Computer Architecture : A Programmer's Perspective

Abhishek Somani, Debdeep Mukhopadhyay

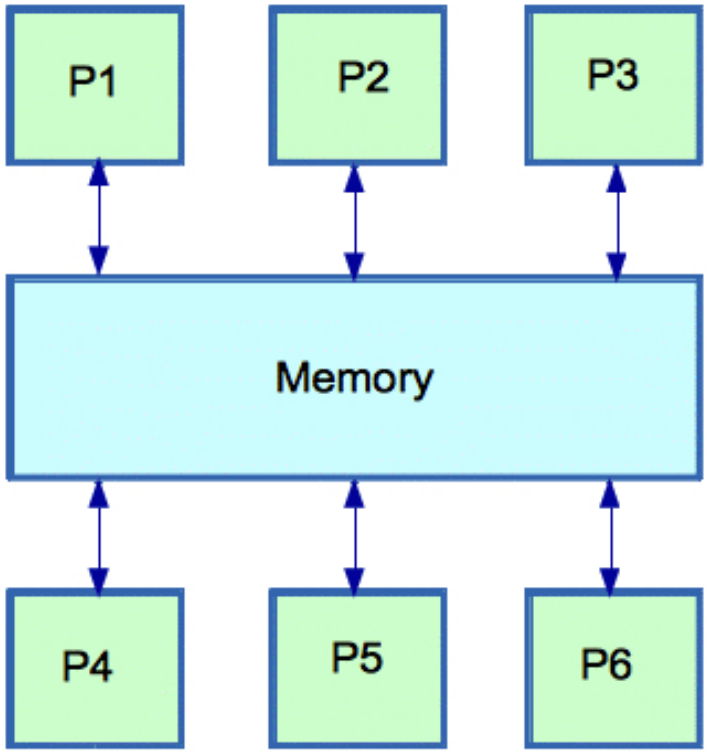
Mentor Graphics, IIT Kharagpur

September 9, 2016

Overview

Outline

Communication Cost



- Communication cost in PRAM model : 1 unit per access
- Does it really hold in practice even within a single processor ?

Spot the difference

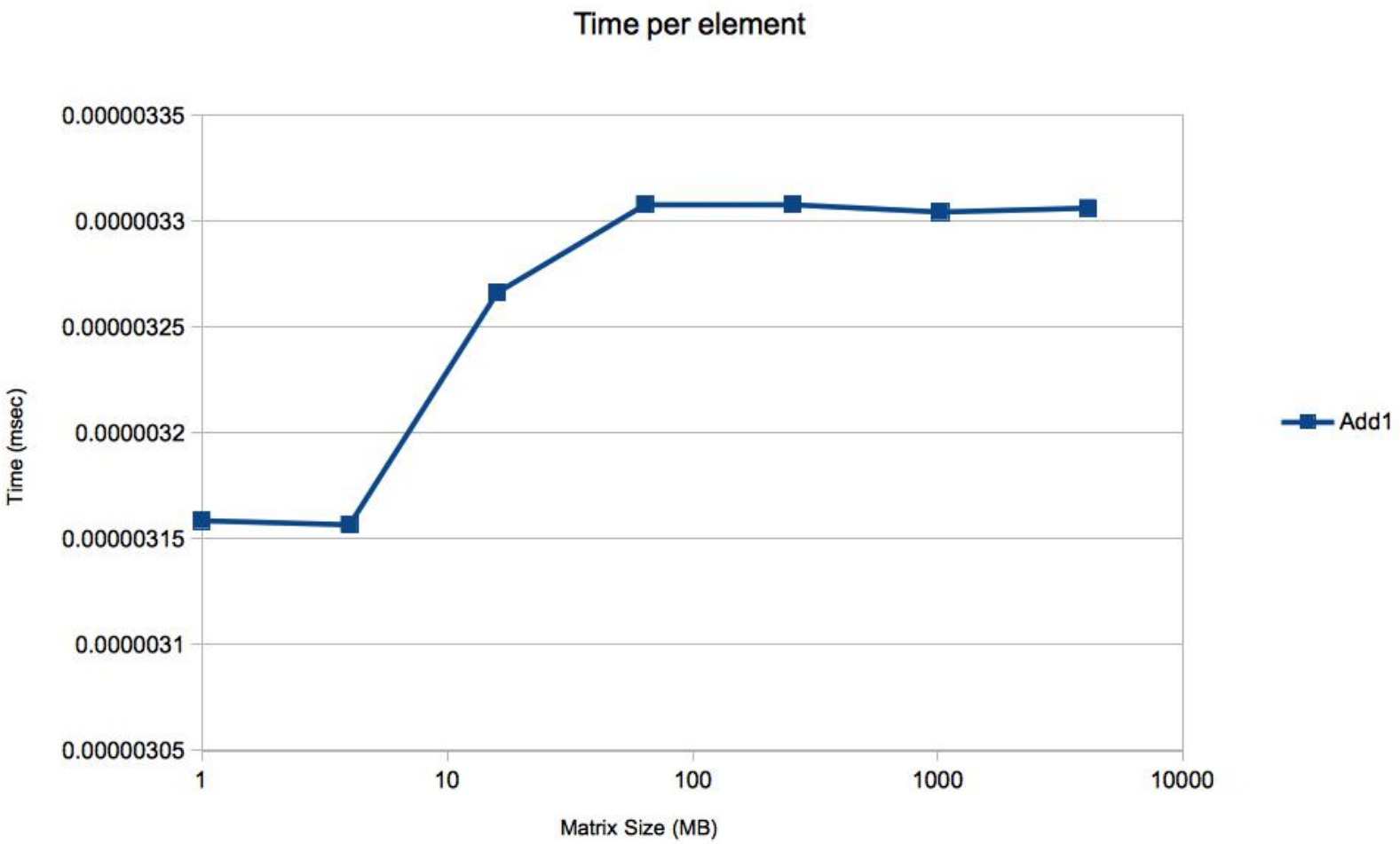
Add1

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
      result += A[n*i + j];
```

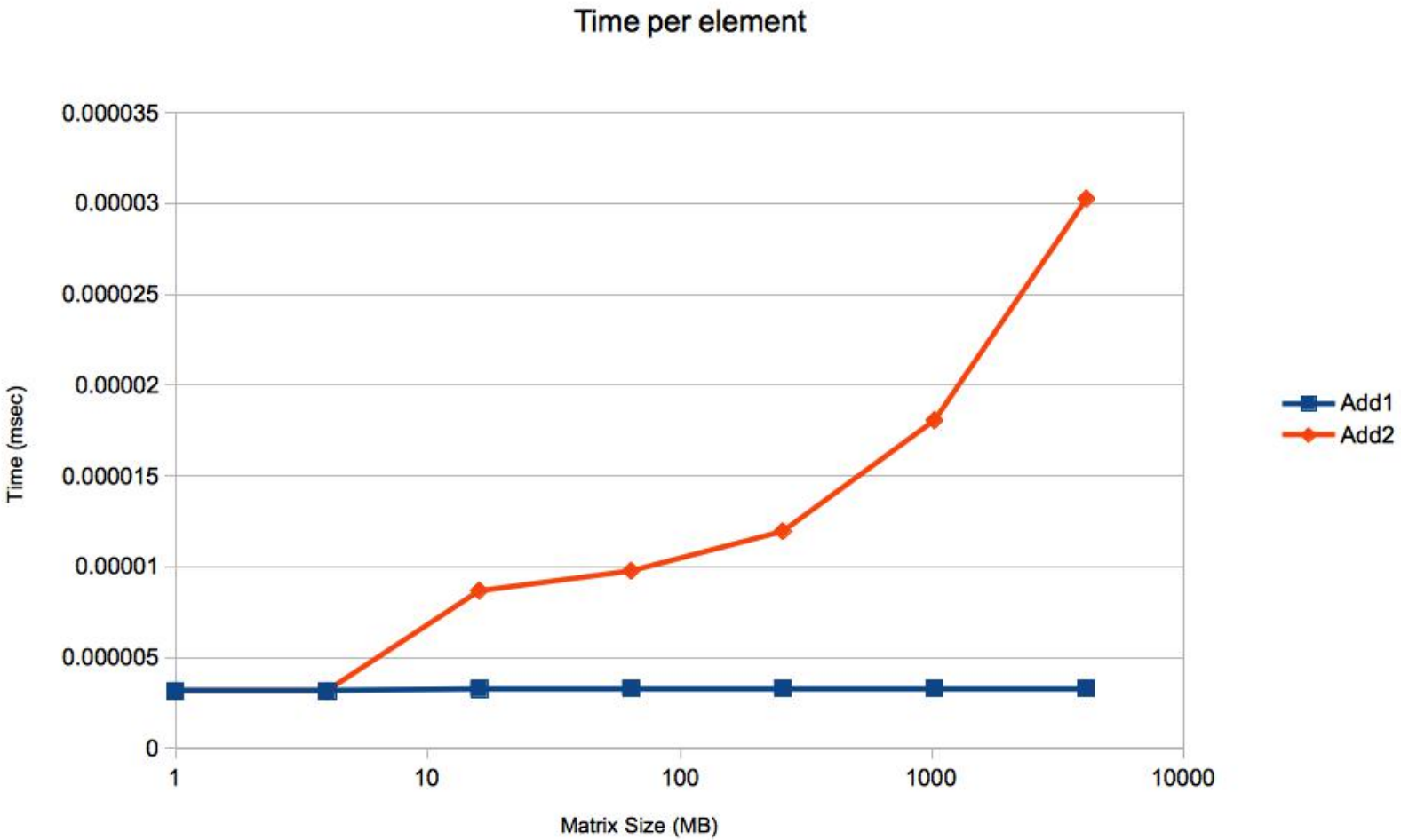
Add2

```
for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j)
      result += A[i + n*j];
```

Time Performance



Time Performance ...



Outline

Simple Addition

```
int add(const int numElements,
        double * arr) {
    double sum = 0.0;
    for(int i = 0; i < numElements; i += 1)
        sum += arr[i];
    return sum;
}
```

```
int stride2Add(const int numElements,
               double * arr) {
    double sum = 0.0;
    for(int i = 0; i < 2*numElements; i += 2)
        sum += arr[i];
    return sum;
}
```

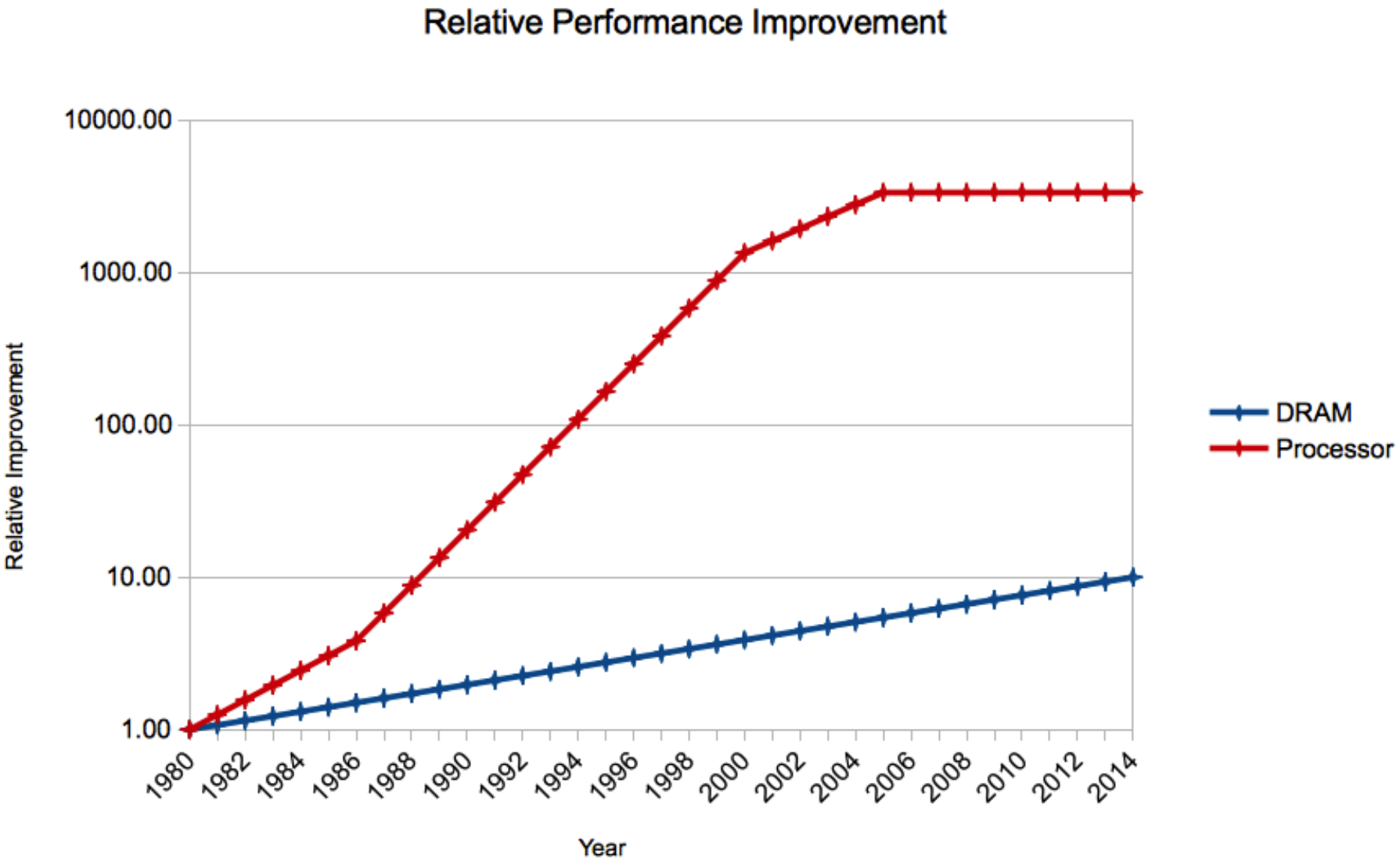
Strided Addition

```
int stridedAdd(const int numElements,
               const int stride,
               double * arr) {
    double sum = 0.0;
    const int lastElement = numElements * stride;
    for(int i = 0; i < lastElement; i += stride)
        sum += arr[i];
    return sum;
}
```

$$\text{Throughput} = \frac{\text{Number of Elements}}{\text{Time}} = \frac{\text{Number of Elements}}{\frac{\text{Clock cycles}}{\text{Clock Speed}}}$$

- For a fixed number of elements, how would stride impact throughput ?
- For a fixed stride, how would the number of elements impact throughput ?

Performance Gap between Single Processor and DRAM



Intel Core i7

- Clock Rate : 3.2 GHz
- Number of cores : 4
- Data Memory references per core per clock cycle : 2 64-bit references
- Peak Instruction Memory references per core per clock cycle : 1 128-bit reference
- Peak Memory bandwidth : 25.6 billion 64-bit data references + 12.8 billion 128-bit instruction references = 409.6 GB/s
- DRAM Peak bandwidth : 25 GB/s
- How is this gap managed ?

Memory Hierarchy

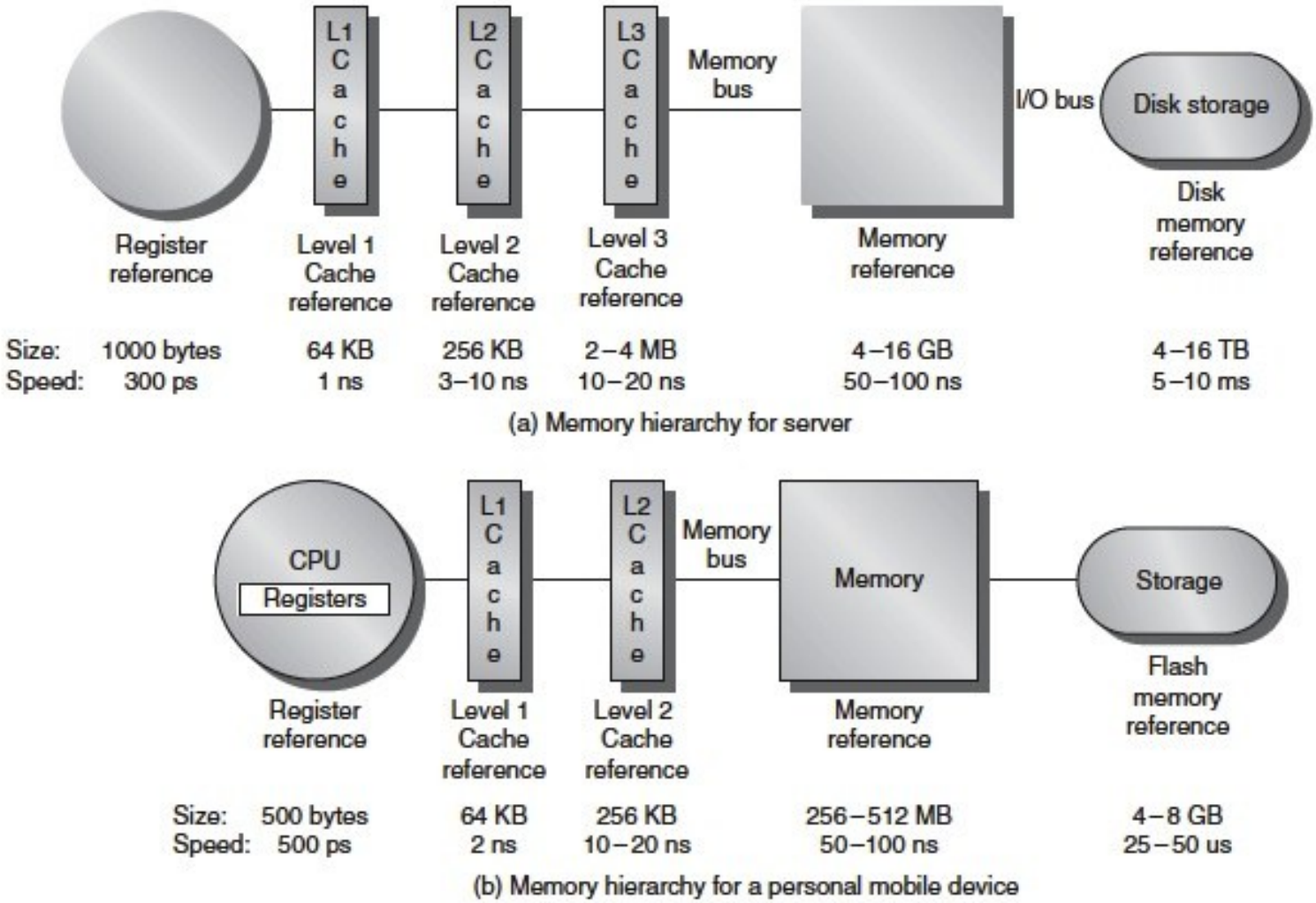


Figure : Courtesy of John L. Hennessey & David A. Patterson

Memory Hierarchy in Intel Sandybridge

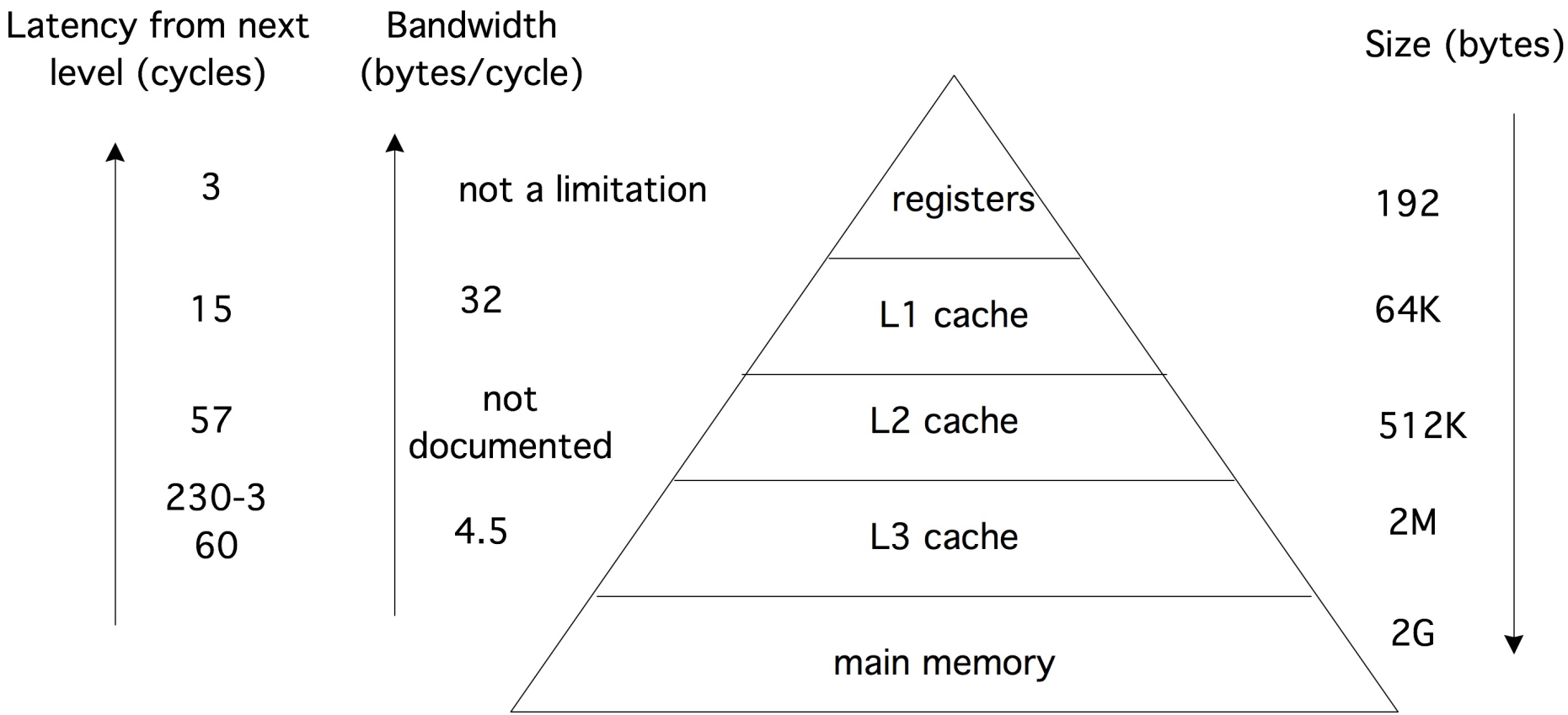
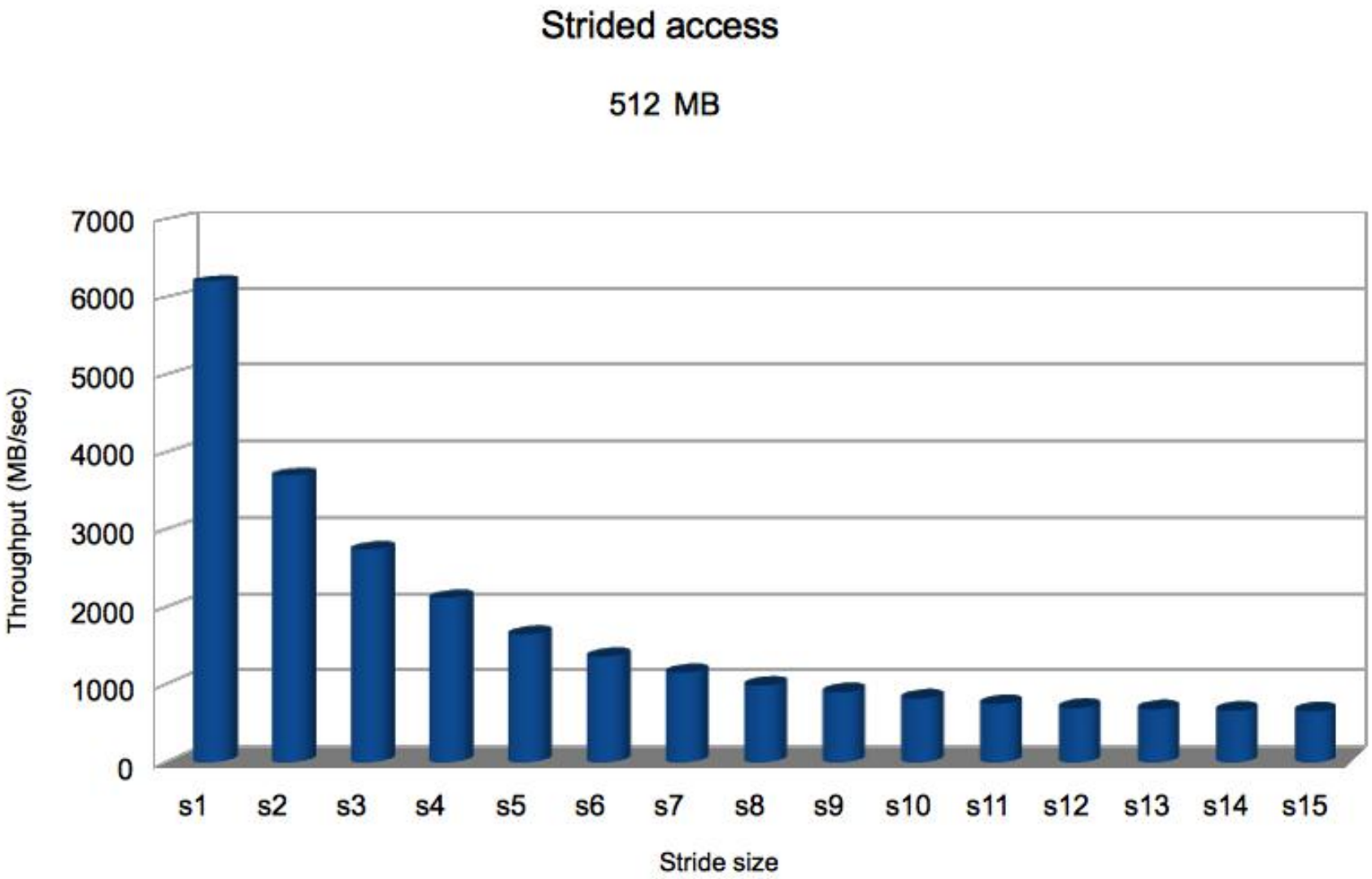


Figure : Courtesy of Victor Eijkhout

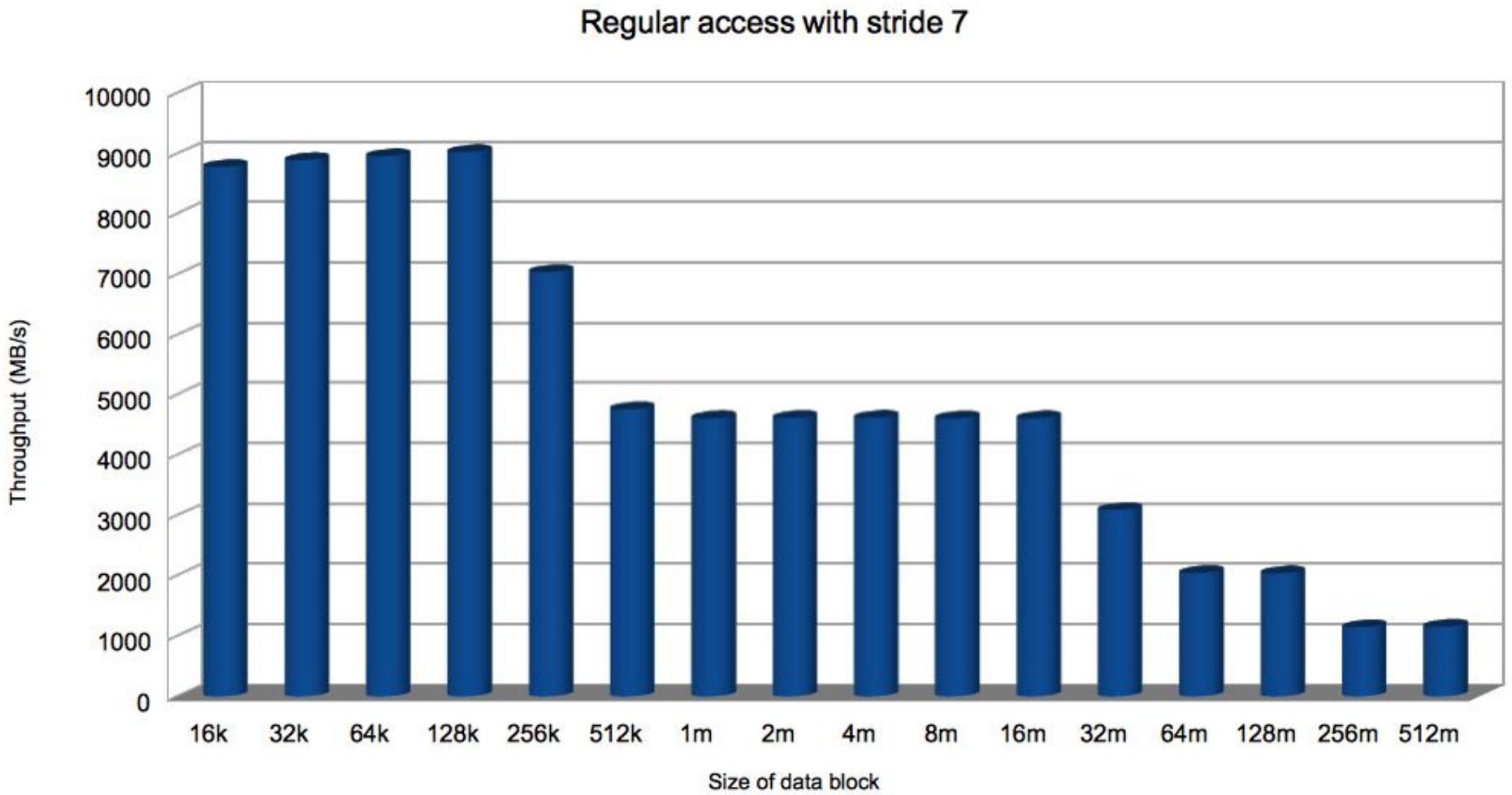
Details of experimental Machine

- Intel Xeon CPU E5-2697 v2
- Clock speed : 2.70GHz
- Number of processor cores : 24
- Caches :
  - L1D : 32 KB, L1I : 32 KB
  - Unified L2 : 256 KB
  - Unified L3 : 30720 KB
  - Line size : 64 Bytes
- 10.5.18.101, 10.5.18.102, 10.5.18.103, 10.5.18.104

Impact of stride : Spatial Locality



Impact of size : Temporal Locality



Outline

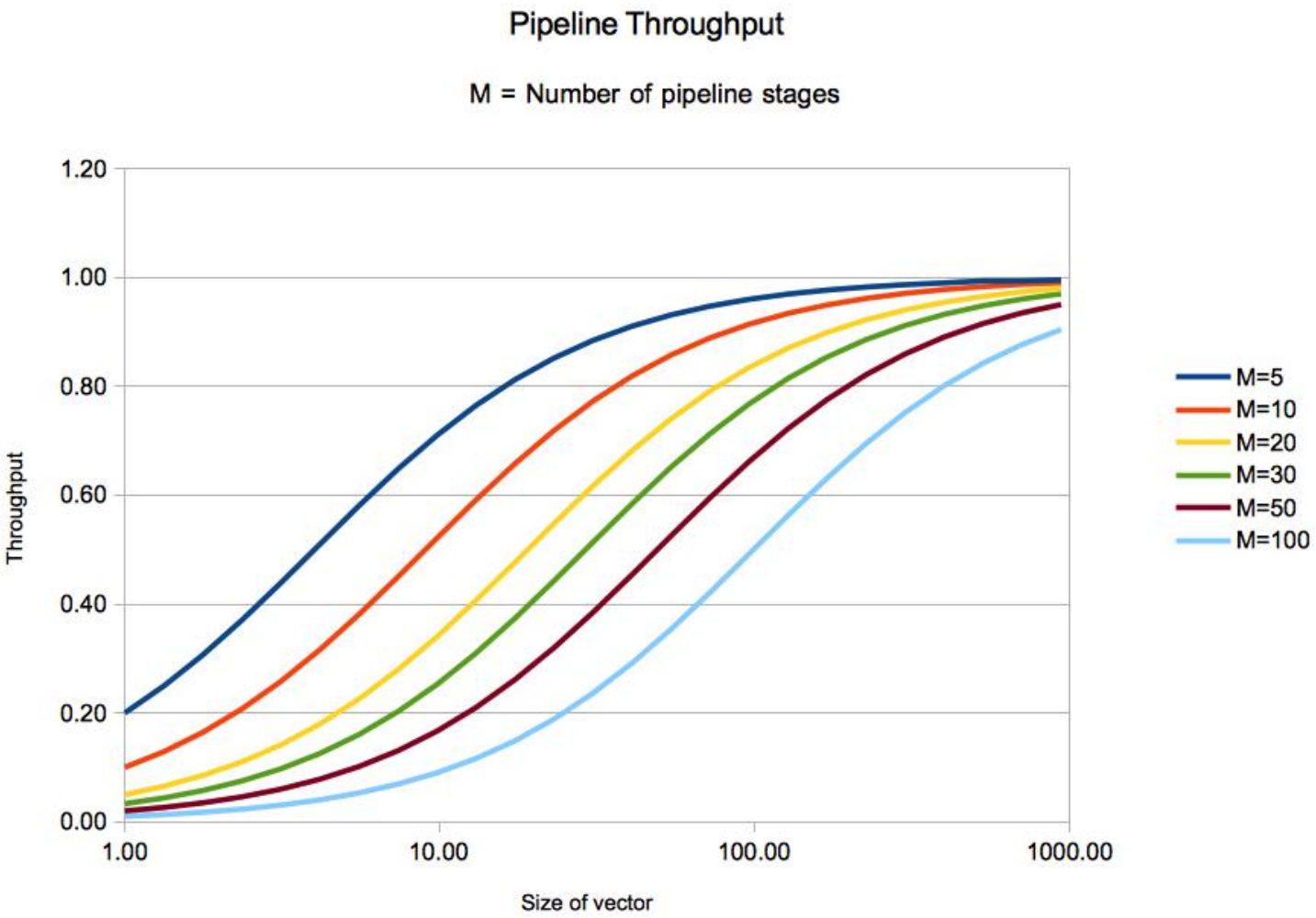
Pipelining

- Factory Assembly Line analogy
- Fetch - Decode - Execute pipeline
- Improved throughput (instructions completed per unit time)
- Latency during initial "wind-up" phase
- Typical microprocessors have overall 10 - 35 pipeline stages
- Can the number of pipeline stages be increased indefinitely ?

Pipelining Stages

- Pipeline depth : M
- Number of independent, subsequent operations : N
- Sequential time, $T_{seq} = MN$
- Pipelined time, $T_{pipe} = M + N - 1$
- Pipeline speedup, $\alpha = \frac{T_{seq}}{T_{pipe}} = \frac{MN}{M+N-1} = \frac{M}{1+\frac{M-1}{N}}$
- Pipeline throughput, $p = \frac{N}{T_{pipe}} = \frac{N}{M+N-1} = \frac{1}{1+\frac{M-1}{N}}$

Pipelining Stages...



Pipeline Magic

Scale1

```
for (int i = 0; i < n; ++i)
  A[i] = scale * A[i];
```

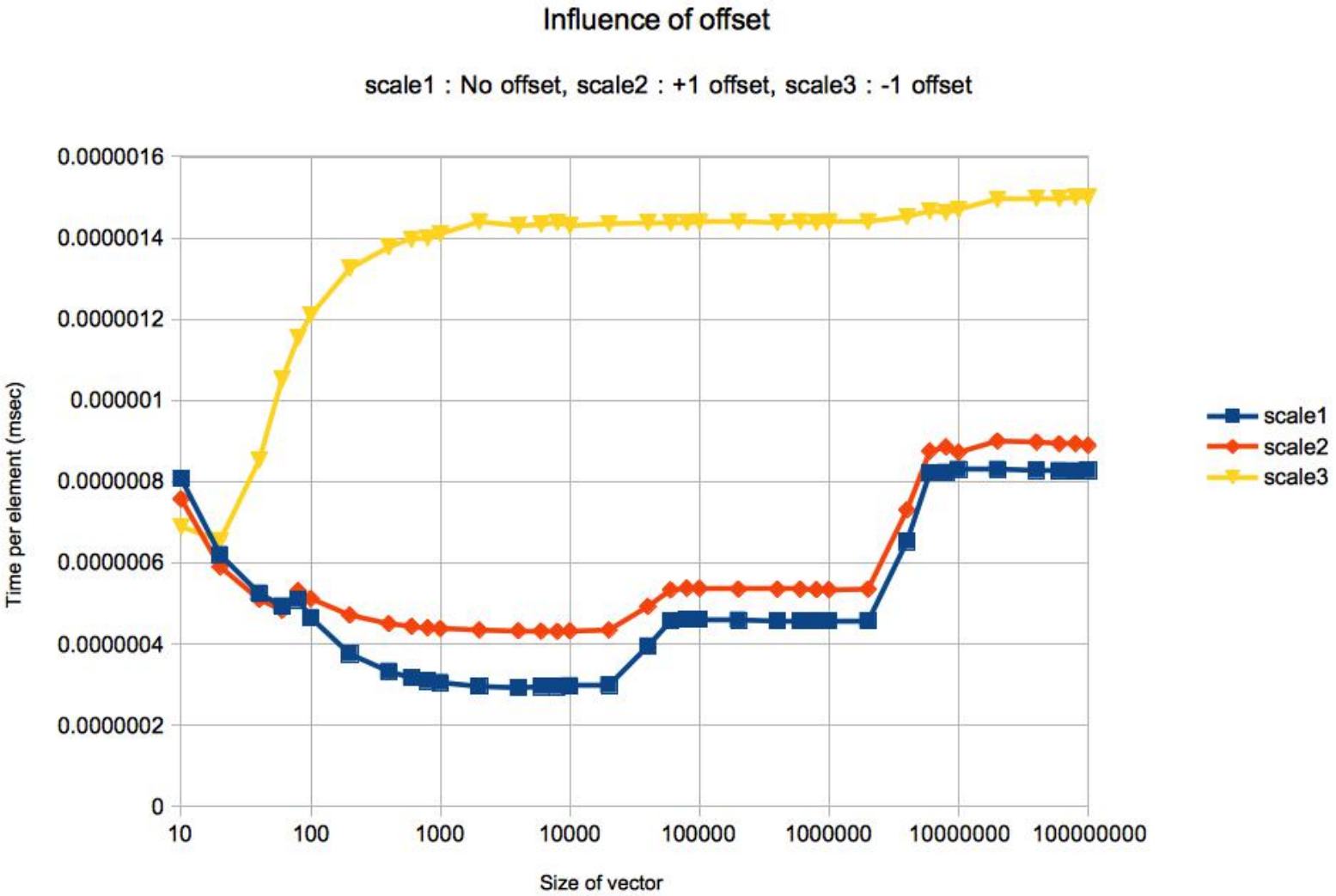
Scale2

```
for (int i = 0; i < n-1; ++i)
  A[i] = scale * A[i+1];
```

Scale3

```
for (int i = 1; i < n; ++i)
  A[i] = scale * A[i-1];
```

Pipeline Magic...



Software Pipelining

- Pipelining can be effectively used for scale1 and scale2, but not scale3
  - scale1 : Independent loop iterations
  - scale2 : False dependency between loop iterations
  - scale3 : Real dependency between loop iterations
- Software pipelining
  - Interleaving of instructions in different loop iterations
  - Usually done by the compiler
- Number of lines in assembly code generated by gcc under -O3 optimization
  - scale1 : 63
  - scale2 : 73
  - scale3 : 18

Superscalarity

- Direct instruction-level parallelism
- Concurrent fetch and decode of multiple instructions
- Multiple floating-point pipelines can run in parallel
- Out-of-order execution and compiler optimization needed to properly exploit superscalarity
- Hard for compiler generated code to achieve more than 2-3 instructions per cycle
- Modern microprocessors are up to 6-way superscalar
- Very high performance may require assembly level programming

SIMD

- Single Instruction Multiple Data
- Wide registers - up to 512 bits
  - 16 integers
  - 16 floats
  - 8 doubles
- Intel : SSE, AMD : 3dNow!, etc.
- Advanced optimization options in recent compilers can generate relevant code to utilize SIMD
- *Compiler intrinsics* can be used to manually write SIMD code

Outline

Outline

Why is matrix multiplication important?

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	King Abdullah University of Science and Technology Saudi Arabia	**Shaheen II** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	196,608	5,537.0	7,235.2	2,834
8	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
9	Forschungszentrum Juelich (FZJ) Germany	**JUQUEEN** - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301

Matrix Representation

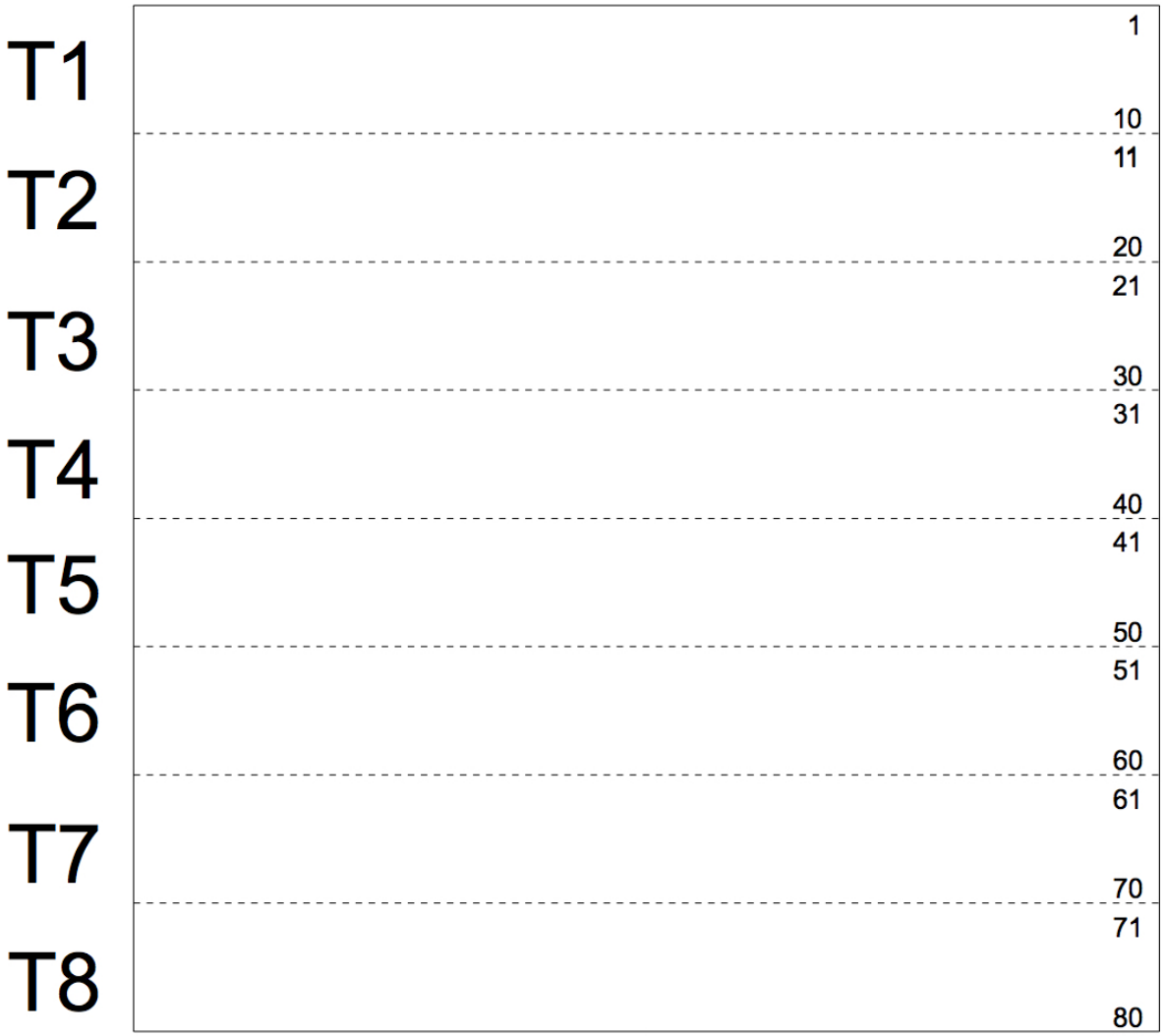
- Single array contains entire matrix
- Matrix arranged in row-major format
- **m**×**n** matrix contains **m** rows and **n** columns
- $A(i, j)$ is the matrix entry at i^{th} row and j^{th} column of matrix **A**
- It is the $(i \times n + j)^{th}$ entry in the matrix array

Triple nested loop

```
void square_dgemm (int n, double* A, double* B, double* C)
{
    for (int i = 0; i < n; ++i)
    {
        const int iOffset = i*n;
        for (int j = 0; j < n; ++j)
        {
            double cij = 0.0;
            for( int k = 0; k < n; k++ )
                cij += A[iOffset+k] * B[k*n+j];
            C[iOffset+j] += cij;
        }
    }
}
```

- Total number of multiplications : n^3

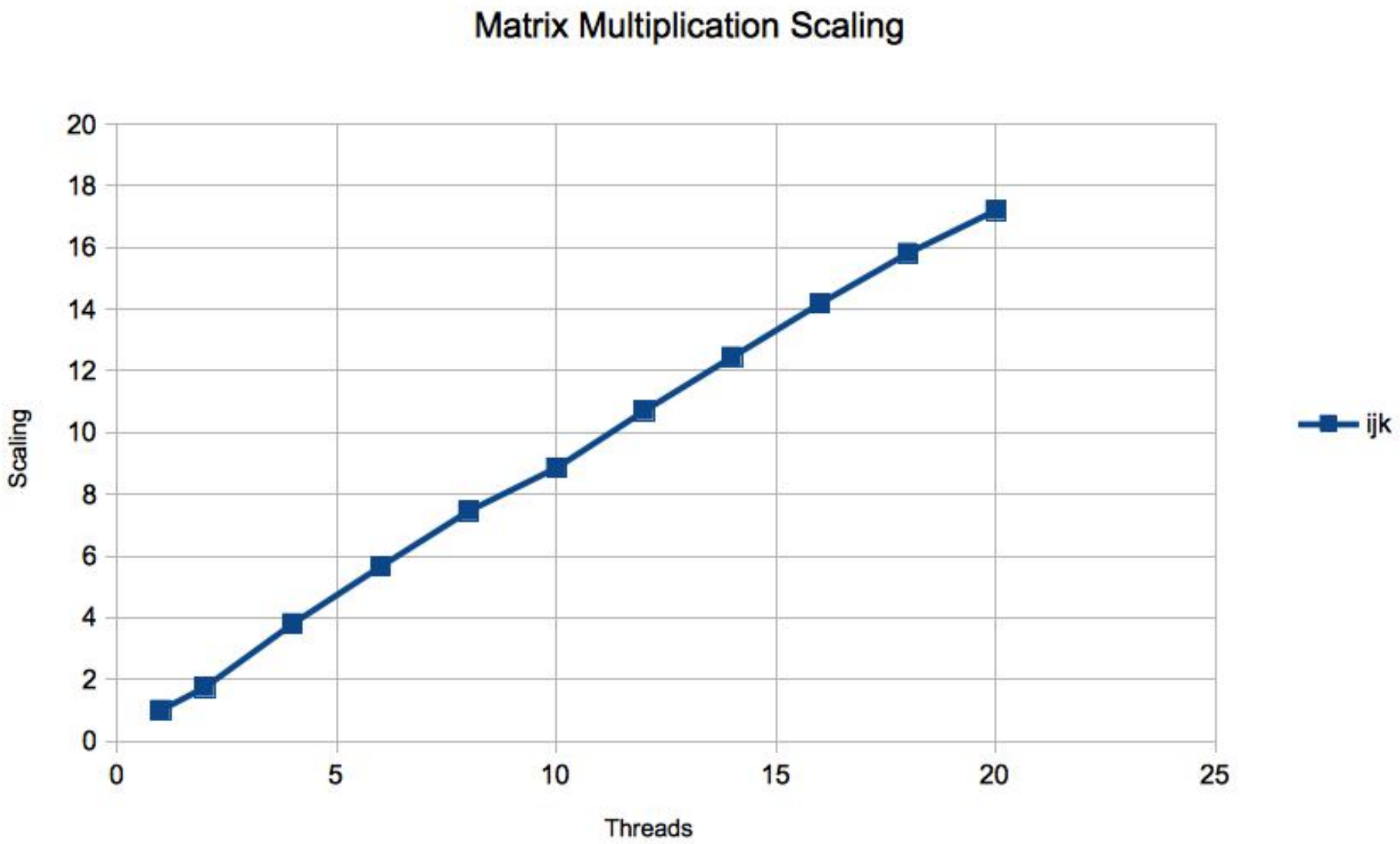
Row-based data decomposition in matrix C



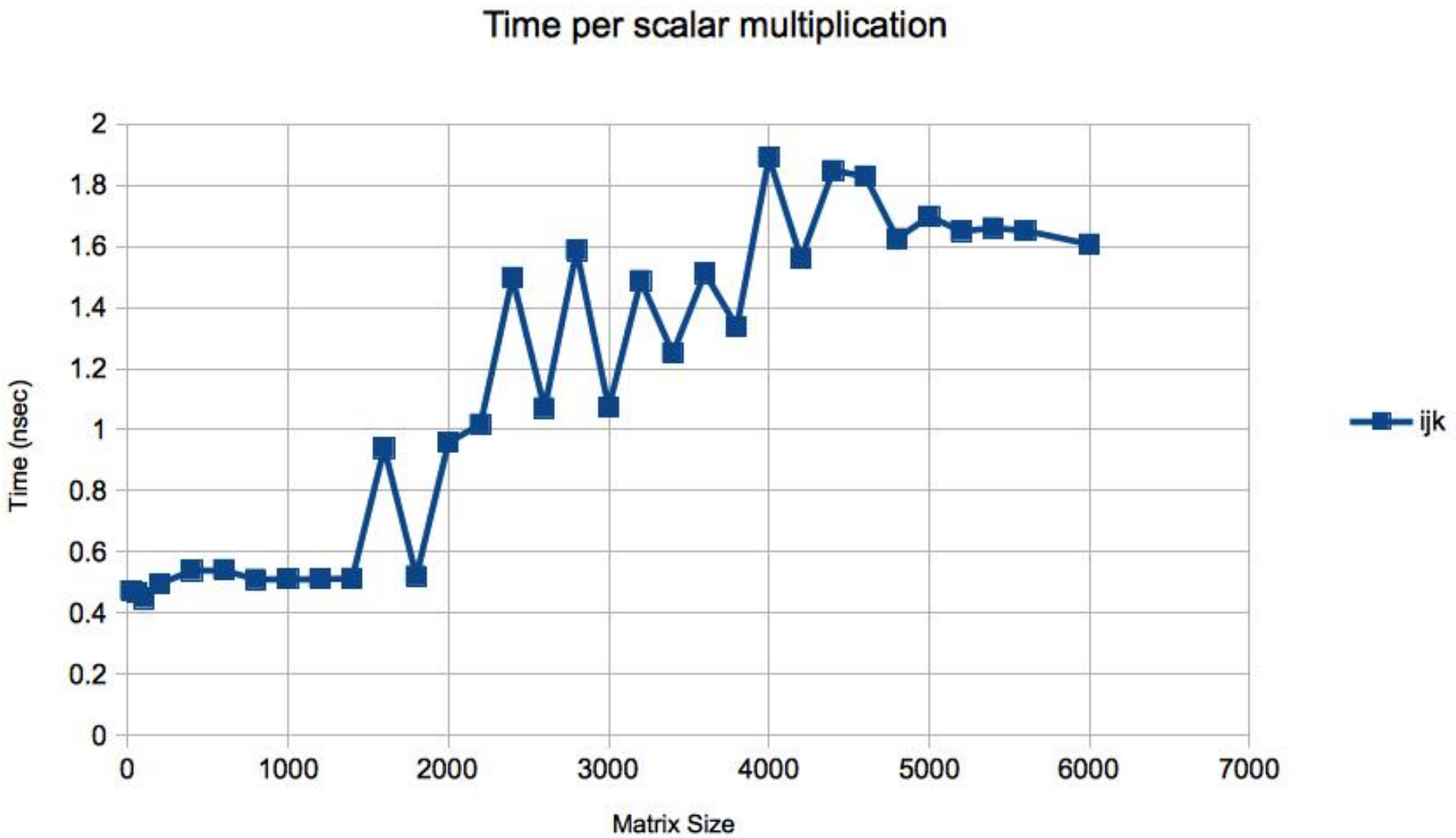
Parallel Multiply

```
void square_dgemm (int n, double* A, double* B, double* C)
{
#pragma omp parallel for schedule(static)
    for (int i = 0; i < n; ++i)
    {
        const int iOffset = i*n;
        for (int j = 0; j < n; ++j)
        {
            double cij = 0.0;
            for( int k = 0; k < n; k++ )
                cij += A[iOffset+k] * B[k*n+j];
            C[iOffset+j] += cij;
        }
    }
}
```

(Almost) Perfect Scaling for matrix of size 6000 $\times$ 6000



How good is the serial performance?

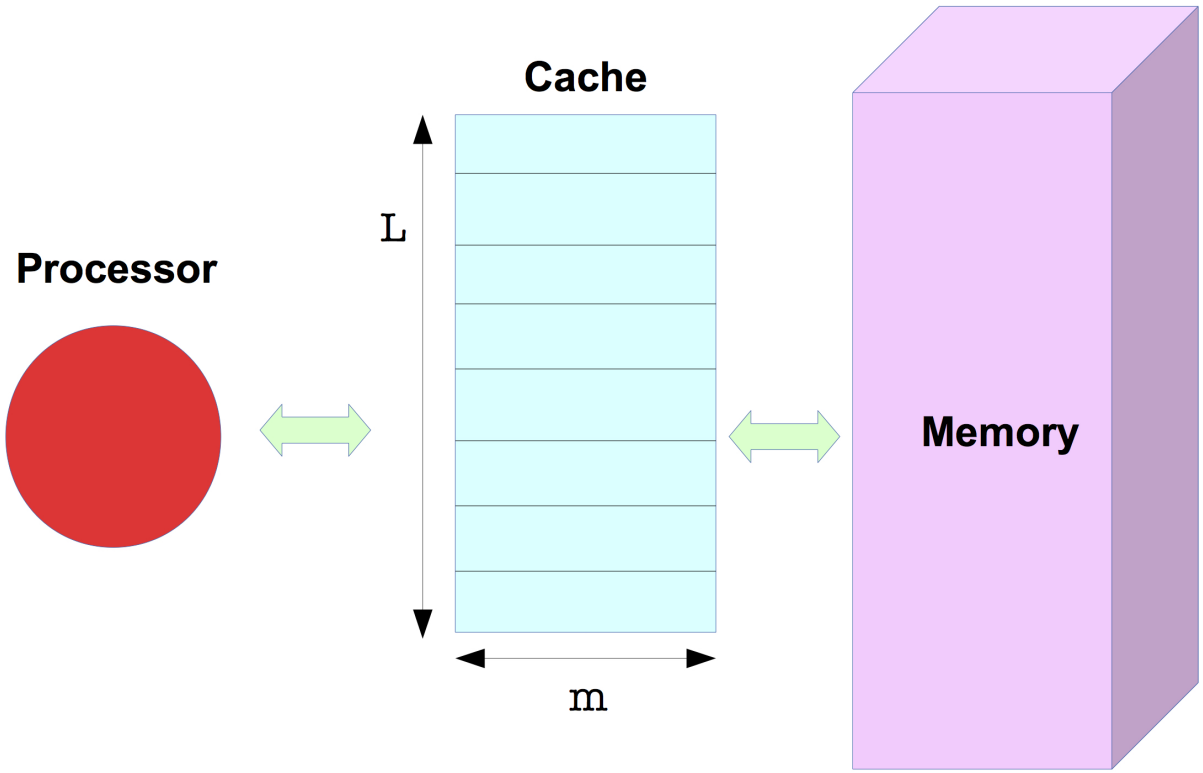


How good is the serial performance?

- Normalized time becomes almost 4x when size of matrix grows from 1000 to 6000
- Experiments done on 3.2 GHz machine
- More than 5 clock cycles taken per double precision multiplication for 6000$\times$6000 matrix !!!

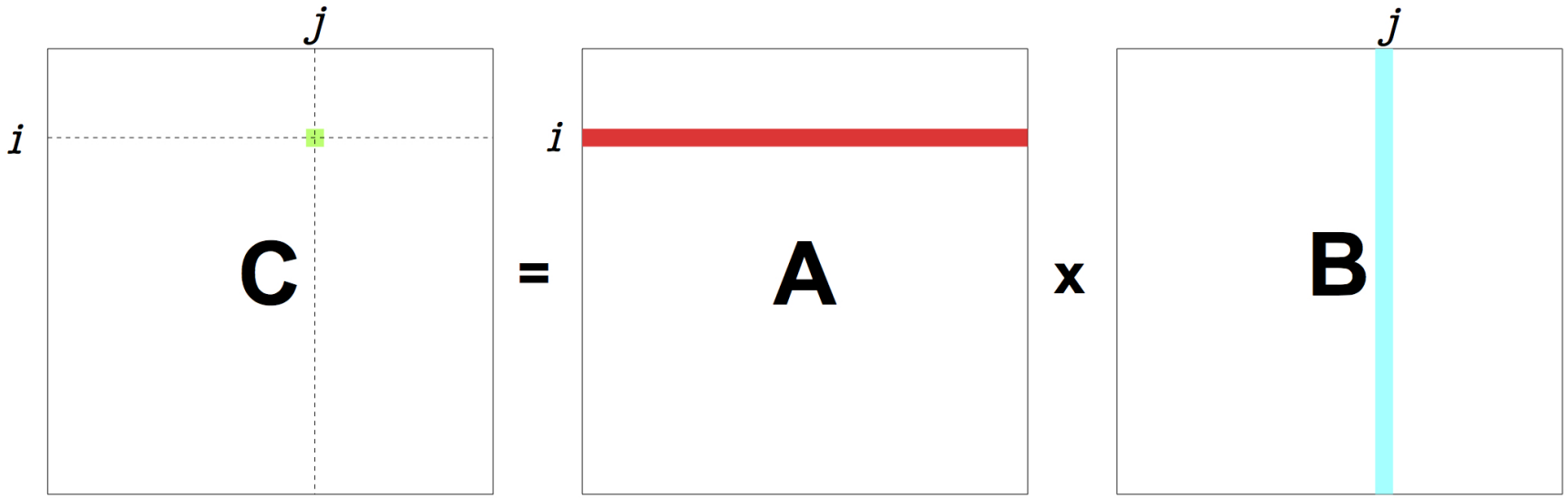
Outline

Memory Hierarchy Model for analysis



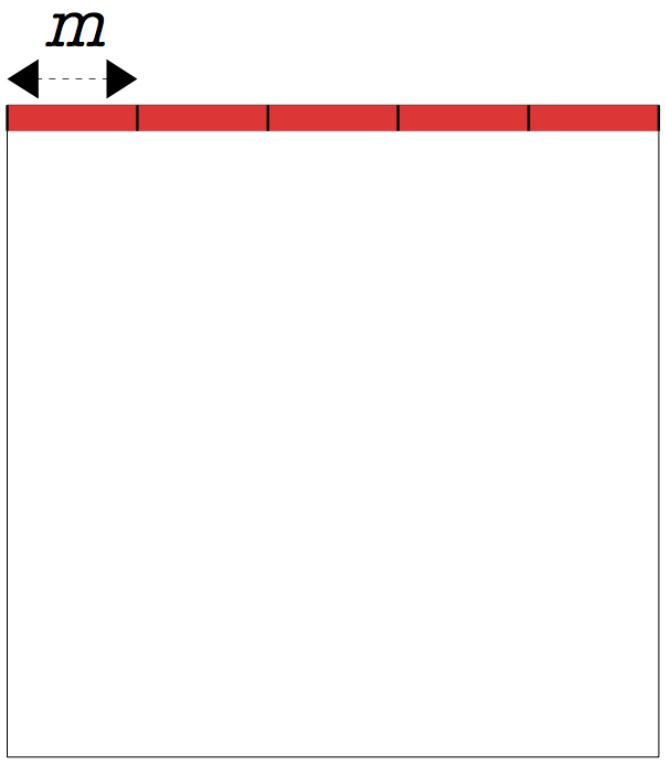
- **L** lines of capacity **m** double precision numbers each
- Tall Cache assumption : $L > m$
- Replacement Policy : Least Recently Used
- No Hardware Prefetching

Memory Access Pattern during multiplication



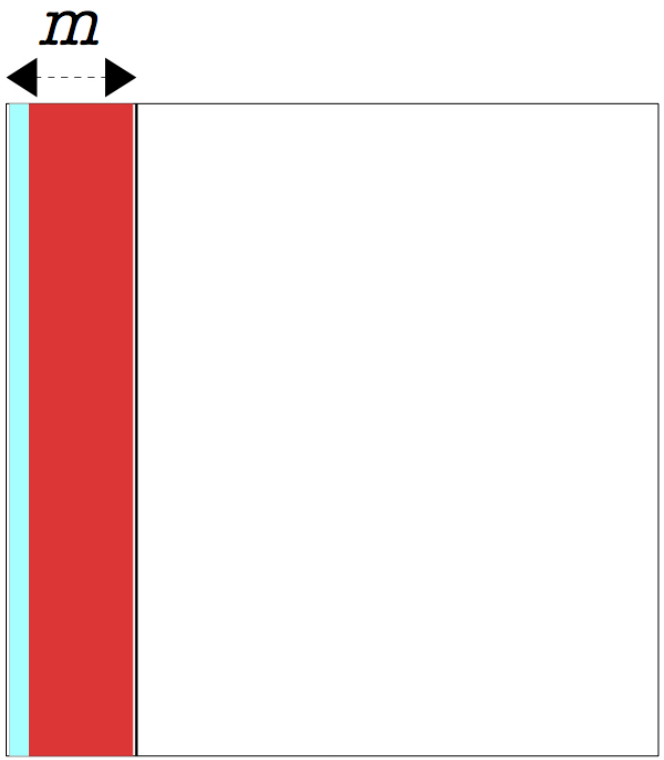
- **A**, **B** and **C** are square matrices of size $n \times n$
- **n** is large, i.e., $n > L$

Memory Access Pattern for A



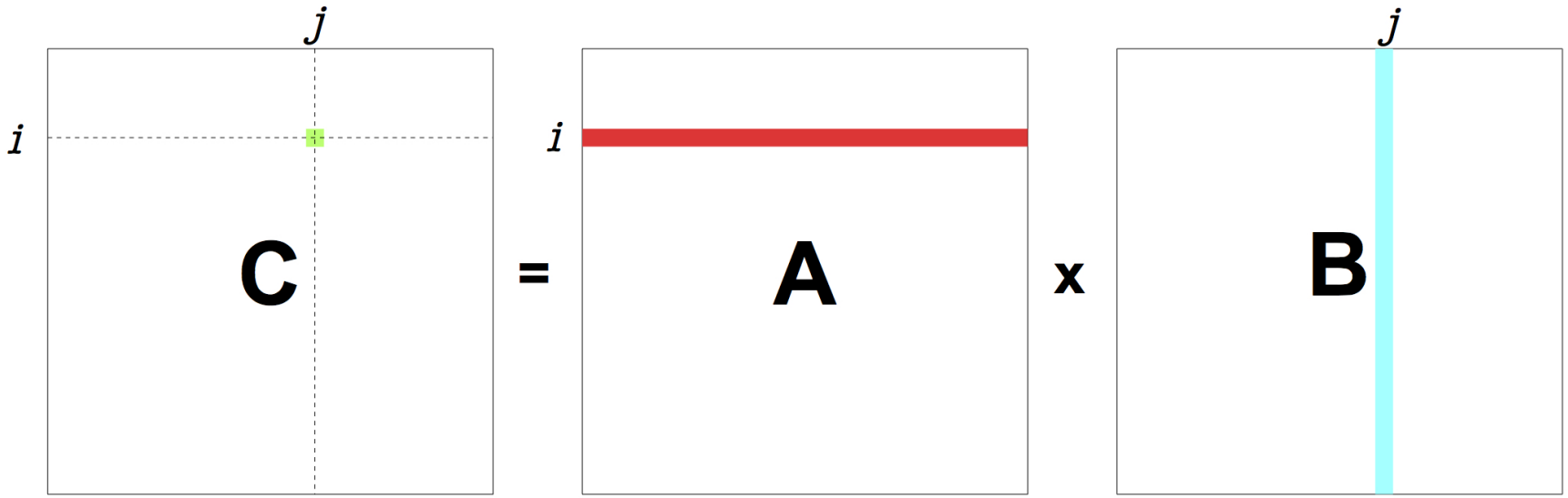
- Sequential access : Accessing a row requires $\frac{n}{m}$ cache misses

Memory Access Pattern for B



- Strided access : Accessing a column requires **n** cache misses

Total cache misses



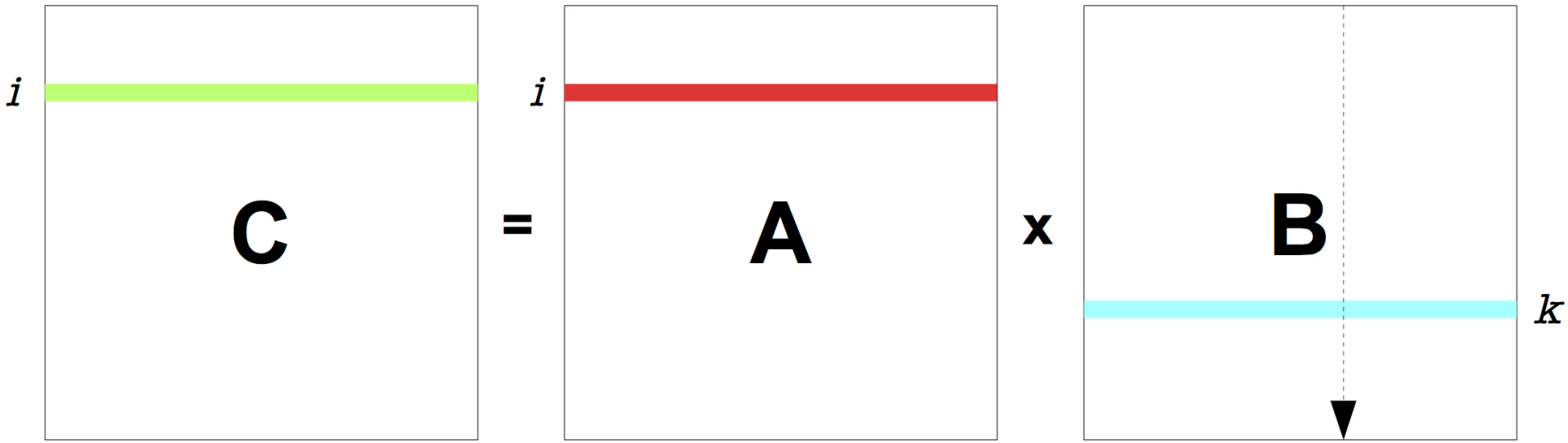
- For computing every $C(i,j)$, the number of cache misses : $1 + \frac{n}{m} + n$
- If $n < mL$, total cache misses : $\frac{2n^2}{m} + n^3$
- If $n > mL$, total cache misses : $\frac{n^2}{m} + \frac{n^3}{m} + n^3$

Is $n < mL$ a practical assumption?

- 64 bytes cache line size means $m = 8$
- 256 KB L2 cache means $mL = 32768$
- For practical problems $n < 10 - 15k$
- Thus $n < mL$ and the total cache misses : $\frac{2n^2}{m} + n^3 = \Theta(n^3)$
- Can this be improved ?

Outline

Alternate Memory Access Pattern



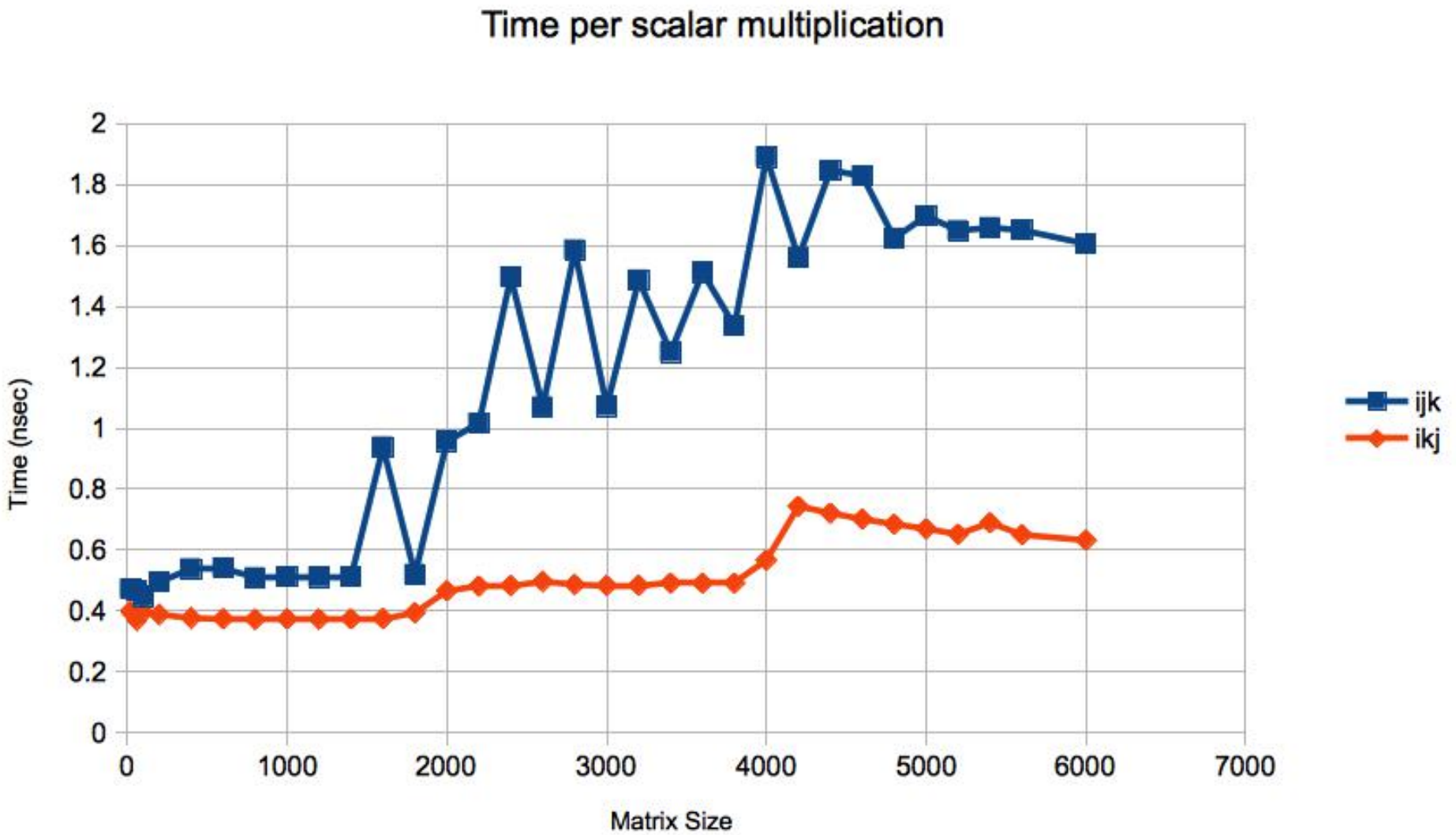
- For computing the row $C(i,:)$, cache misses : $\frac{2n}{m} + \frac{n^2}{m}$
- Total cache misses : $\frac{2n^2}{m} + \frac{n^3}{m} = \Theta(\frac{n^3}{m})$

Improved Multiply

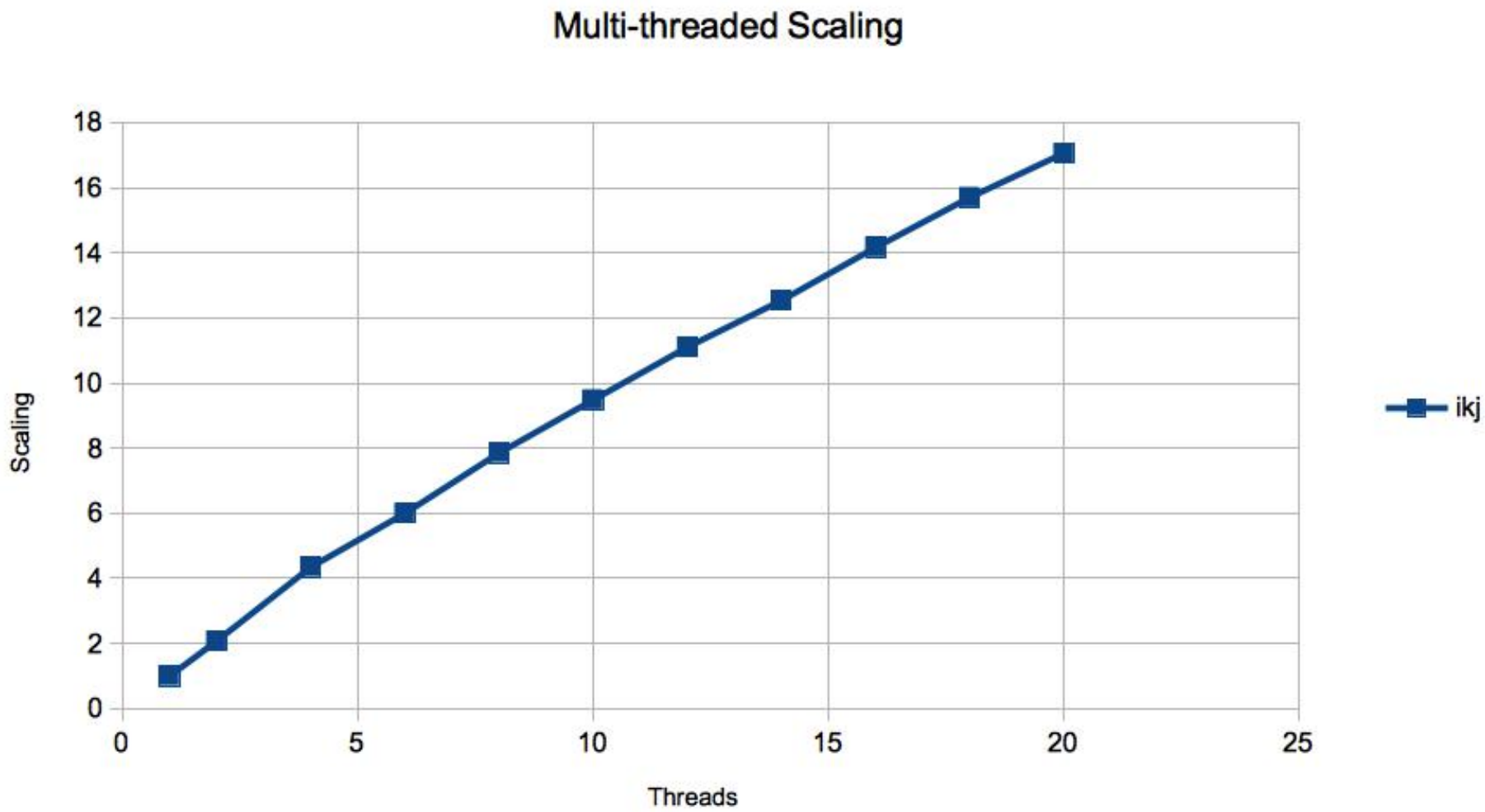
```
void square_dgemm (int n, double* A, double* B, double* C)
{
    for (int i = 0; i < n; ++i)
    {
        const int iOffset = i*n;
        for( int k = 0; k < n; k++ )
        {
            const int kOffset = k*n;
            for (int j = 0; j < n; ++j)
                C[iOffset+j] += A[iOffset+j] * B[kOffset+j];
        }
    }
}
```

- Triple-nested loop with the order of (i, j, k) changed to (i, k, j)

ikj versus ijk

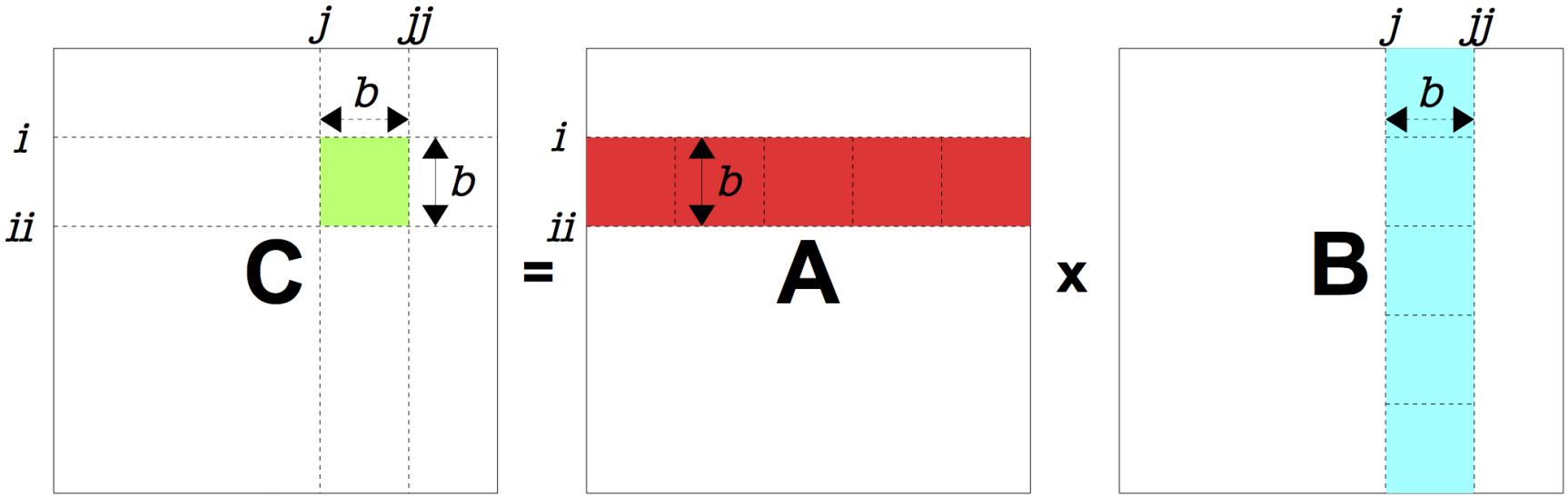


(Almost) Perfect Scaling for matrix of size 6000 $\times$ 6000



Outline

Blocking / Tiling



- Assumptions for analysis : $n\%b = 0$ and $b\%m = 0$
- Cache misses in loading a block : $\frac{b^2}{m}$
- Cache misses in finding a block of C : $\frac{b^2}{m} + \frac{b^2}{m}\frac{2n}{b} = \frac{b^2}{m} + \frac{2nb}{m}$
- Total cache misses : $\frac{n^2}{b^2}(\frac{b^2}{m} + \frac{2nb}{m}) = \frac{n^2}{m} + \frac{2n^3}{mb} = \Theta(\frac{n^3}{mb})$

Choosing blocking parameter **b**

- The 3 blocks for A, B and C should **just** fit in the cache
- $3b^2 = mL$, i.e., $b = \sqrt{\frac{mL}{3}}$
- For L1 cache of capacity 32KB, $mL = 4096$ and $b = 36.95$
- A good value for b is 32
- Total cache misses : $\frac{2n^3}{mb} = \frac{2\sqrt{3}n^3}{m\sqrt{mL}} = \Theta(\frac{n^3}{m\sqrt{\text{Cache Size}}})$

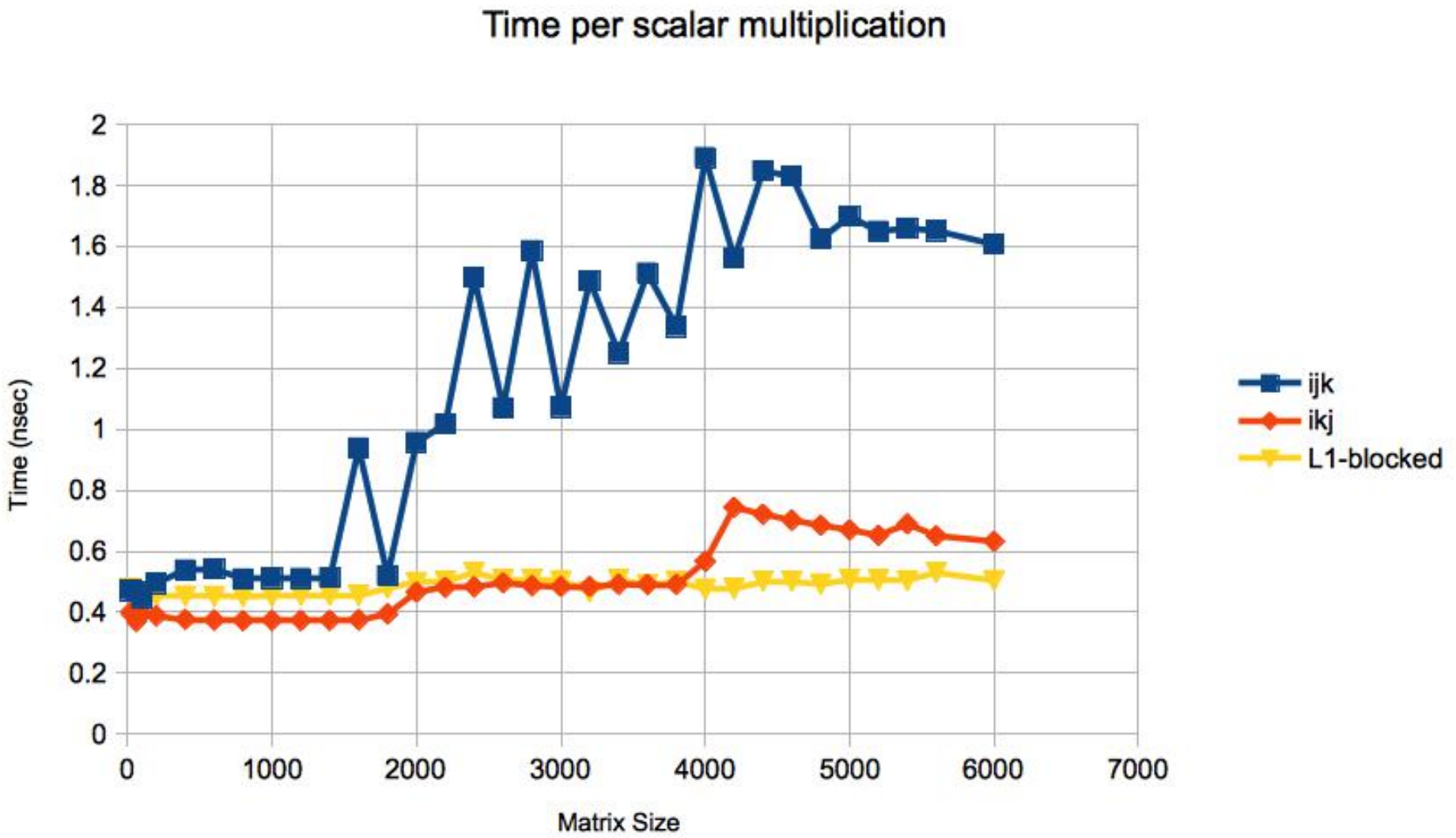
Tiled Multiply

```
void square_dgemm (int n, double* A, double* B, double* C)
{
    for (int i = 0; i < n; i += BLOCK_SIZE)
    {
        const int iOffset = i * n;
        for (int j = 0; j < n; j += BLOCK_SIZE)
            for (int k = 0; k < n; k += BLOCK_SIZE)
            {
                /* Correct block dimensions if block "goes off
                    edge of" the matrix */
                int M = min (BLOCK_SIZE, n-i);
                int N = min (BLOCK_SIZE, n-j);
                int K = min (BLOCK_SIZE, n-k);
                /* Perform individual block dgemm */
                do_block(n, M, N, K, A + iOffset + k,
                        B + k*n + j, C + iOffset + j);
            }
    }
}
```

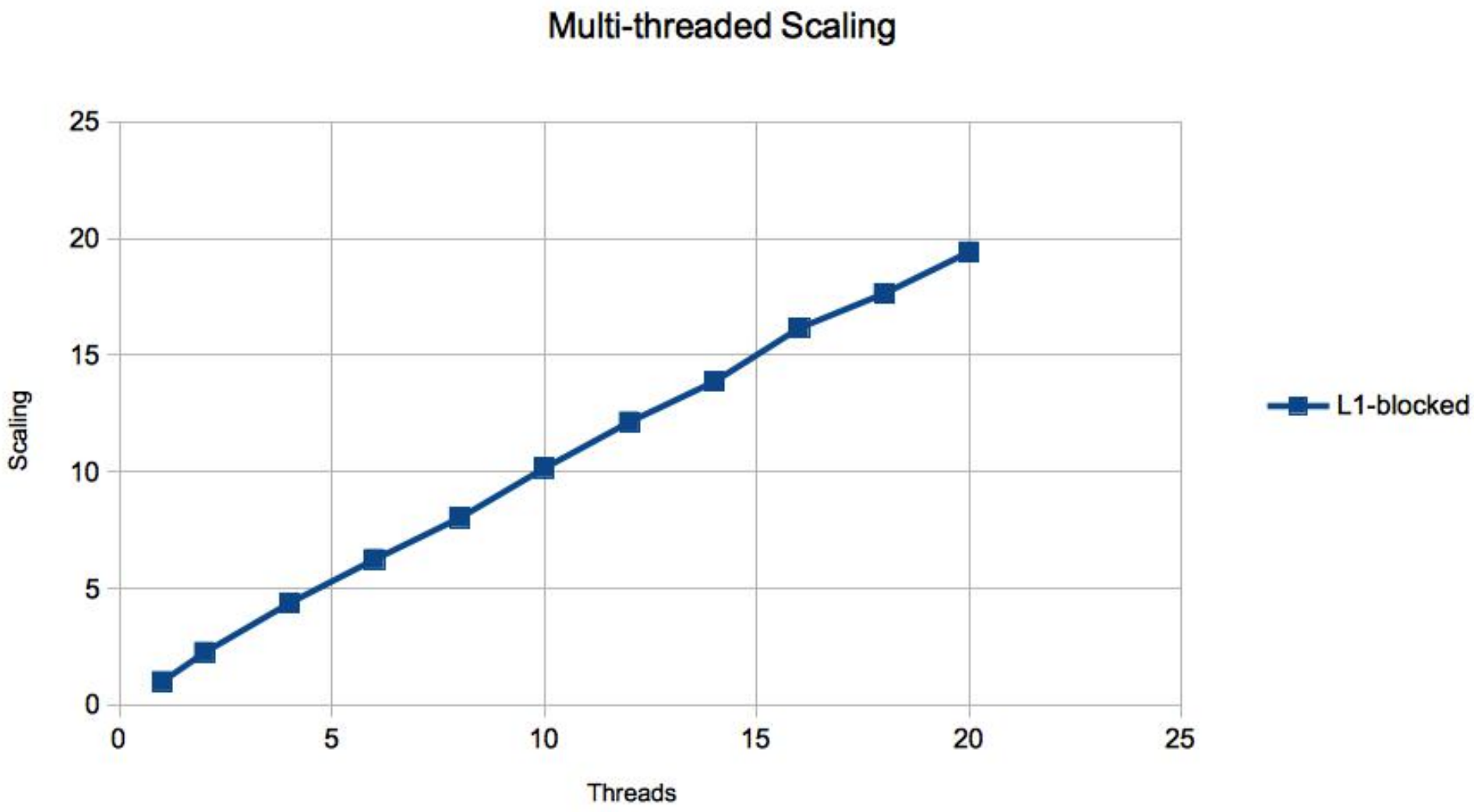
Tiled Multiply...

```
static void do_block (int n, int M, int N, int K, double* A,
    double* B, double* C)
{
    for (int i = 0; i < M; ++i)
    {
        const int iOffset = i*n;
        for (int j = 0; j < N; ++j)
        {
            double cij = 0.0;
            for (int k = 0; k < K; ++k)
                cij += A[iOffset+k] * B[k*n+j];
            C[iOffset+j] += cij;
        }
    }
}
```

Tiled versus Normal



Tiled MT scaling



Instruction Level Parallelism

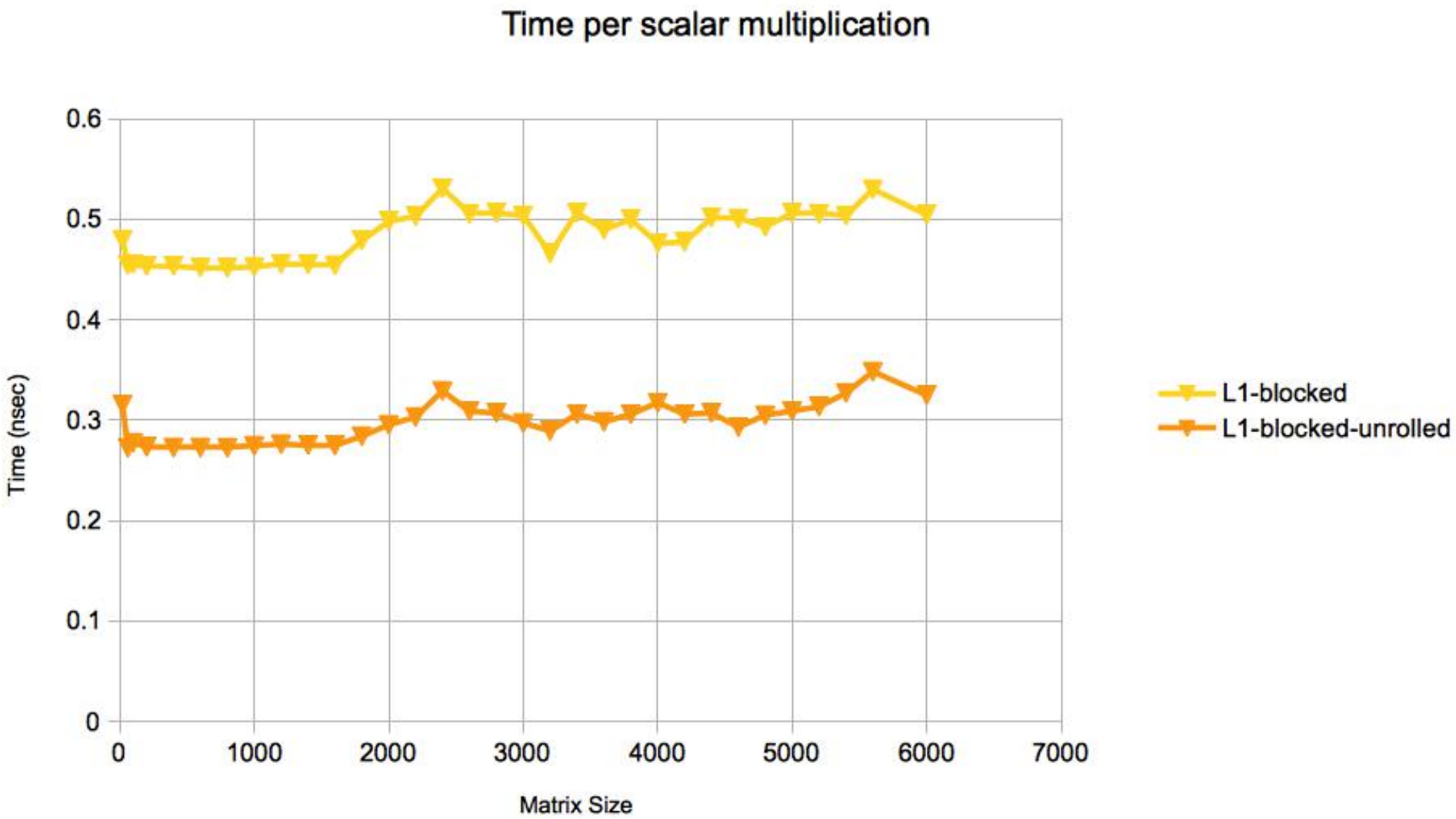
- Given that we have made the data being worked upon available in the cache closest to the processor, we could use some ILP
- ILP kicks in when there is significant amount of independent work available in a single block of code
- Loop unrolling can help us achieve that
- Compilers also unroll loop but in this case there are too many nesting levels for the compiler to do the correct thing

Tiled Multiply with unrolling

```
for (int k = 0; k < K; ++k)
    cij += A[iOffset+k] * B[k*n+j];
```

```
for (int k = 0; k < K; k += 8)
{
    const double d0 = A[iOffset+k] * B[k*n+j];
    const double d1 = A[iOffset+k+1] * B[(k+1)*n+j];
    const double d2 = A[iOffset+k+2] * B[(k+2)*n+j];
    const double d3 = A[iOffset+k+3] * B[(k+3)*n+j];
    const double d4 = A[iOffset+k+4] * B[(k+4)*n+j];
    const double d5 = A[iOffset+k+5] * B[(k+5)*n+j];
    const double d6 = A[iOffset+k+6] * B[(k+6)*n+j];
    const double d7 = A[iOffset+k+7] * B[(k+7)*n+j];
    cij += (d0 + d1 + d2 + d3 + d4 + d5 + d6 + d7);
}
```

Tiled Multiply with unrolling...



What about the L2 cache?

- In addition to L1, blocking can be done for the L2 cache also $\Rightarrow$ 2-level tiled code
- Next programming assignment

Automatic tuning

- Manual optimization and tuning is tedious and error-prone
- Entire process needs to be redone in full for any new architecture
- Multi-threaded optimization adds further complexity
- Code generated automatically by parameterized code generators
  - Automatically Tuned Linear Algebra Software
  - Portable High Performance ANSI C
- Essentially a search problem

Outline

Dual Core

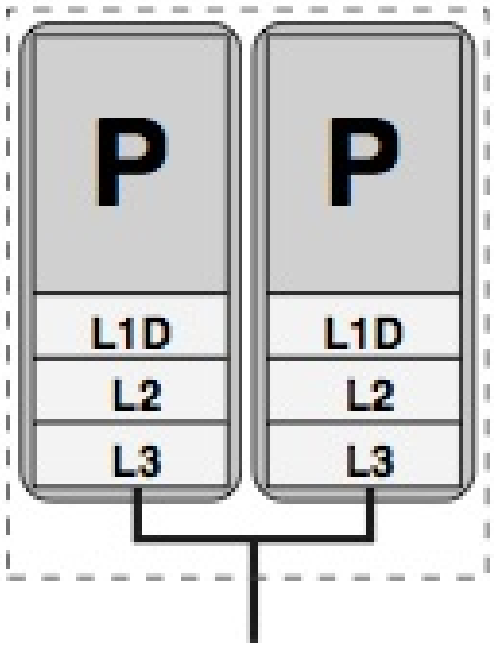


Figure : Courtesy of G. Hager & G. Wellein

- Each core has it's own cache for all levels
- eg., Intel Montecito

Quad Core

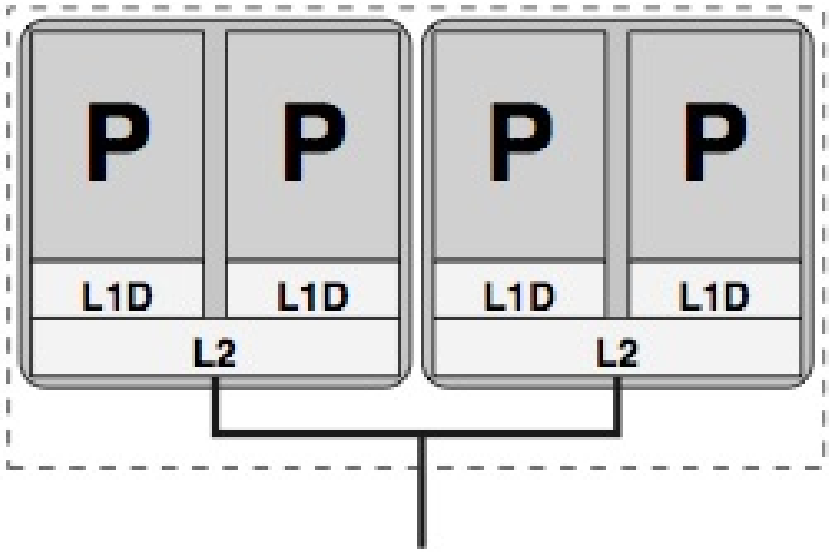


Figure : Courtesy of G. Hager & G. Wellein

- Separate L1, Shared L2 (2 dual-core L2 groups)
- Shared cache enables inter-core communication without going to the main memory
- Reduced latency and improved bandwidth
- eg., Intel Harpertown

Hexa Core

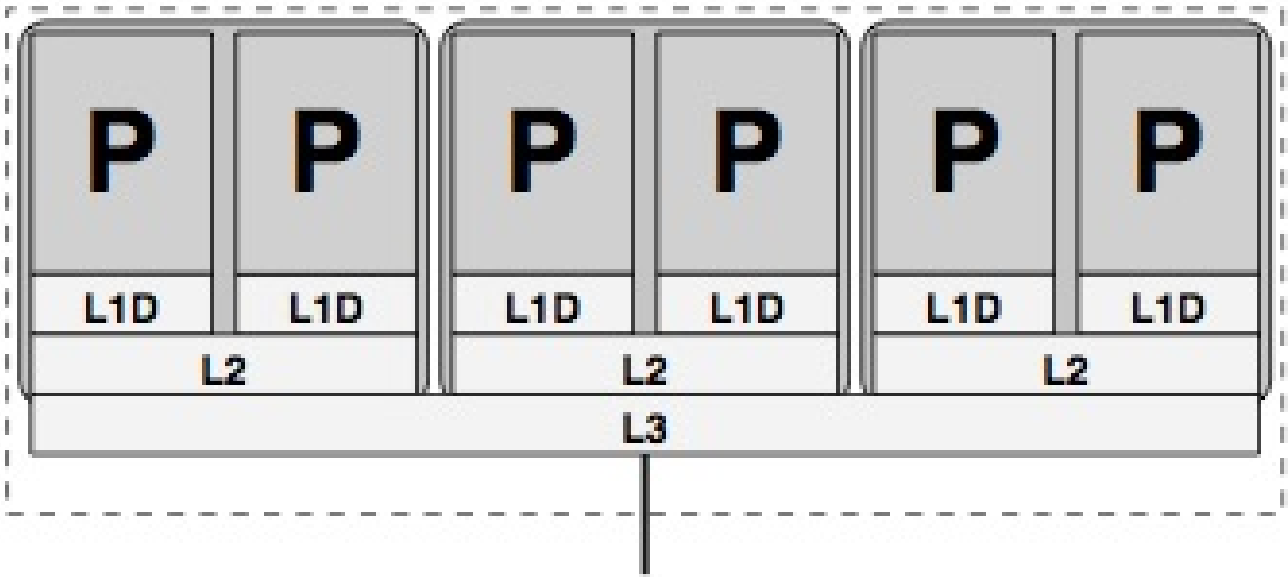


Figure : Courtesy of G. Hager & G. Wellein

- 6 single-core L1 groups, 3 dual-core L2 groups
- L3 shared for all cores
- Cache bandwidth shared across number of cores connected
- eg., Intel Dunnington

Uniform Memory Access (UMA)

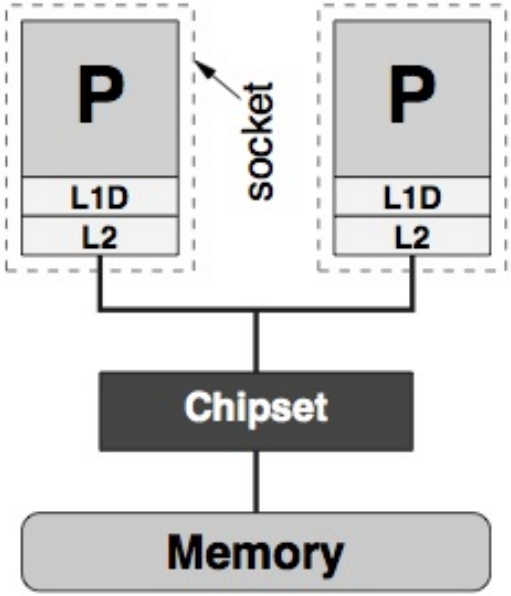


Figure : Courtesy of G. Hager & G. Wellein

- 2 single-core CPUs share a common FrontSide bus (FSB)
- Arbitration protocols built into the CPUs
- Chipset connects to memory and other I/O systems
- Data can be transfered to/from only one CPU at a time

Uniform Memory Access...

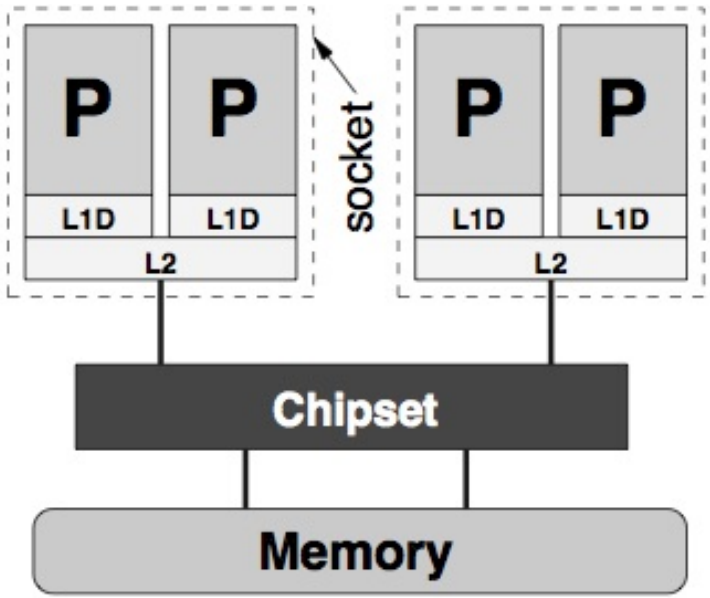


Figure : Courtesy of G. Hager & G. Wellein

- FSB not shared by sockets
- Role of chipset becomes more important
- *Anisotropic* system - Cores on same socket are "closer" than those on other sockets

Integrated Memory Controller

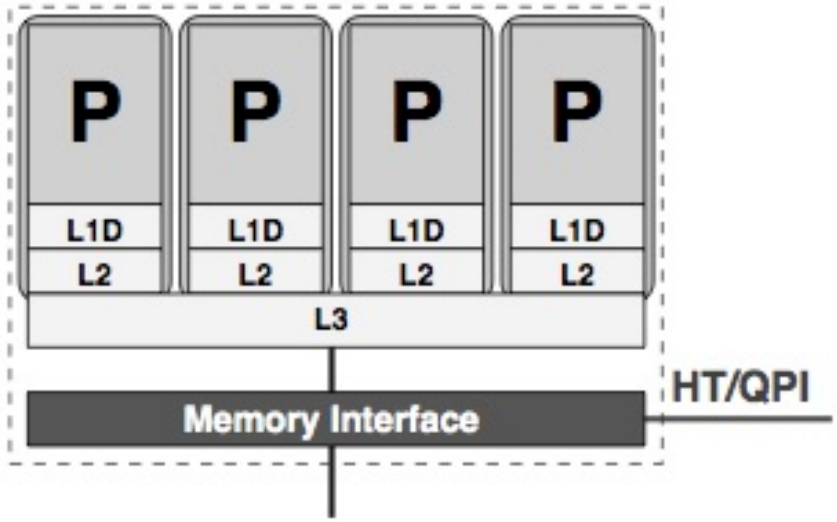


Figure : Courtesy of G. Hager & G. Wellein

- Integrated memory controller allows direct connection to memory and/or other sockets
- Intel QuickPath (QPI), AMD HyperTransport (HT)
- eg. Intel Nehalem, AMD Shanghai

ccNUMA

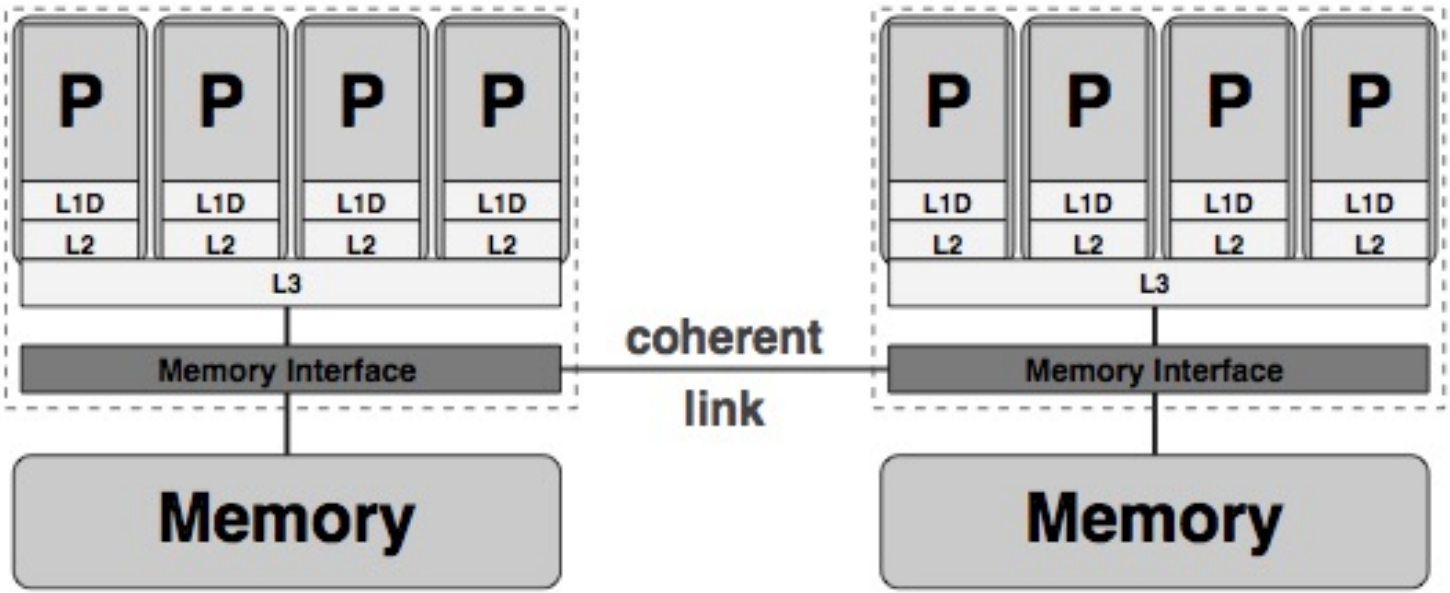


Figure : Courtesy of G. Hager & G. Wellein

- cache-coherent Non Uniform Memory Access
- Every UMA building block is a Locality Domain (LD)
- Provides scalable bandwidth for large number of processors

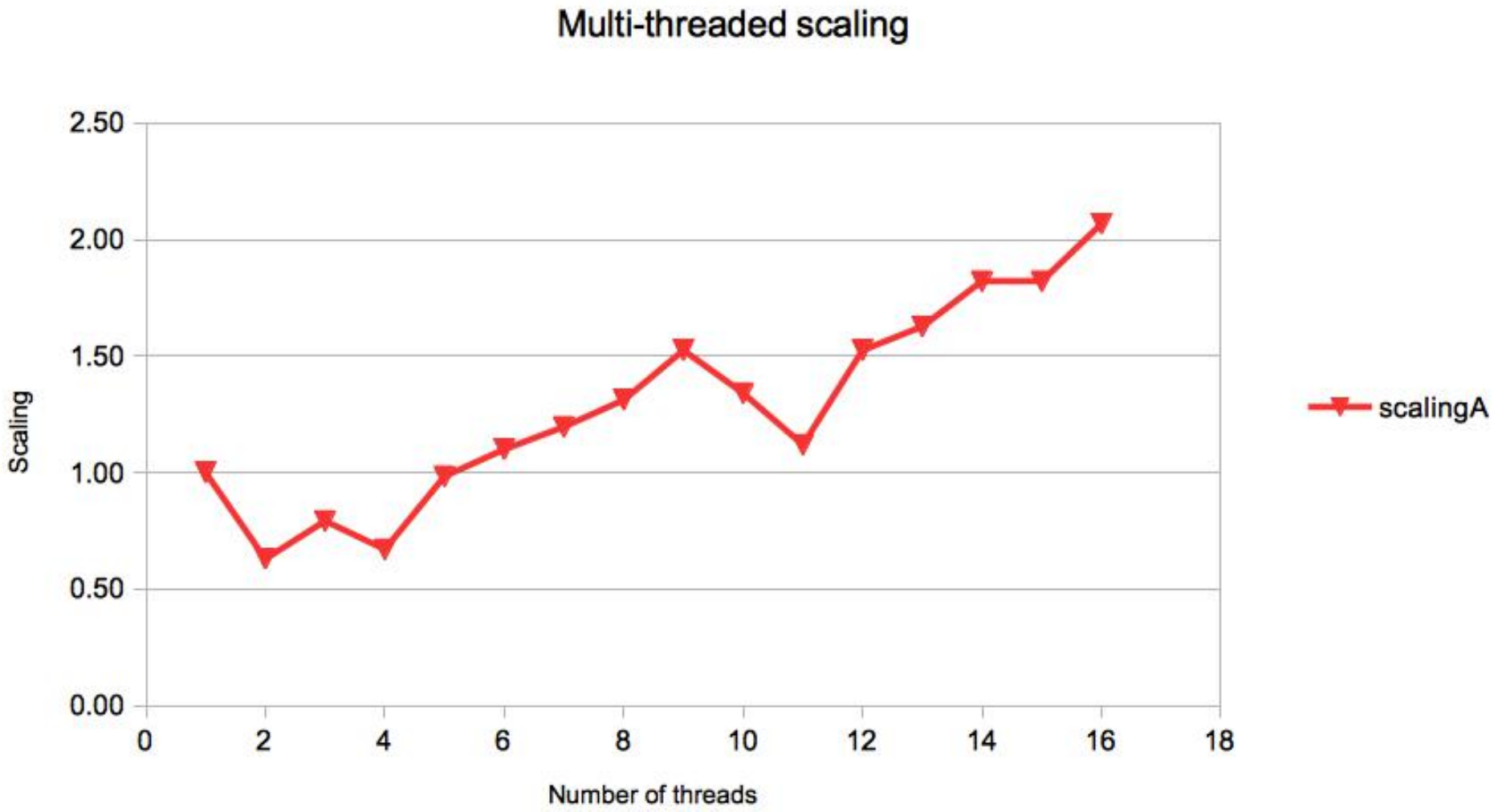
Cache Coherence

- Explicit logic required to maintain cache coherence
- MESI protocol
  - Modified : Cache line modified in this cache, resides in no other cache
  - Exclusive : Read from memory, not modified yet, resides in no other cache
  - Shared : Read from memory, not modified yet, may reside in other caches
  - Invalid : Data in cache line is garbage
- Cache coherence traffic can hurt application performance if same cache line is modified frequently by different locality domains (false sharing).

Back to π

```
    const double deltaX = 1.0/(double)numPoints;
    double pi = 0.0;
    omp_set_num_threads(numThreads);
    double components[numThreads];
    for(int i = 0; i < numThreads; ++i)
        components[i] = 0.0;
#pragma omp parallel shared(components)
    {
        const int nt = omp_get_num_threads();
        const int pointsPerThread = numPoints/nt;
        const int threadId = omp_get_thread_num();
        double xi = (0.5 + pointsPerThread * threadId) * deltaX;
        for(int i = 0; i < pointsPerThread; ++i)
        {
            components[threadId] += 4.0/(1 + xi * xi);
            xi += deltaX;
        }
    }
    for(int i = 0; i < numThreads; ++i)
        pi += components[i];
    pi *= deltaX;
```

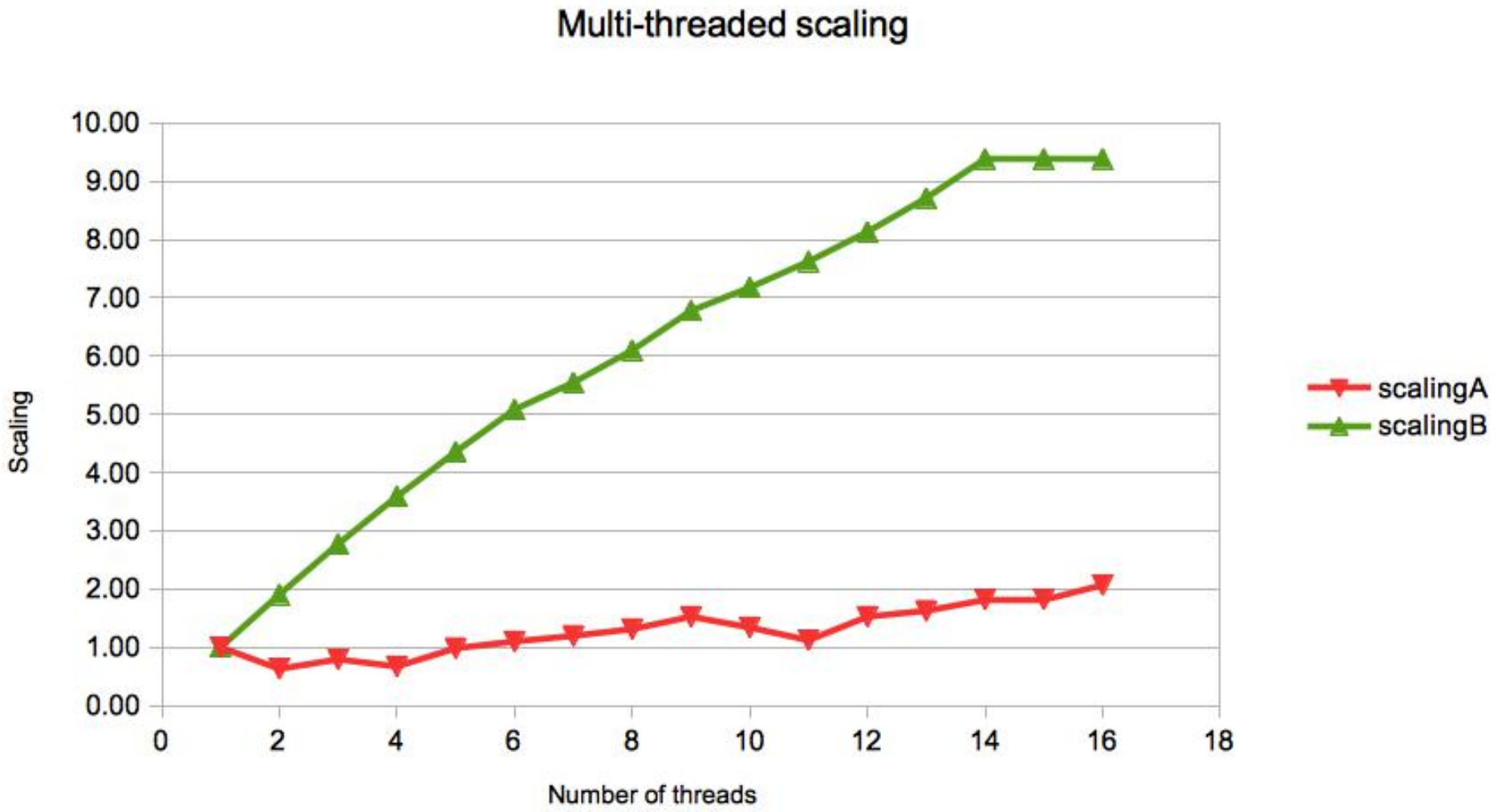
False Cache Line Sharing



Back to π ...

```
    const double deltaX = 1.0/(double)numPoints;
    double pi = 0.0;
    omp_set_num_threads(numThreads);
    double components[numThreads];
#pragma omp parallel shared(components)
    {
        const int nt = omp_get_num_threads();
        const int pointsPerThread = numPoints/nt;
        double myComponent = 0.0;
        const int threadId = omp_get_thread_num();
        double xi = (0.5 + pointsPerThread * threadId) * deltaX;
        for(int i = 0; i < pointsPerThread; ++i)
        {
            myComponent += 4.0/(1 + xi * xi);
            xi += deltaX;
        }
        components[threadId] = myComponent;
    }
    for(int i = 0; i < numThreads; ++i)
        pi += components[i];
    pi *= deltaX;
```

False Cache Line Sharing ...



Non Uniform Access Time

numactl --hardware

```
available:  2 nodes (0-1)
node 0 cpus:  0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32
34 36 38 40 42 44 46
node 0 size:  131026 MB
node 0 free:  124290 MB
node 1 cpus:  1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33
35 37 39 41 43 45 47
node 1 size:  131072 MB
node 1 free:  126752 MB
node distances:
node 0 1
0:  10 20
1:  20 10
```

Highly scalable ccNUMA

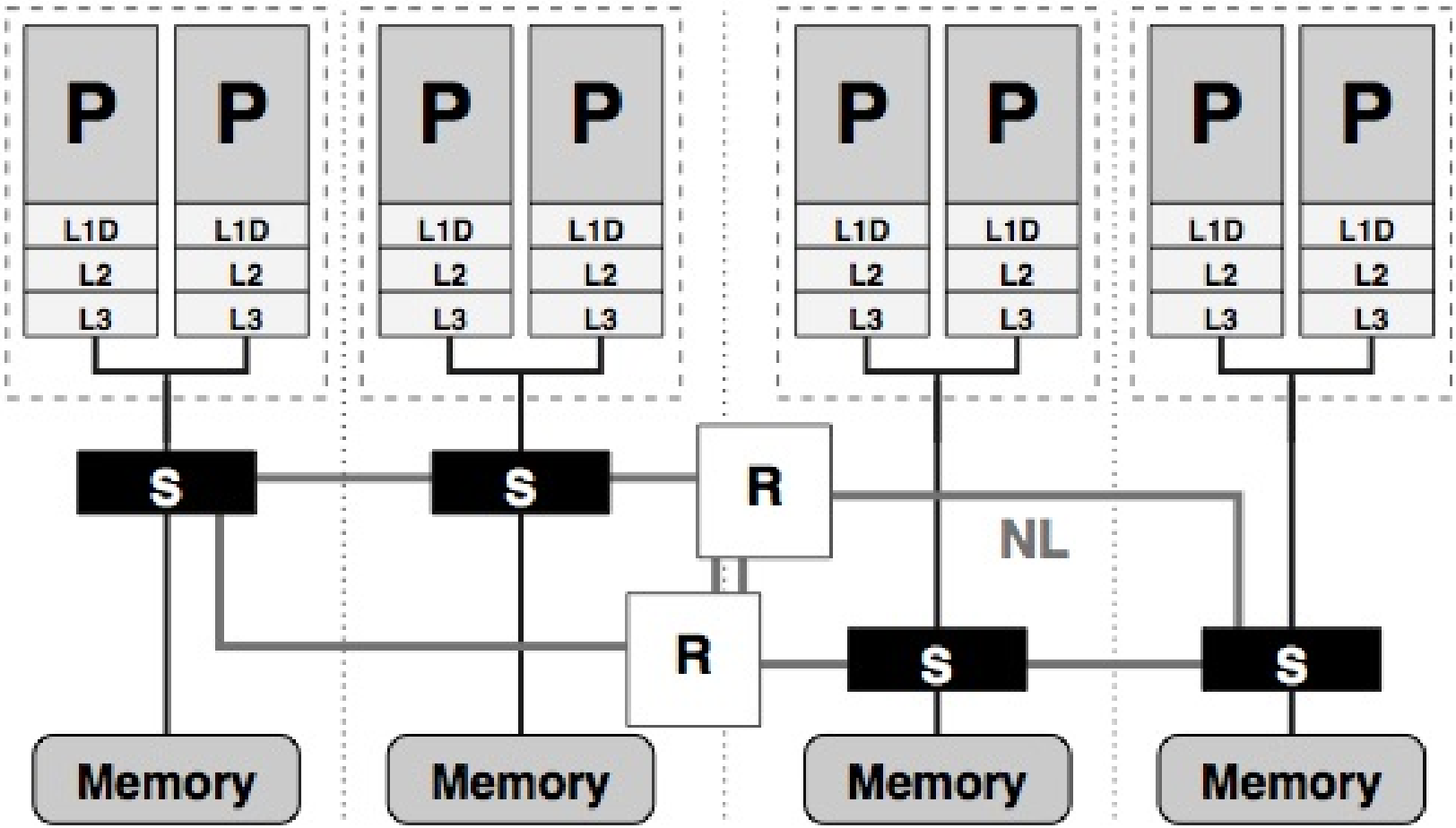


Figure : Courtesy of G. Hager & G. Wellein

Outline

Function Based Profiling

- Compiler modifies each function call to log the number of calls, its callers and the time taken
- Best suited when each function call takes significant time
- Overhead significant if many functions with short runtime
- eg., gprof from GNU binutils package

Line Based Profiling

- Program is sampled at regular intervals and program counter and current call stack are recorded
- Program needs to run long enough for results to be accurate
- Possible to get profiling information down to the source and assembly level
- eg., gperftools from Google, Vtune Amplifier from Intel

Hardware Performance Counters

- Special on-chip registers which get incremented every time a certain event occurs
- Example events
  - Bus transactions
  - Mis-predicted branches
  - Cache Misses at various levels
  - Pipeline stalls
  - Number of loads and stores
  - Number of instructions executed
- eg., Vtune Amplifier from Intel, oprofile, PAPI

Optimize Memory Access

- Sequential access of data in arrays
- If $arr[i][j]$ is in the cache, $arr[i][j+1]$ is likely to be in the cache also. However, $arr[i+1][j]$ is NOT likely to be in the cache
- Avoid using nested containers like vector of vectors for storing matrices
- Redesign data-structures for locality of access

Optimize Memory Access...

```
const int size = 10000;
int a[size], b[size];
for(int i = 0; i < size; ++i)
{
    b[i] = func(a[i]);
}
```

```
typedef struct { int a; int b;} myPair;
myPair ab[size];
for(int i = 0; i < size; ++i)
{
    ab[i].b = func(ab[i].a);
}
```

Minimize Jumps/Branches

- Use inline functions for short functions.
- Replace long if...else if...else if... chains by switch statements.
  - Such chains may lead to frequent branch mis-prediction.
  - Pipelined architectures incur severe cost (15-20 cycles) for every mis-predicted branch.
  - Compiler may optimize switch into a table lookup requiring a single jump.
  - If converting to switch is not possible, put the most common clauses at the beginning of the if chain.
- Where applicable, replace deeply recursive functions by iterative ones. eg., BFS, DFS.

Exploit instruction level parallelism

- Most modern servers have 4-way superscalar cores.
- Blocks of code (eg., in the body of a loop) should have enough independent instructions.
- Unrolling of loops may help in achieving this.
- Inlined functions (small ones) also help, better register optimization being the other benefit.

Loop Unrolling Example

```
for(int i = 0; i < 100; ++i)
{
    if(i % 2 == 0)
        func1(i);
    else
        func2(i);
    func3(i);
}
```

```
for(int i = 0; i < 100; i += 2)
{
    func1(i);
    func3(i);
    func2(i+1);
    func3(i+1);
}
```

General compiler based optimizations

- First and foremost job : Correct and reliable mapping of high-level source code to machine code
- Major code transformation areas
  - Function inlining
  - Constant folding
  - Constant propagation
  - Common subexpression elimination
  - Register variables
  - Branch analysis
  - Loop analysis
  - Algebraic Reduction

Function Inlining

```
double square (double a)
{
    return a * a;
}

double parabola (double b)
{
    return square(b) + 1.0;
}
```

```
double parabola (double b)
{
    return b * b + 1.0;
}
```

Constant folding

```
double x, y, z;
y = x * (17.0/19.0);
z = x * 17.0 / 19.0;
```

```
double x, y, z;
y = x * 0.89473684210526316374;
z = x * 17.0 / 19.0;
```

Constant propagation

```
double parabola (double b)
{
    return b * b + 1.0;
}

double x, y;
x = parabola( 13.5 );
y = x * 2.3;
```

```
double x, y;
x = 183.25;
y = 421.475;
```

Common subexpression elimination

```
double a, b, c, d;
b = (a + 6.0);
c = (a + b) * (a + b);
d = (a + b) / a;
```

```
double a, b, c, d, temp;
temp = a + a + 6.0;
c = temp * temp;
d = temp / a;
```

Branch Analysis : Join identical branches

```
double x, y, z;
bool b;
if( b )
{
    y = parabola(x);
    z = y + 4.0;
}
else
{
    y = square(x);
    z = y + 4.0;
}
...
```

```
if( b )
{
    y = parabola(x);
}
else
{
    y = square(x);
}
z = y + 4.0;
...
```

Branch Analysis : Eliminate jumps

```
int foo (int a, bool b)
{
    if(b)
        a = a * 4;
    else
        a = a * 5;
    return a;
}
```

```
int foo (int a, bool b)
{
    if(b)
    {
        a = a * 4;
        return a;
    }
    else
    {
        a = a * 5;
        return a;
    }
}
```

Other Compiler Optimizations

- Loop unrolling
- Loop invariant code motion
- Instructions reordering and scheduling
- Pointer elimination
- ...

Standard optimization options for gcc

Option	Details	Number of optimization flags
-O0	Default, Fast compilation and low memory usage during compilation	0
-O1	Quick and light transformations that preserve execution ordering	39
-O2	More optimizations with instruction reordering and inlining	83
-O3	Heavy duty optimizations with a lot of transformations	93
-Ofast	O3 with fast, standards incompliant floating point calculations	94
-Os	Optimize for size of executable	66

Helping the compiler

- Compilers can not optimize across modules
- Declare objects and fixed size arrays (not very large) inside functions that need them; Avoid dynamic memory allocation
- Write programs to access data in arrays sequentially; Compilers can not do this transformation
- Use __restrict keyword for pointers when the program logic rules out pointer aliasing
- ALWAYS PROFILE AFTER MAKING A SIGNIFICANT CHANGE

Pointer Aliasing

```
void foo (int * a, int * p)
{
    for( int i = 0; i < 1000; ++i)
        a[i] = *p + 2;
}

void func1 ()
{
    int arr[1000];
    foo(arr, &arr[10]);
}

void func2 ()
{
    int arr[1000], b = 20;
    foo(arr, &b);
}
```

Further Reading

- Introduction to High Performance Computing for Scientists and Engineers - Hager, Wellein : Chapter 1, 2, 3
- `http://agner.org/optimize/optimizing_cpp.pdf`