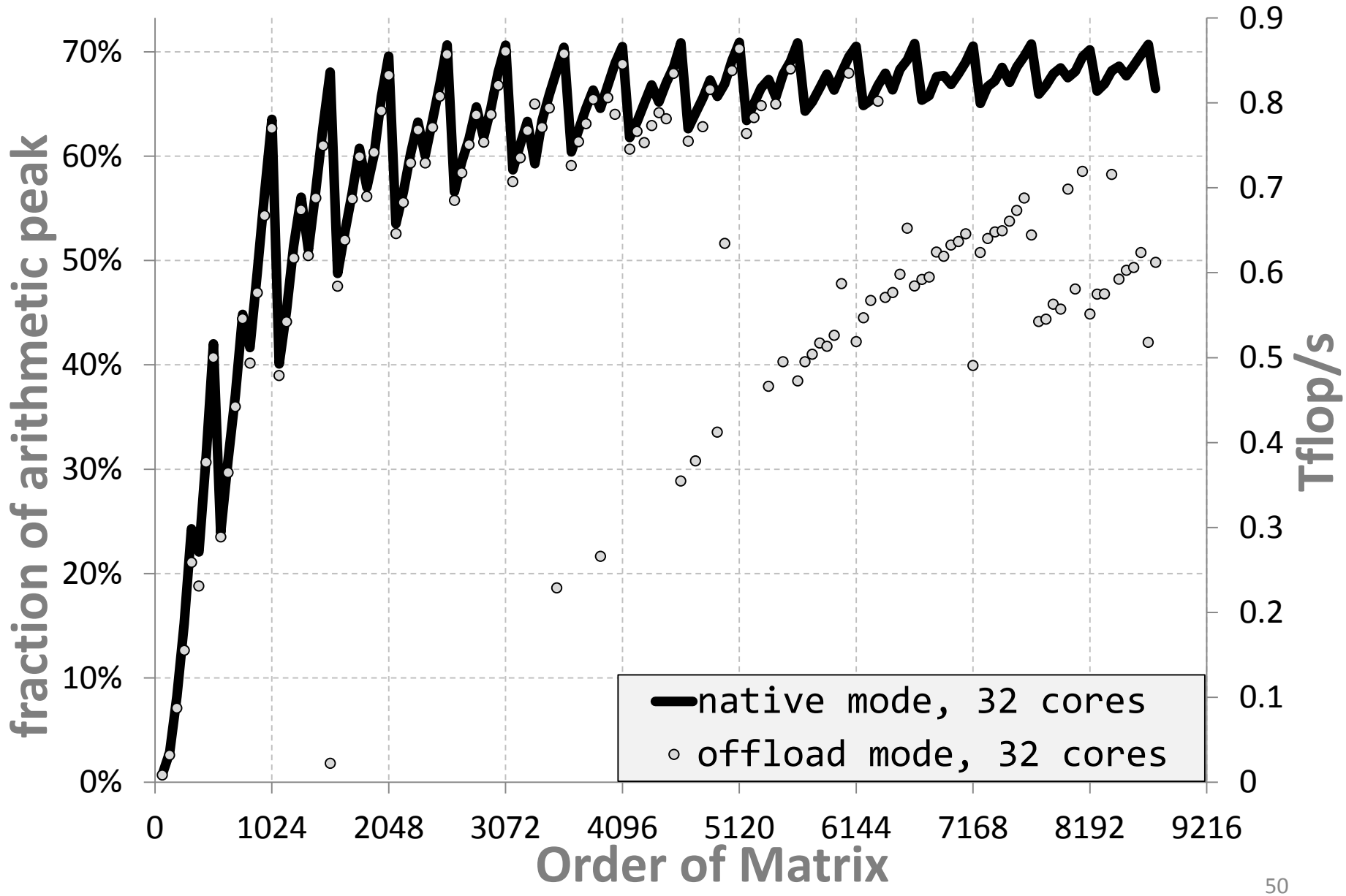
Host: compile, upload the binary and libiomp5.so

```
icc -mmic -O3 -mkl bench.cpp
sudo micput 192.168.1.100 ./a.out /tmp/a.out
sudo micput 192.168.1.100 /opt/intel/composerxe_mic/
     compiler/lib/mic/libiomp5.so /tmp
sudo ssh -i /opt/intel/mic/id_rsa root@192.168.1.100
```

MIC: configure 2MB page allocation and affinity

```
echo 500 > /proc/sys/vm/nr_hugepages
export KMP_AFFINITY=compact
export LD_LIBRARY_PATH=/tmp
/tmp/a.out
```

Using 32 cores: native vs offload mode



Part VI

MKL SGETRF (≈LINPACK benchmark)

Recommended settings for SGETRF

Similar to SGEMM:

- 64B alignment
- Padding
- Size multiple of 64

Allocate extra trailing space (for prefetching?)

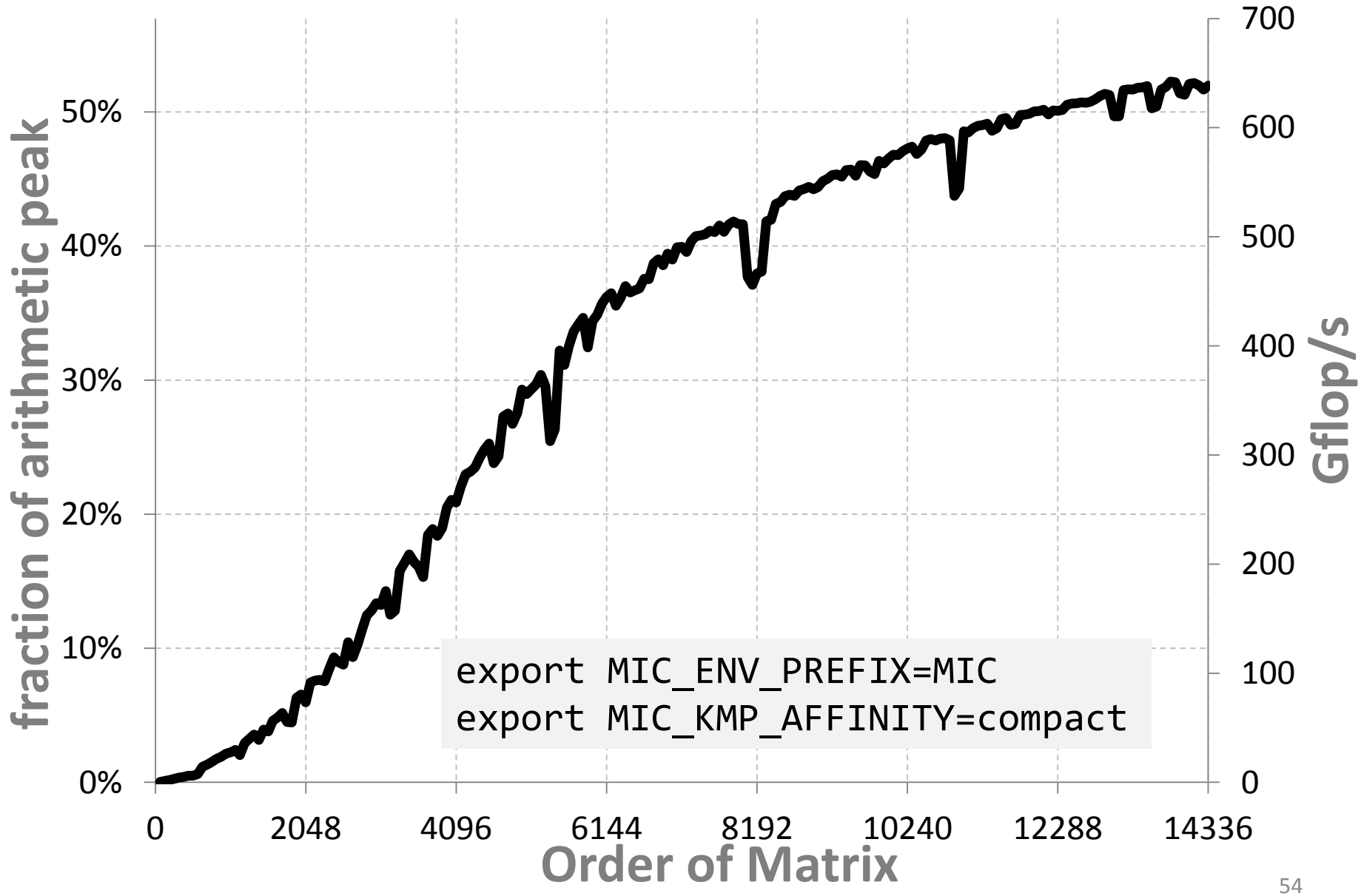
```
export MIC_ENV_PREFIX=MIC
export MIC_USE_2MB_BUFFERS=1K
export MIC_KMP_AFFINITY=compact
```

Benchmarking SGETRF

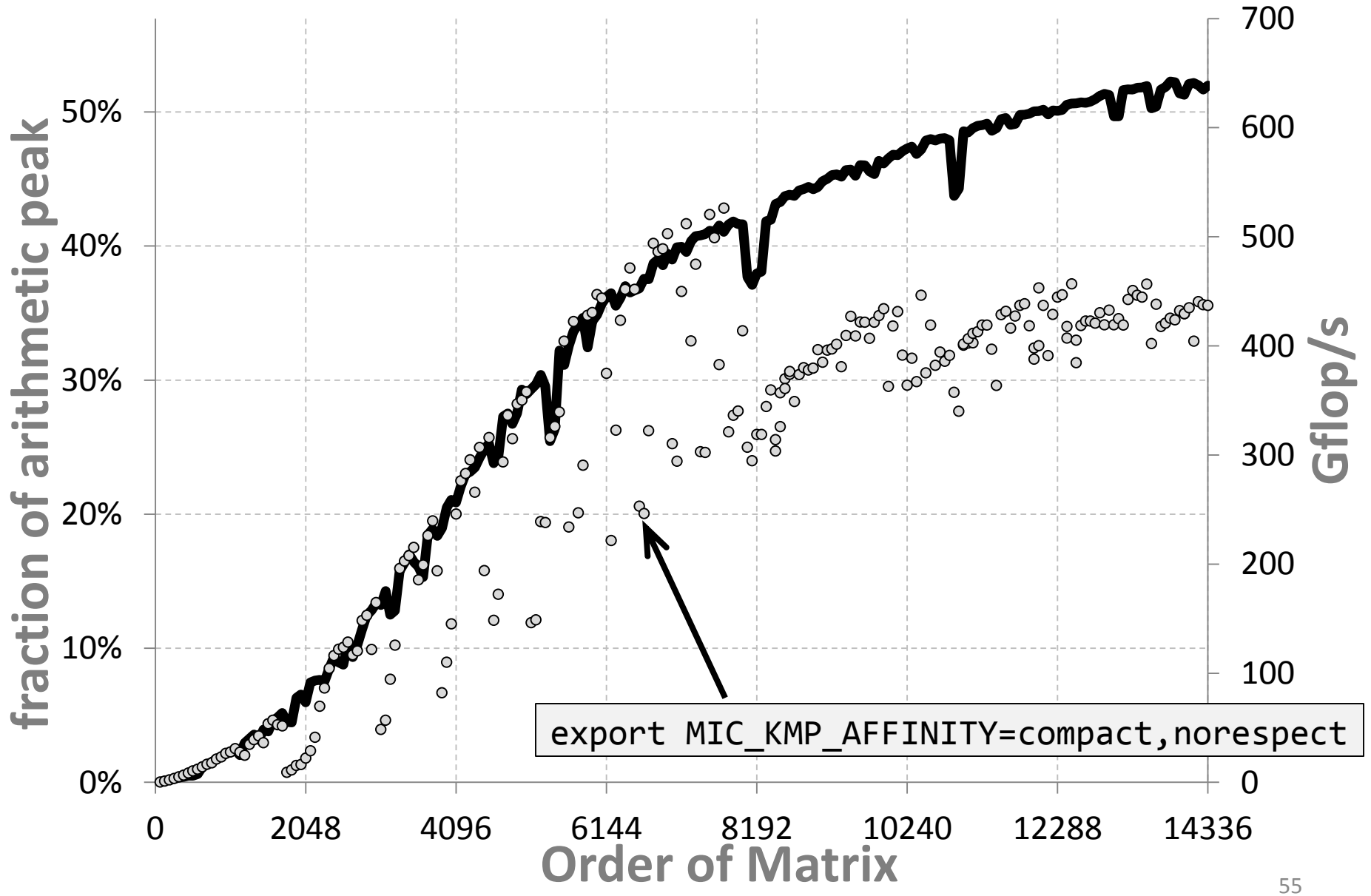
```
#include <stdio.h>
#include <mkl.h>
#include <mkl_lapack.h>
main( )
{
    const int nmax = 14336;
    float *A = (float*)malloc( (nmax+128)*(nmax|64)*sizeof(float) );
    for( int n = 64; n <= nmax; n += 64 )//size multiple of 64
    {
        double t;
        int ipiv[nmax], lda = n|64;//padding
        #pragma offload target(mic) in(A:length((n+128)*lda) align(64))
        {
            for( int info, i = 0; i < 3; i++ )
            {
                for( int j = 0; j < n*lda; j++ ) A[j] = drand48( )*2-1;
                t = dsecnd( );
                sgetrf( &n,&n, A, &lda, ipiv, &info );
                t = dsecnd() - t;
            }
        }
        printf( "%d %g Gflop/s\n", n, 2e-9*n*n*n/3/t );
    }
}
```

extra space for prefetching?

52% of peak only (was 68% in SGEMM)



Using all 32 cores is troublesome again



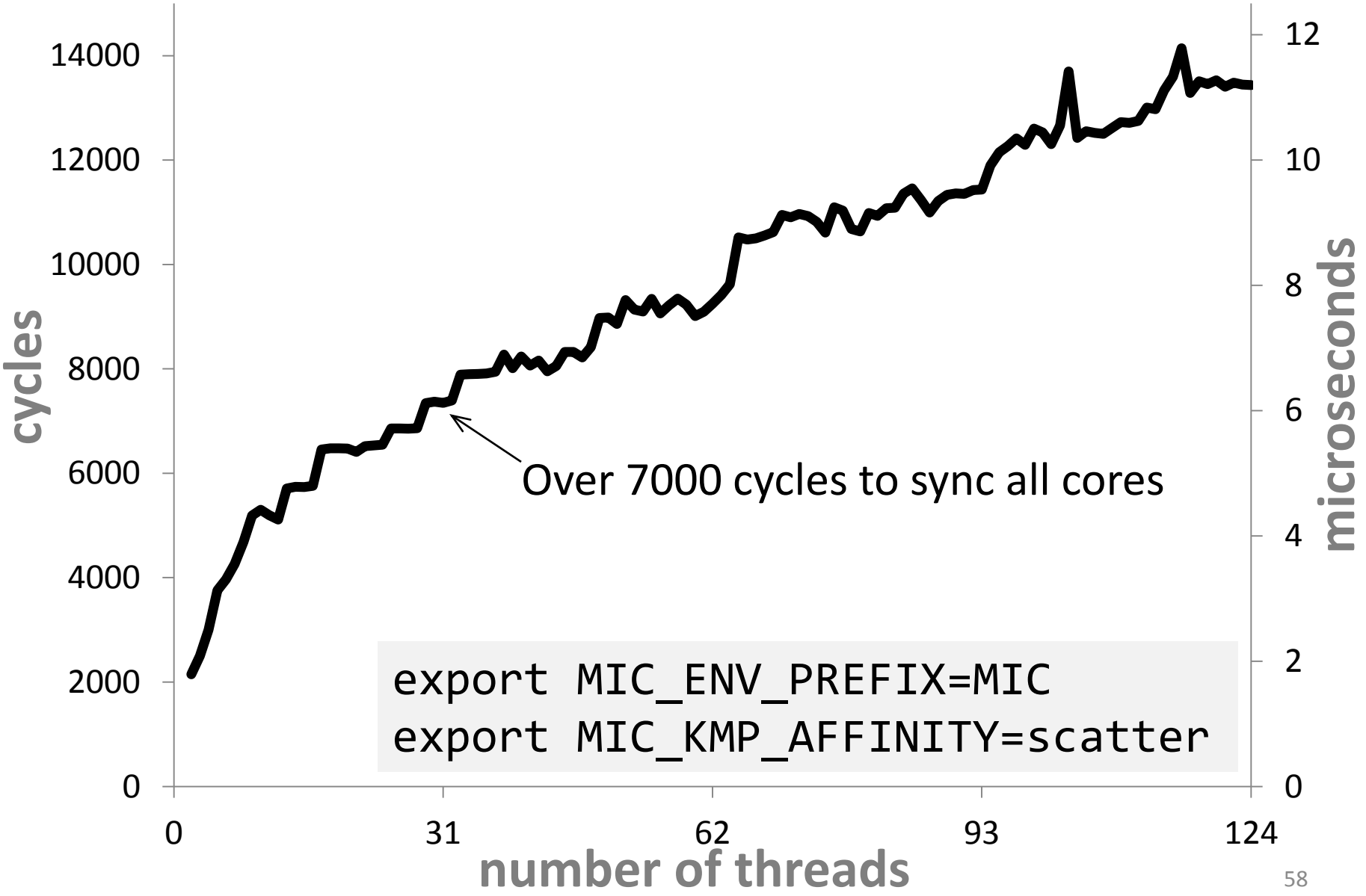
Part VII

Global barrier

A simplistic barrier benchmark

```
#include <stdio.h>
#include <immintrin.h>
#include <omp.h>
main()
{
    for( int nthreads = 2; nthreads <= 124; nthreads++ )
    {
        double cycles;
        #pragma offload target(mic)
        {
            int n = 100000;
            __int64 c0 = _rdtsc();
            #pragma omp parallel num_threads(nthreads)
            {
                for( int i = 0; i < n; i++ )
                {
                    #pragma omp barrier
                }
            }
            cycles = (_rdtsc() - c0)/(double)n;
        }
        printf( "%d threads, %g cycles\n", nthreads, cycles );
    }
}
```

10,000 cycles per barrier?



Barrier cost is important

N.B. 7,000 cycles ≈ 7,000,000 flops at peak rate

Expensive synchronization is a common motivation for heterogeneous algorithms

E.g. synchronization cost with GPU is ≈5μs, so we typically leave synchronization-heavy work on CPU

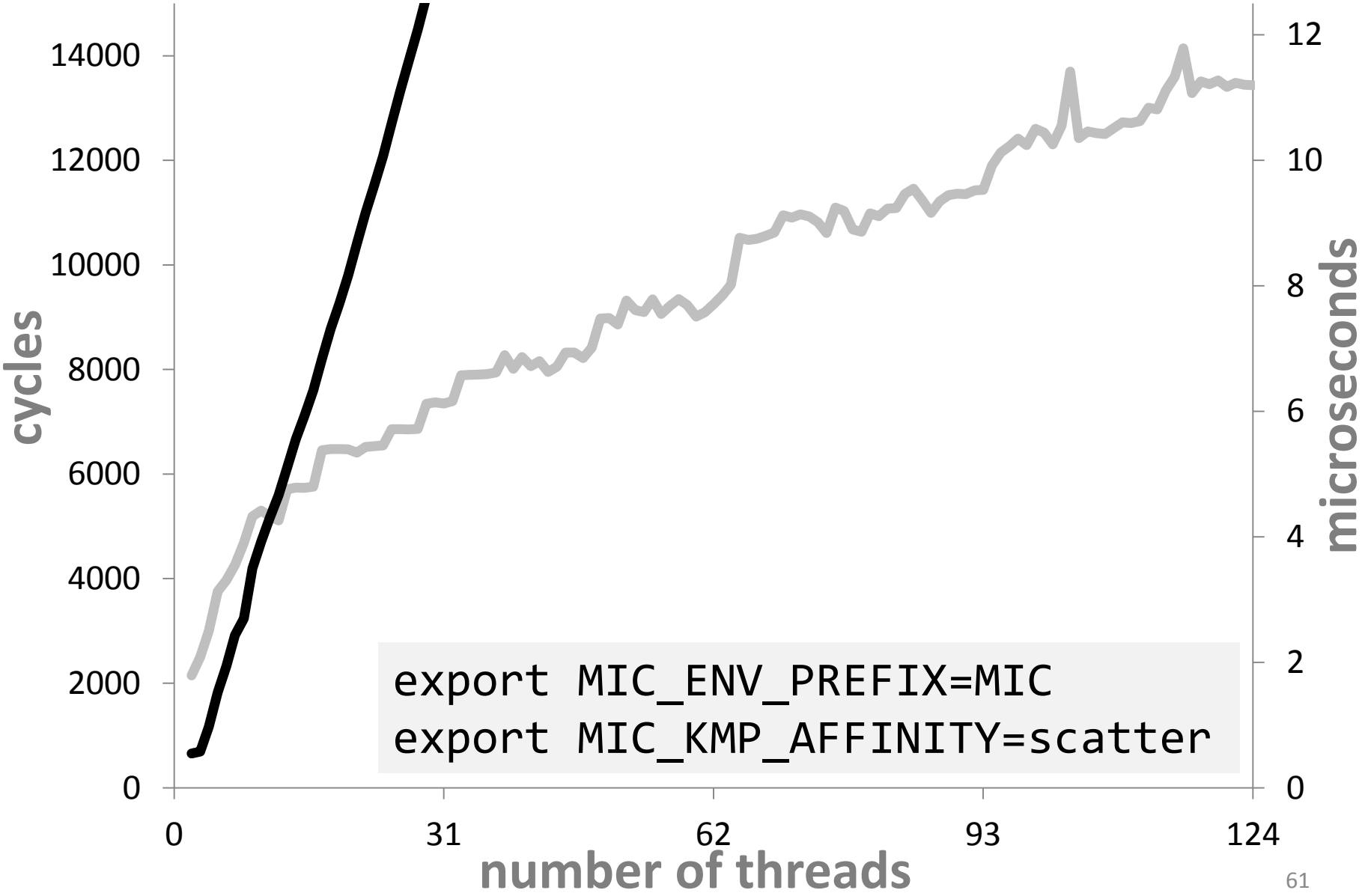
- More expensive synchronization on a device = more work should be run elsewhere

How sync across die can be as expensive as sync across PCIe?

Naïve barrier

```
#include <stdio.h>
#include <immintrin.h>
main()
{
   for( int nthreads = 2; nthreads <= 124; nthreads++ )
   {
      double cycles;
      #pragma offload target(mic)
      {
         int n = 100000;
         volatile int counter = 0;
         __int64 c0 = _rdtsc();
         #pragma omp parallel num_threads(nthreads)
         {
            for( int i = 0; i < n; i++ )
            {
              __sync_fetch_and_add( &counter, 1 );//vote
              while( counter < (i+1)*nthreads ); //wait until all voted
            }
         }
         cycles = (_rdtsc() - c0)/(double)n;
      }
      printf( "%d threads, %g cycles\n", nthreads, cycles );
   }
}
```

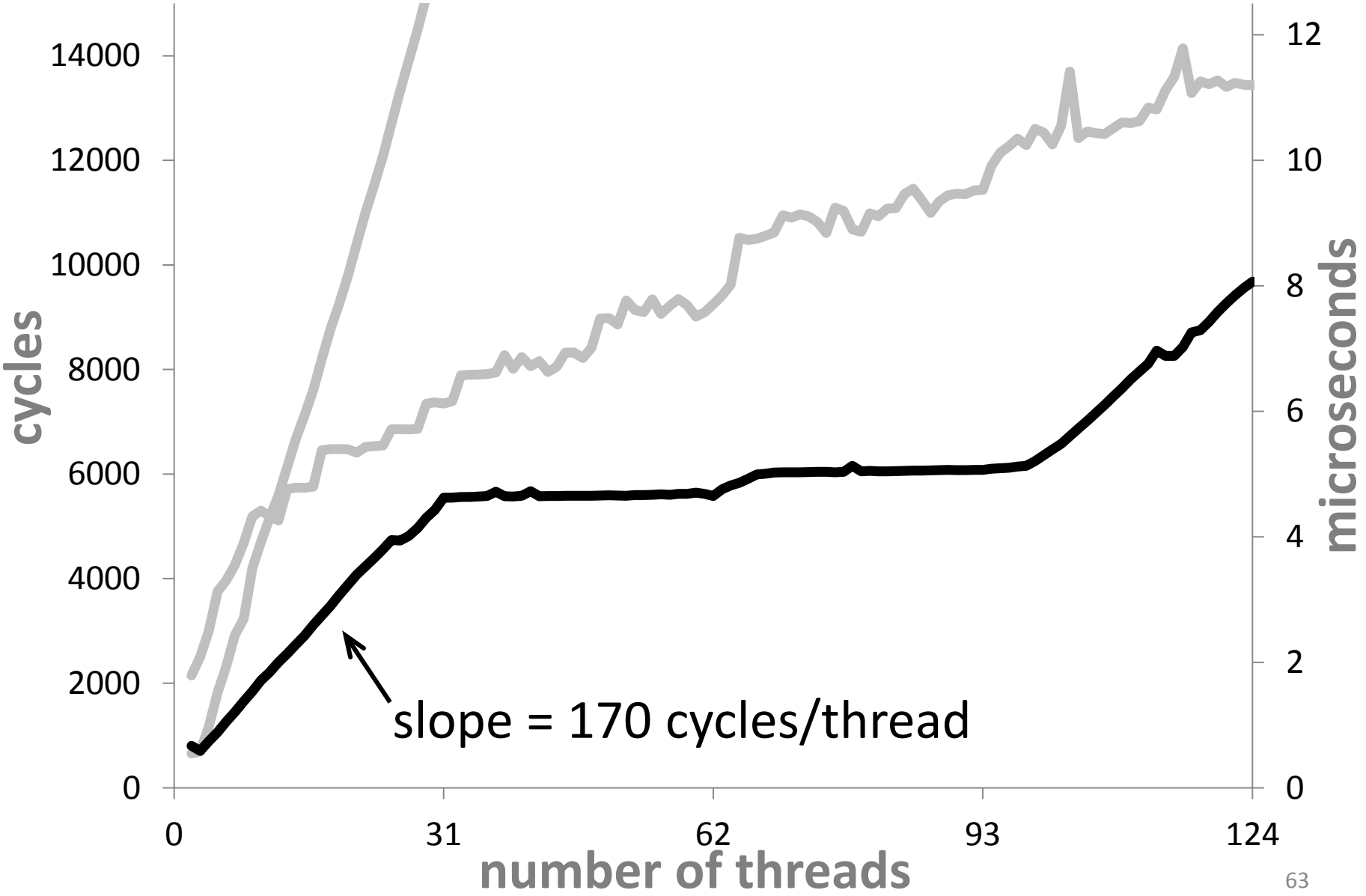
Naïve barrier is slower



Simple barrier with sense

```
#include <stdio.h>
#include <immintrin.h>
main()
{
   for( int nthreads = 2; nthreads <= 124; nthreads++ )
   {
      double cycles;
      #pragma offload target(mic)
      {
         int n = 100000;
         __declspec(align(64)) volatile int counter = 0, sense = 0;
         __int64 c0 = _rdtsc();
         #pragma omp parallel num_threads(nthreads)
         {
            for( int i = 1; i <= n; i++ )
            {
                int old = __sync_fetch_and_add( &counter, 1 );
                if( old+1 == i*nthreads ) sense = i;
                else                      while( sense < i );
            }
         }
         cycles = (_rdtsc() - c0)/(double)n;
      }
      printf( "%d threads, %g cycles\n", nthreads, cycles );
   }
}
```

Simple barrier beats OpenMP



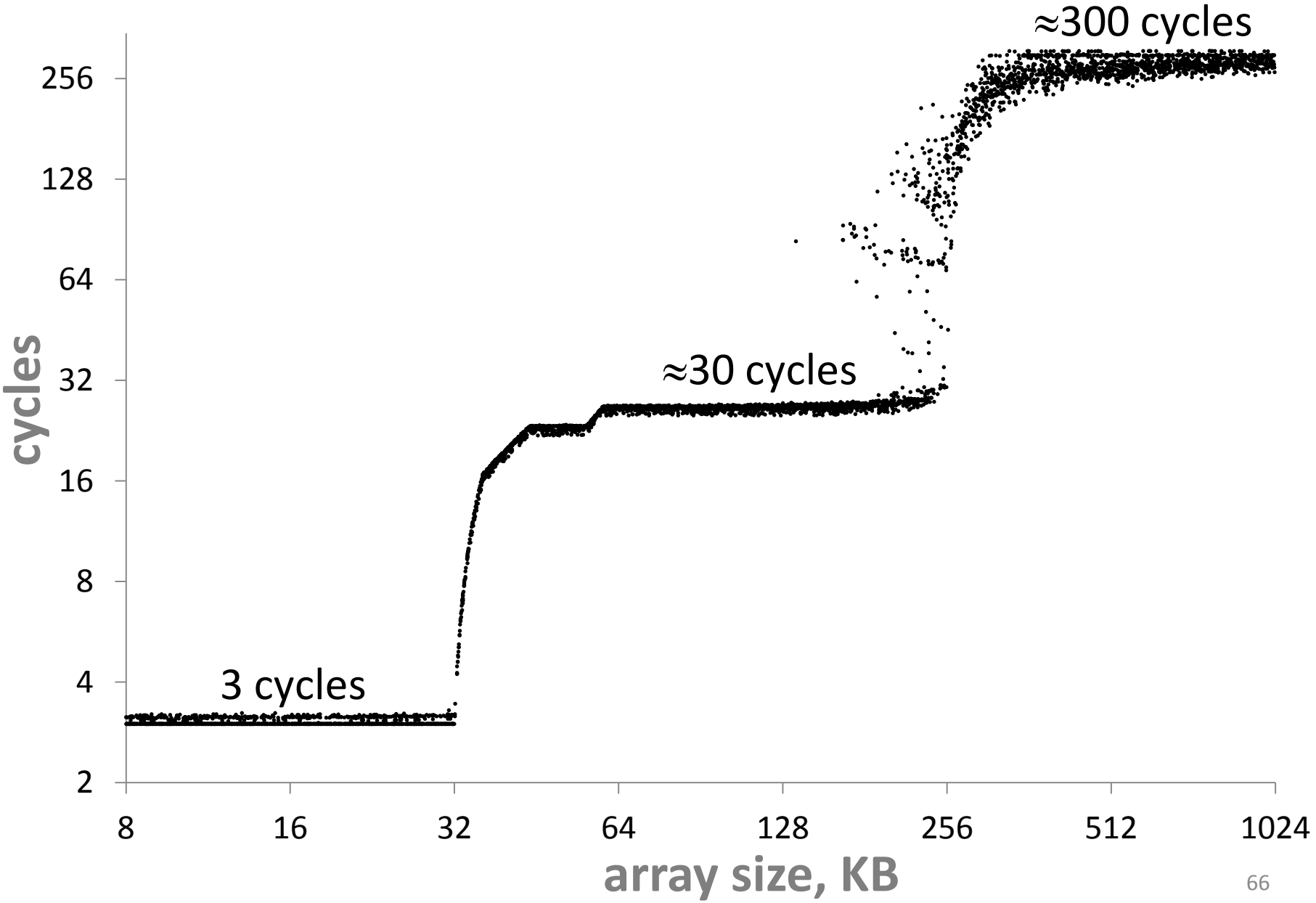
Part VIII

Memory and cache latency

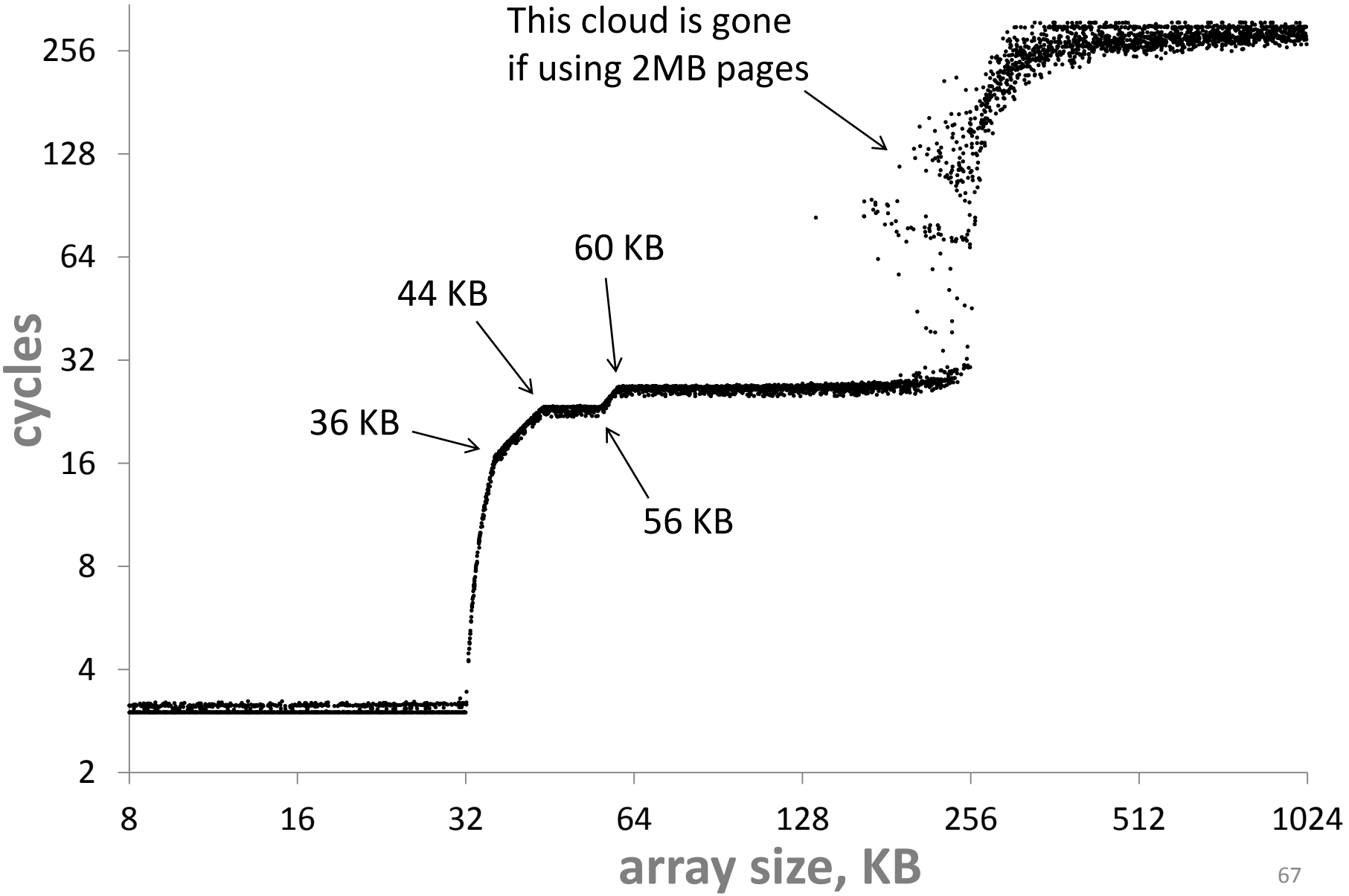
Pointer chasing benchmark

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <immintrin.h> //this time we use time stamp counter
main( )
{
    int nmin = 1<<10, nmax = 1<<17, reps = 1<<20;
    long *next = (long*)malloc( nmax*sizeof(long) );
    for( int i = 0; i < 5000; i++ )
    {
        int n = exp( log(nmin) + drand48()*(log(nmax)-log(nmin)));
        for( int i = 0; i < n; i++ ) next[i] = (i+8)%n; //stride = cache line
        float c;
        long j = lrand48()%n;
        #pragma offload target(mic) in(next:length(n))
        {
            for( int it = 0; it < n; it++ ) j = next[j]; //warm up
            __int64 t = _rdtsc();
            for( int it = 0; it < reps; it += 2 ) j = next[next[j]]; //run
            c = (float)( _rdtsc() - t ) / reps;
        }
        printf( "%.3f KB, %.3f cycles, %ld\n", n*sizeof(long)/1024.f, c, j );
    }
}
```

Classical geometric latency distribution



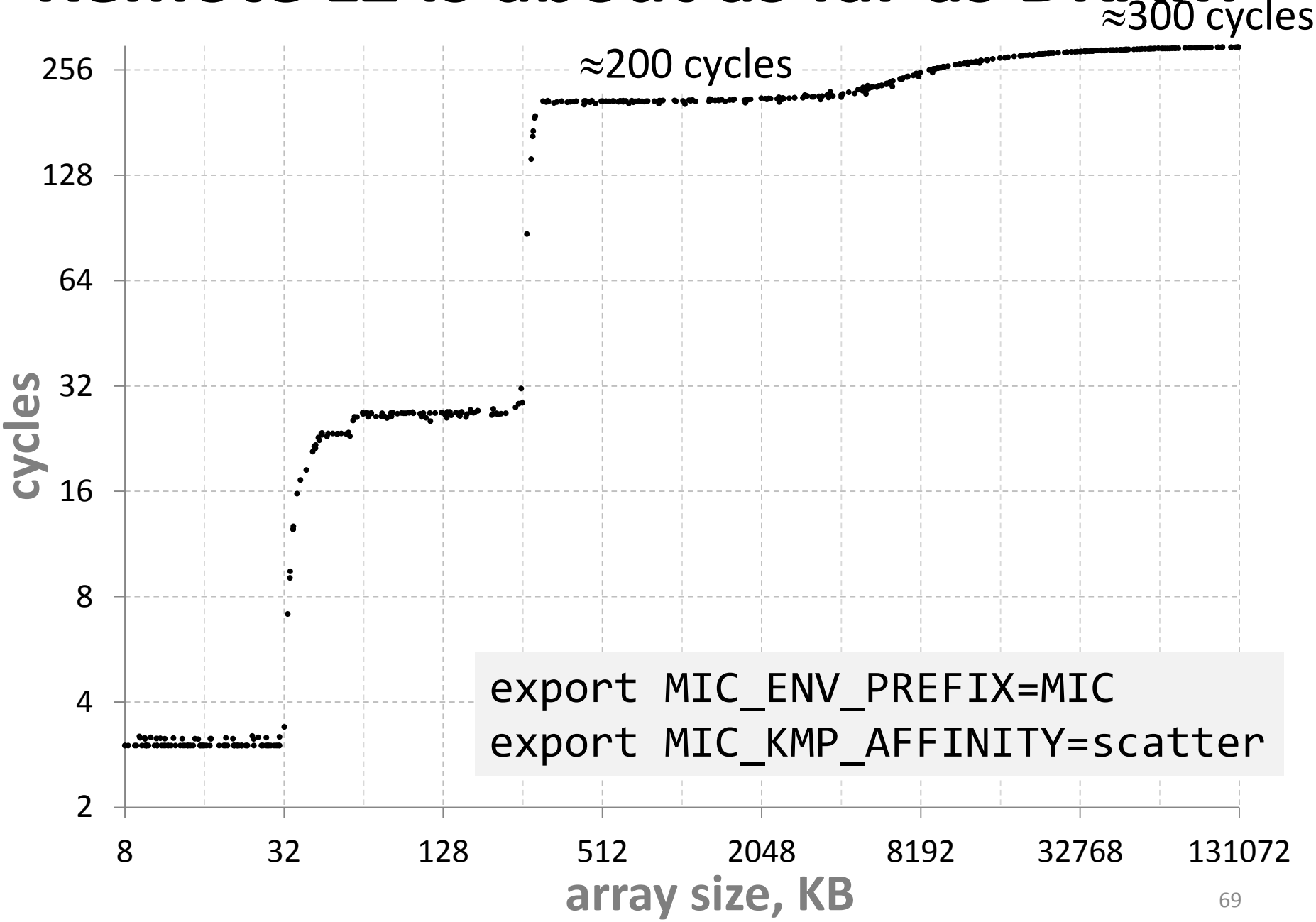
Fancy L2 structure @ multiples of 4KB



Put array into distributed L2

```
#include <omp.h>
#include <math.h>
#include <stdio.h>
#include <immintrin.h>
main( ) {
   int nmin = 1<<10, nmax = 1<<24, reps = 1<<20, nthreads = 31;
   long *next = (long*)malloc( nmax*sizeof(long) );
   for( int i = 0; i < 500; i++ )
   {
      float cycles;
      int n = exp( log(nmin) + drand48()*(log(nmax)-log(nmin)));
      long j = lrand48()%n;
      #pragma offload target(mic) in(next:length(n))
         #pragma omp parallel num_threads(nthreads)
         {
            int tid = omp_get_thread_num(), block = (n+nthreads-2)/(nthreads-1);
            if( tid > 0 )
               for( int i = (tid-1)*block; i < tid*block && i < n; i++ )
                  next[i] = (i+8)%n;//hope to get it in distributed L2
            #pragma omp barrier
            if( tid == 0 ) //thread 0 runs the bench
            {
                for( int it = 0; it < n; it++ ) j = next[j];//warm up
                __int64 t = _rdtsc();
                for( int it = 0; it < reps; it += 2 ) j = next[next[j]];//bench
                cycles = (float)( _rdtsc() - t ) / reps;
            }
      }
      printf( "%.3f KB %.3f cycles %ld\n", n*sizeof(long)/1024.f, cycles, j );
   }
}
```

Remote L2 is about as far as DRAM



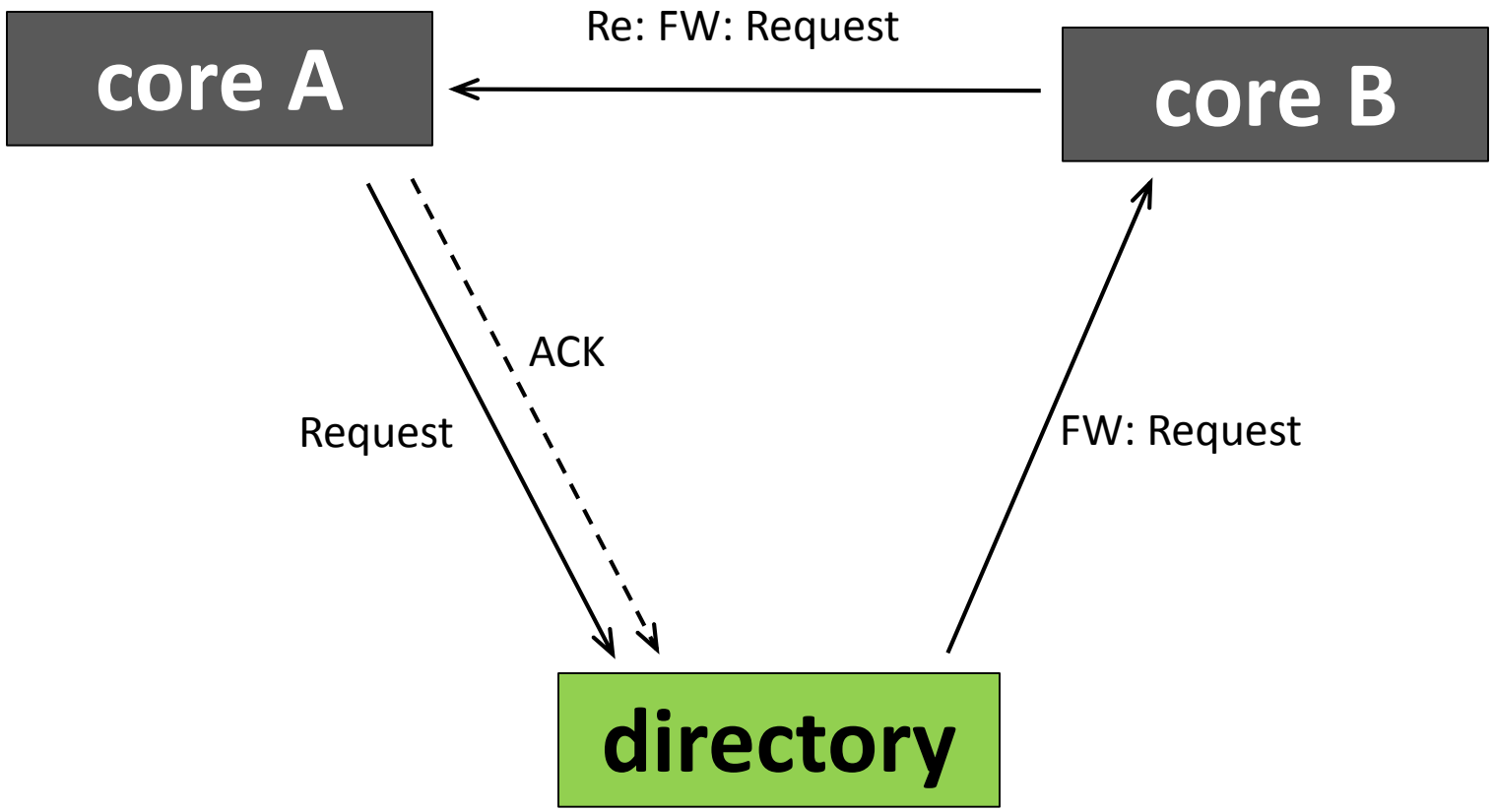
Part IX

Ping-pong latency

Ping-pong benchmark (1/2)

```
#include <stdio.h>
#include <omp.h>
#include <immintrin.h>

__declspec(target(mic)) void ping ( volatile int *p, int n )
{
    for( int i = 0; i < n; i += 2 )
    {
        *p = i+1;              // send a signal to slave
        while( *p != i+2 ); // wait for reply
    }
}

__declspec(target(mic)) void pong( volatile int *p, int n )
{
    for( int i = 0; i < n; i += 2 )
    {
        while( *p != i+1 ); // wait for a signal from master
        *p = i+2;              // reply
    }
}
```

Coherency triangle



Directory is distributed over 32 stations (DTD)

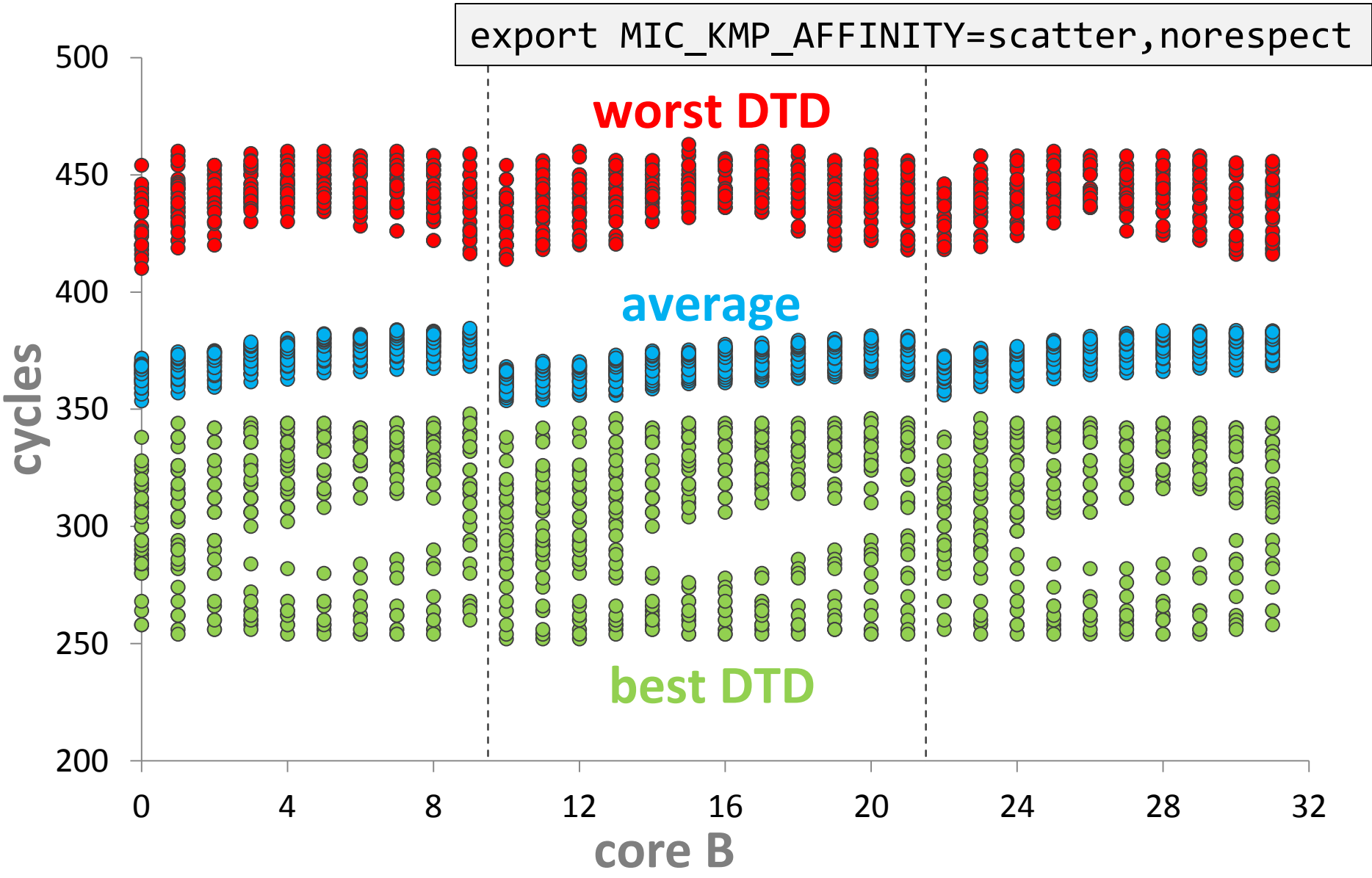
Station number is found as hashed cache line #

Q: how latency depends on the choice of station?

(2/2)

```
main() {
   const int nslots = 16*32, reps = 1000, ncores = 32;
   for( int th1, th2, j = 0; j < ncores*ncores; j++ )
   {
      if( (th1=j%ncores) == (th2=j/ncores) ) continue;
      __int64 t, tmin, tavg = 0, tmax;
      #pragma offload target(mic)
      {
         volatile int slots[nslots][16];//try many different DTDs
         for( int slot = 0; slot < nslots; slot++ )
         {
            slots[slot][0] = 0;
            #pragma omp parallel num_threads(ncores)
               for( int k = 0; k < 3; k++ )
                  if( omp_get_thread_num() == th1)
                  {
                     __int64 t0 = _rdtsc();
                     ping( &slots[slot][0], reps );
                     t = t < (t0=_rdtsc()-t0) && k ? t : t0;
                  }
                  else if( omp_get_thread_num() == th2 )
                     pong( &slots[slot][0], reps );
            tavg += t;
            tmin = slot == 0 || t < tmin ? t : tmin;
            tmax = slot == 0 || t > tmax ? t : tmax;
         }
      }
      printf( "%d %d %g %g %g\n", th1, th2, tmin/(float)reps,
                  tavg/(float)reps/nslots, tmax/(float)reps );
   }
}
```

Worst vs Best: 1.4x gap



Average across all samples: 370 cycles

Compare to 200 cycle latency to remote L2 slice

Why different?

- Each ping requires 2 directory accesses
- First for reading, second for writing:

```
for( int i = 0; i < n; i += 2 )
{
    while( *p < i+1 ); // get shared rights
    *p = i+2;          // get exclusive rights
}
```

Can do using only 1 directory access!

- Get exclusive rights when spinning

New ping-pong kernels

```
#define hint _MM_PFHINT_EX
__declspec(target(mic)) void ping( volatile int *p, int n )
{
#ifdef __MIC__
  for( int i = 0; i < n; i += 2 )
  {
    *p = i+1;
    do { _mm_vprefetch1((const void*)p, hint); } while( *p != i+2 );
  }
#endif
}

__declspec(target(mic)) void pong( volatile int *p, int n )
{
#ifdef __MIC__
  for( int i = 0; i < n; i += 2 )
  {
    do { _mm_vprefetch1((const void*)p, hint); } while( *p != i+1 );
    *p = i+2;
  }
#endif
}
```

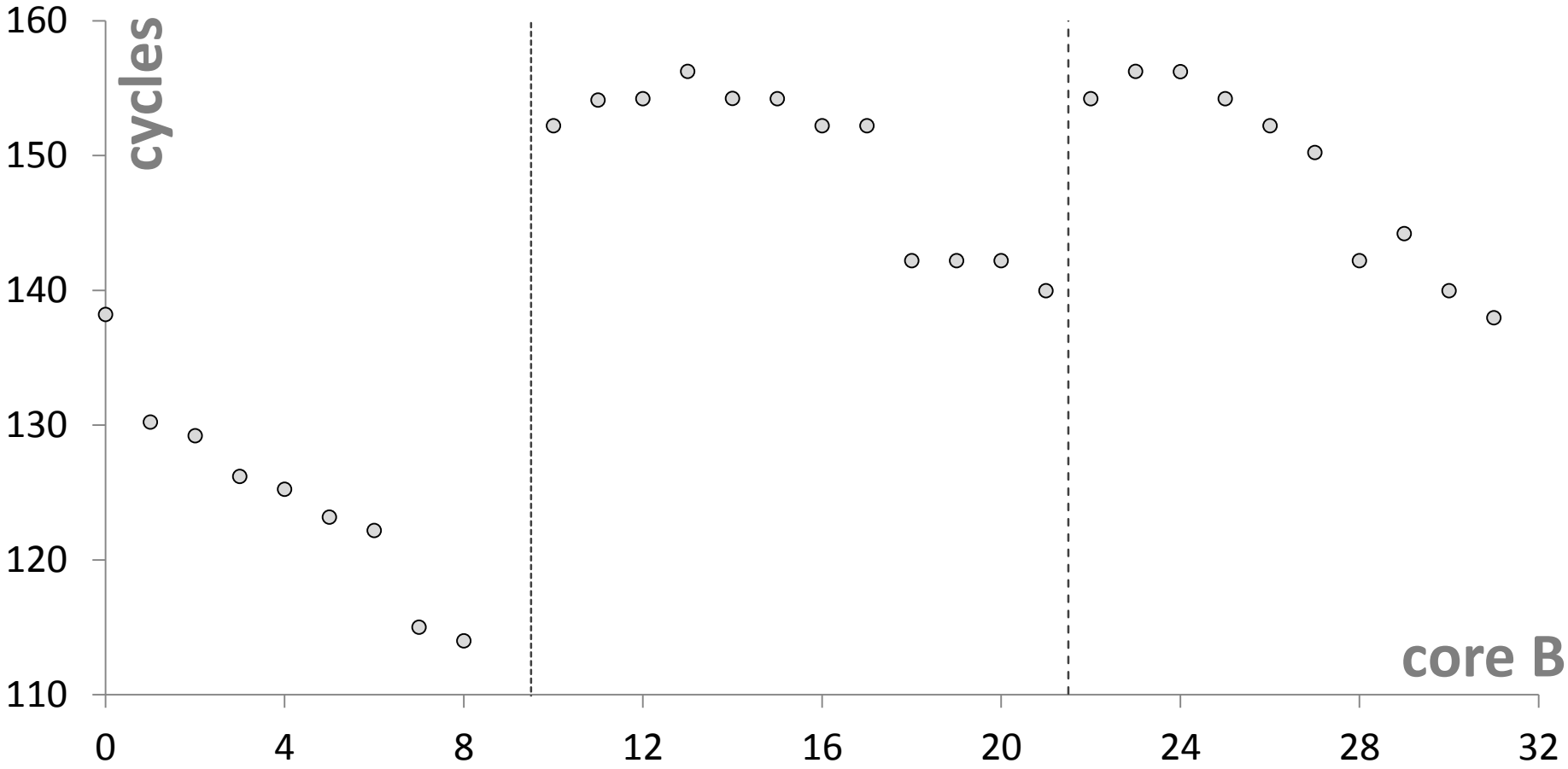
Result: uniform 2.0x-2.3x speedup

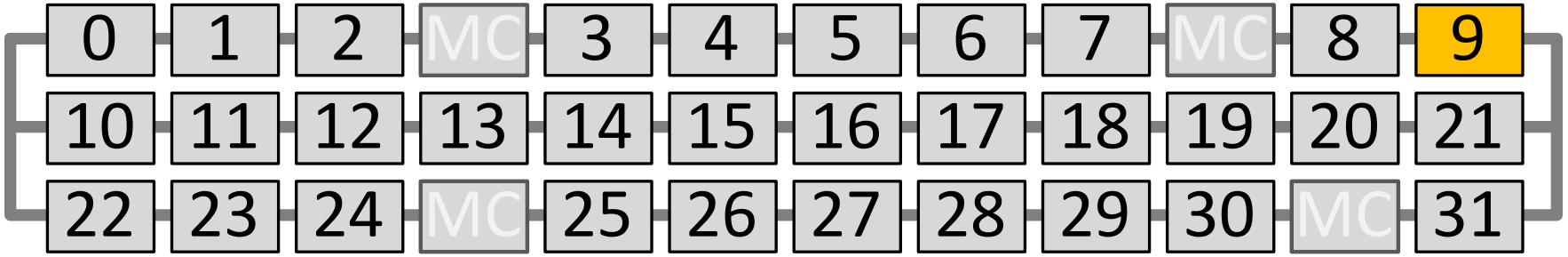
Ping-pong timing summary

	Best core choice	Worst core choice
Best DTD choice	**110**	160
Average	160	180
Worst DTD choice	190	**220**

Up to 2x speedup by optimizing both core and directory placement

Best ping time, core A is #9 (corner)





Part X

Broadcasting shared variable

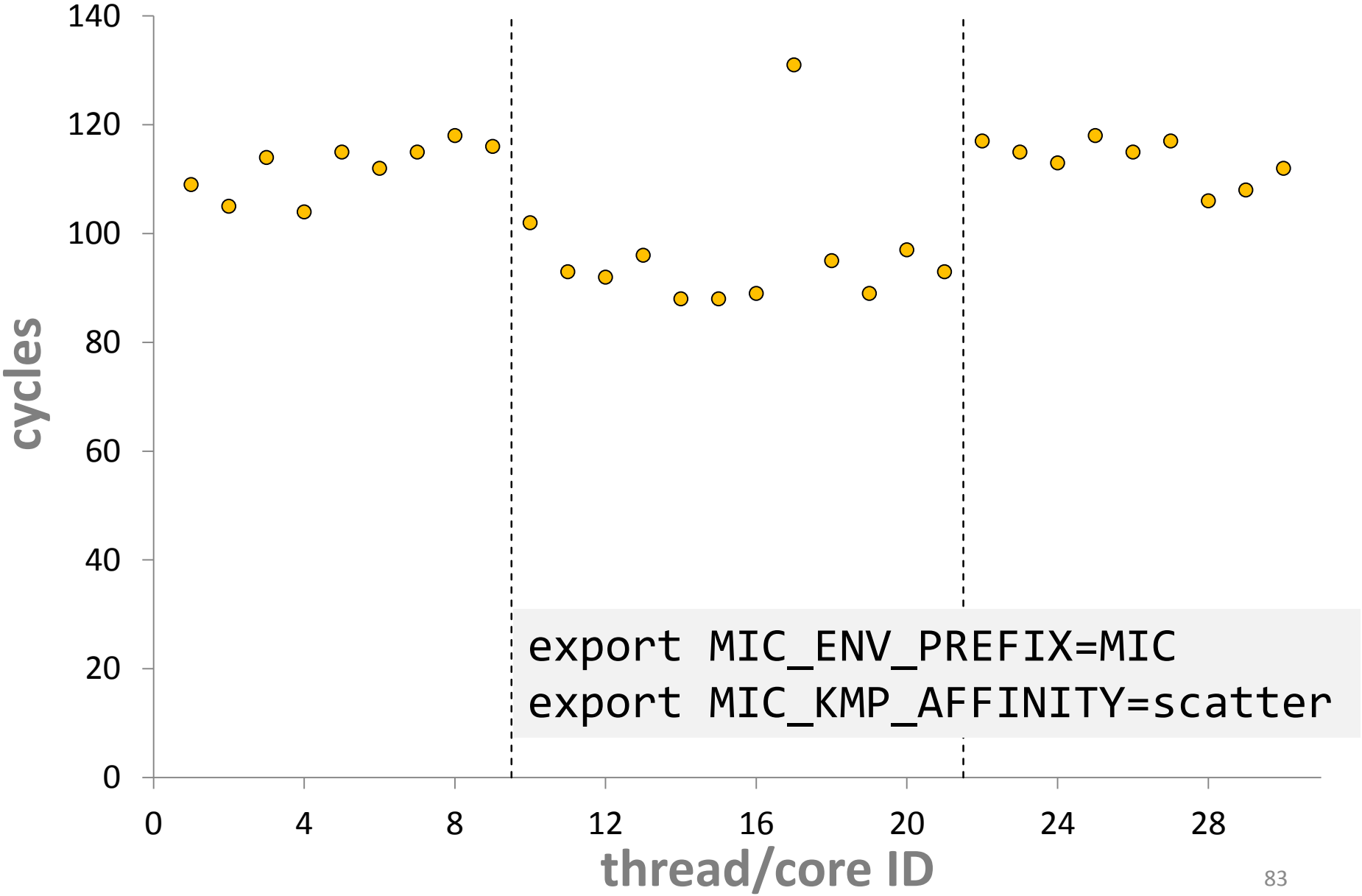
Broadcast benchmark (1/2)

```
#include <mkl.h>
#include <immintrin.h>
__declspec(target(mic)) void master( __int64 *r, volatile int &s, int n )
{
    for( int i = 0; i < n; i++, r += 2 )
    {
        for( double t = dsecnd(); dsecnd() - t < 1e-3; );//pause
        r[0] = _rdtsc(); //time before
        s = i;           //send signal
        r[1] = _rdtsc(); //time after
    }
}
__declspec(target(mic)) void slave( __int64 *r, volatile int &s, int n )
{
    for( int i = 0; i < n; i++, r += 2 )
    {
        do { r[0] = _rdtsc(); } //time before
        while( s != i );        //receive signal
        r[1] = _rdtsc();        //time after
    }
}
```

Broadcast benchmark (2/2)

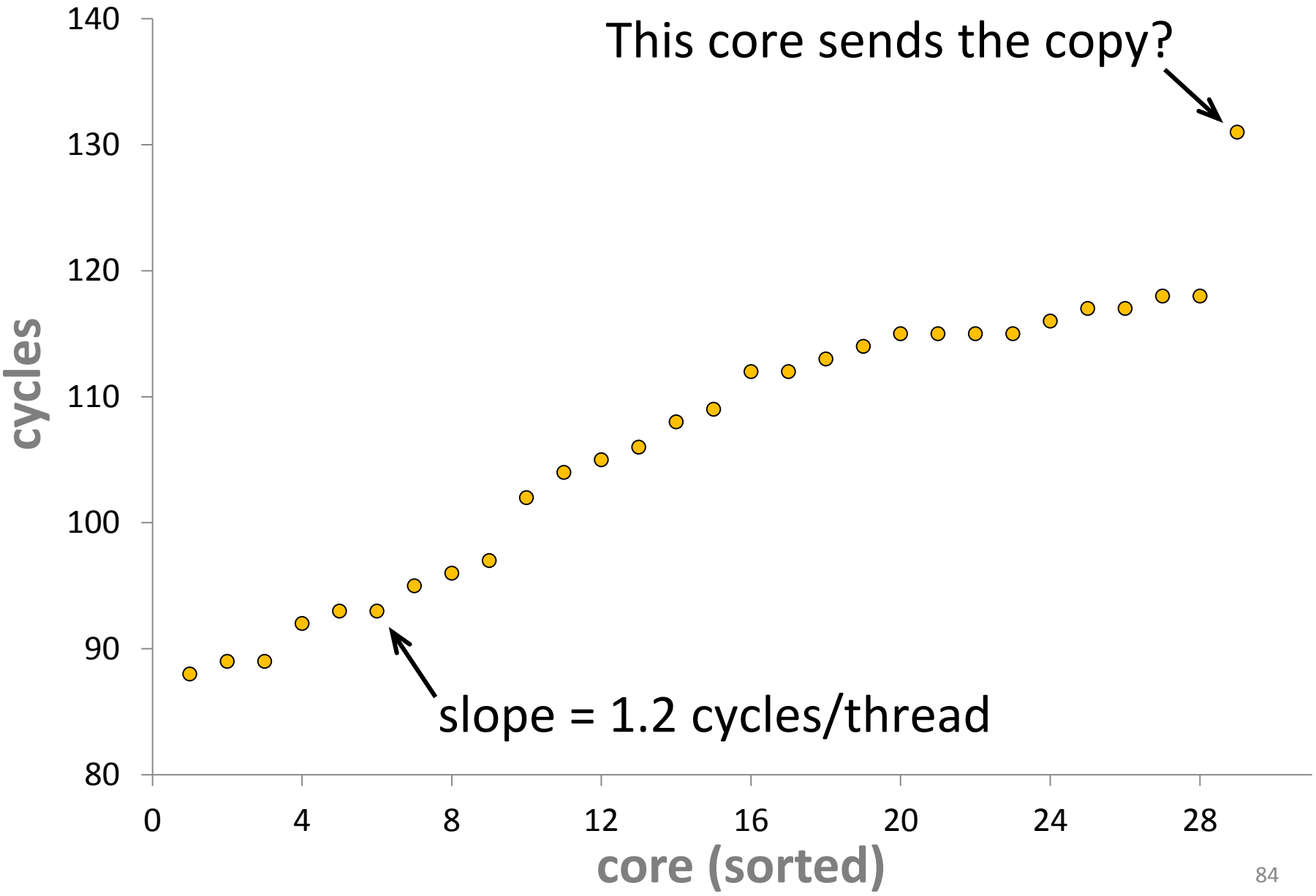
```
#include <omp.h>
#include <stdio.h>
main()
{
  const int nthreads = 31, n = 4;
  __int64 r[nthreads][n][2];
  #pragma offload target(mic)
  {
    volatile int sense = -1;
    #pragma omp parallel num_threads(nthreads)
    {
      int tid = omp_get_thread_num();
      if( tid == 0 ) master( &r[tid][0][0], sense, n );//one master
      else           slave( &r[tid][0][0], sense, n );//many slaves
    }
  }
  for( int t = 0; t < nthreads; t++ )
  {
    for( int i = 0; i < n; i++ ) printf( "%5ld", r[t][i][0]-r[0][i][0] );
    for( int i = 0; i < n; i++ ) printf( "%5ld", r[t][i][1]-r[t][i][0] );
    printf( "\n" );
  }
}
```

Time until invalidation request is received

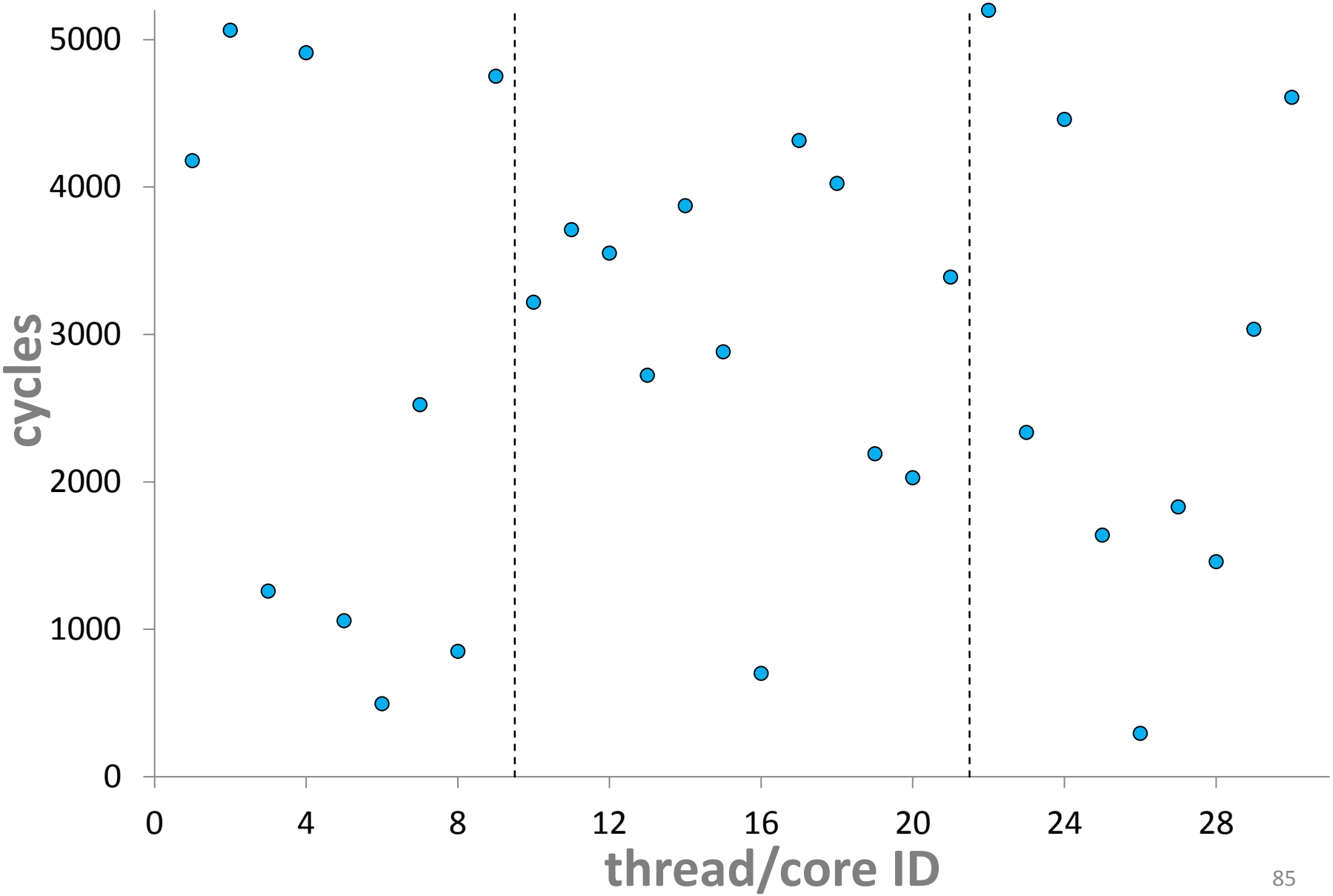



```
export MIC_ENV_PREFIX=MIC
export MIC_KMP_AFFINITY=scatter
```

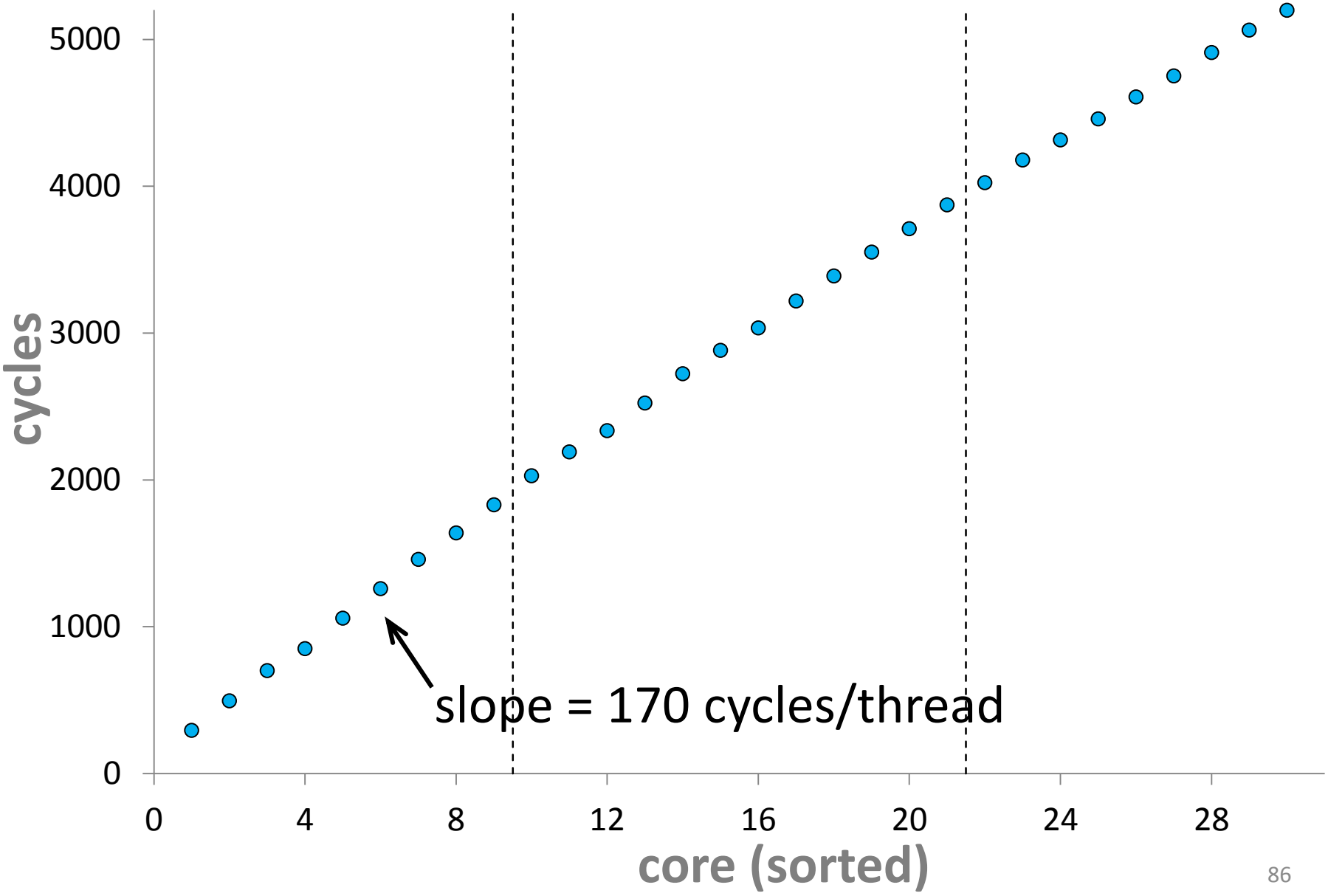
Time until invalidation, sorted



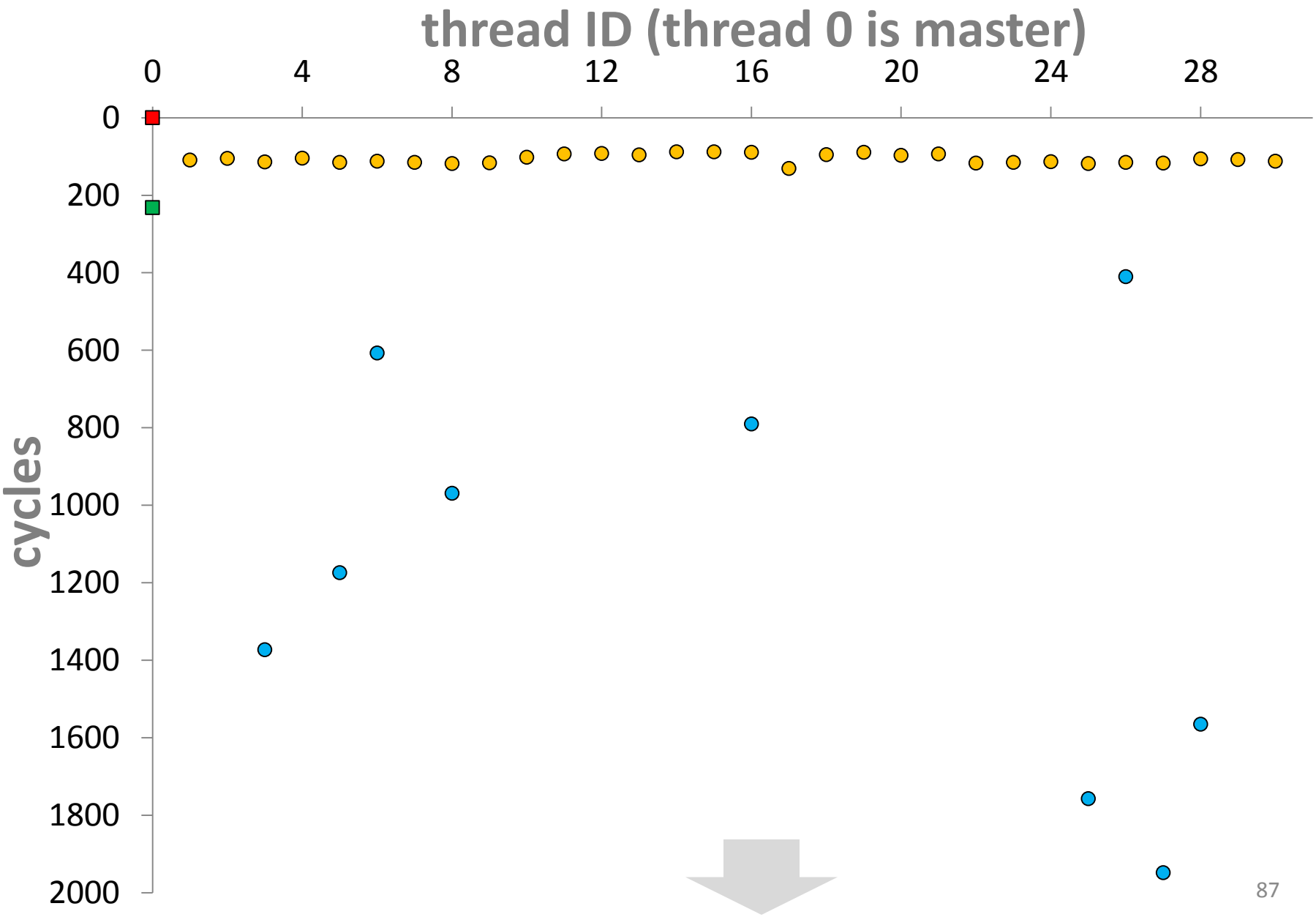
Delay until updated copy is retrieved



Delay until the update, sorted



Master and slave timings together



Programming MIC is fun!

Thanks to Kostadin Ilov for technical support; Roman Dubtsov for help with MKL; Kevin Davis for help on offload overhead; James Cownie for help on OpenMP barrier; Timothy Prince for affinity settings; Charles Congdon for shared virtual memory explanation; Romain Dolbeau for Intel memory consistency model