Lecture 4: Intro to parallel machines and models

David Bindel

4 Feb 2014

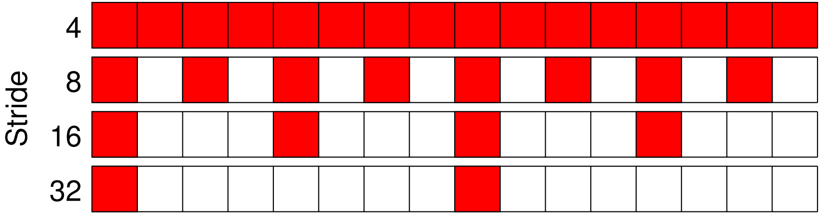
Logistics

- If you want an account on C4, please at least audit!
- HW 0 due midnight
  - If you were confused by membench, today might help
  - Please try to get it in on time even if you registered late
- HW 1 posted
  - You should work in groups of 2–4
  - You *must* work with someone, at least for this project
  - Piazza has a teammate search: use it!
- Note: the entire class will *not* be this low-level!

A memory benchmark (membench)

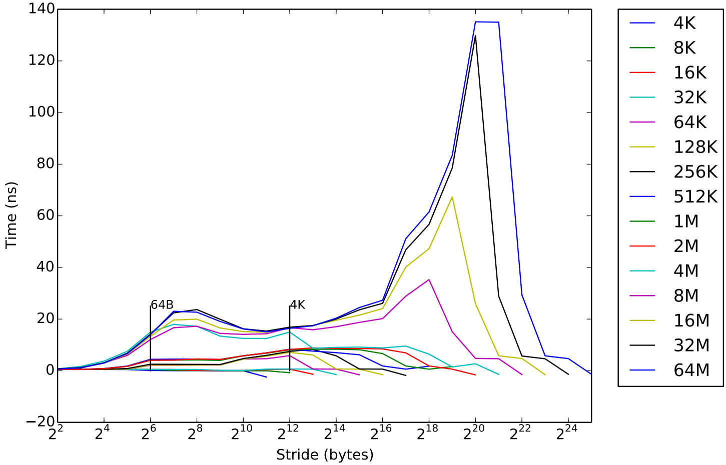
```
for array A of length L from 4 KB to 8MB by 2x
  for stride s from 4 bytes to L/2 by 2x
  time the following loop
    for i = 0 to L by s
      load A[i] from memory
```

Membench in pictures

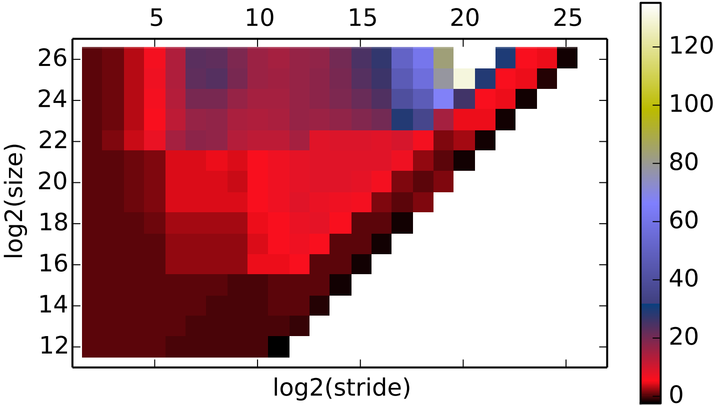


- Size = 64 bytes (16 ints)
- Strides of 4 bytes, 8 bytes, 16 bytes, 32 bytes

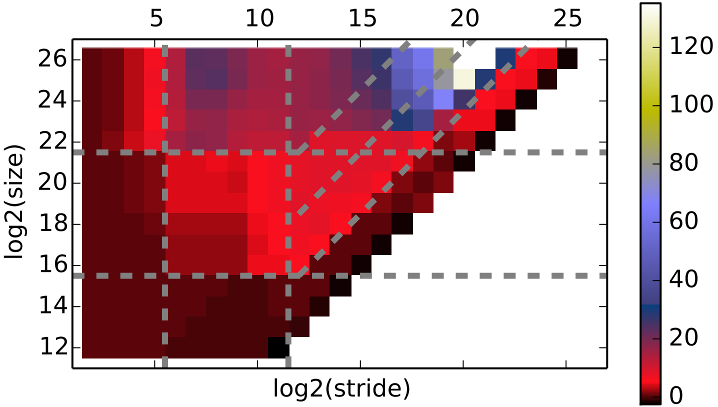
Membench on C4



Membench on C4: another view



Membench on C4



- Vertical: 64B line size (2^5), 4K page size (2^{12})
- Horizontal: 32K L1 (2^{15}), 256K L2 (2^{18}), 6 MB L3
- Diagonal: 8-way cache associativity, 512 entry L2 TLB

Note on storage

0	5	10	15	20
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24

Column major

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Row major

Two standard matrix layouts:

- Column-major (Fortran): A(i,j) at A+i+j*n
- Row-major (C): A(i,j) at A+i*n+j

I default to column major.

Also note: C doesn't really support matrix storage.

Matrix multiply

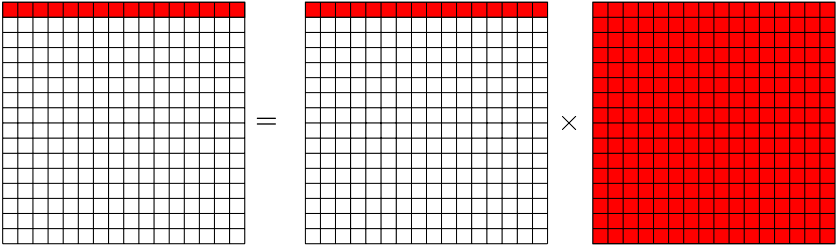
Consider naive square matrix multiplication:

```
#define A(i,j) AA[i+j*n]
#define B(i,j) BB[i+j*n]
#define C(i,j) CC[i+j*n]

for (i = 0; i < n; ++i) {
  for (j = 0; j < n; ++j) {
    C(i,j) = 0;
    for (k = 0; k < n; ++k)
      C(i,j) += A(i,k)*B(k,j);
  }
}
```

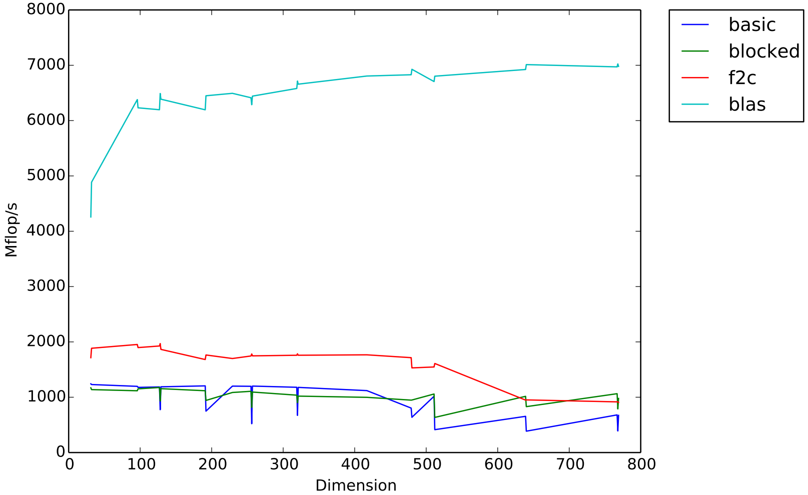
How fast can this run?

One row in naive matmul

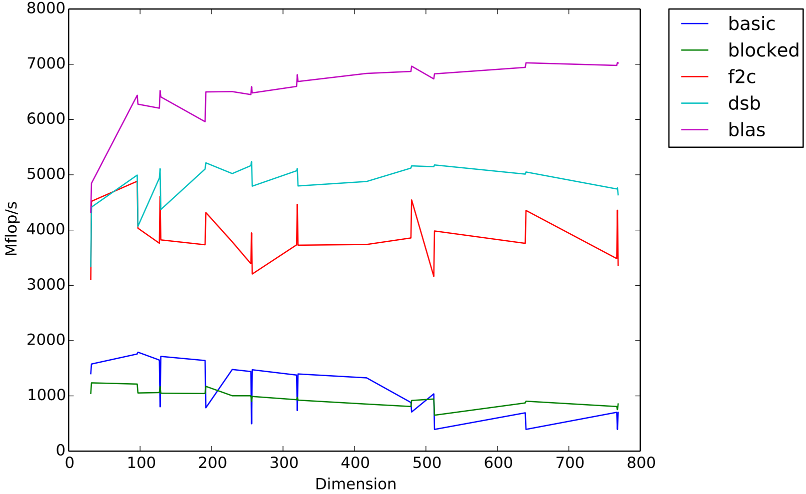


- Access A and C with stride of $8M$ bytes
- Access *all* $8M^2$ bytes of B before first re-use
- Poor *arithmetic intensity*

Matrix multiply compared (GCC)

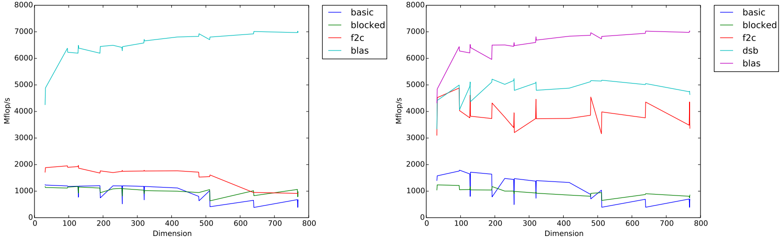


Matrix multiply compared (Intel compiler)



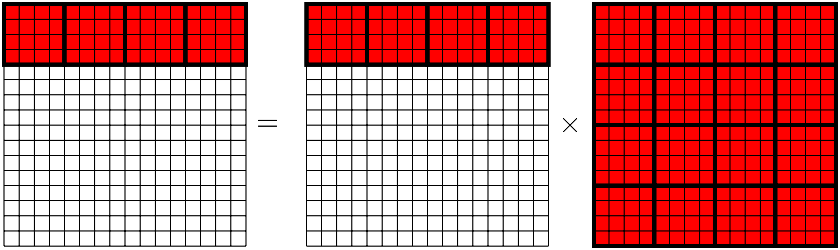
20× difference between naive and tuned!

Hmm...



- Compiler makes some difference (maybe 1.5$\times$)
  - Naive Fortran *is* often faster than naive C
  - But "unfair": `ifort` recognizes matrix multiply!
- Local instruction mix sets "speed of light"
- Access patterns determine how close to light speed we get

Engineering strategy



- Start with small "kernel" multiply
  - Maybe odd sizes, strange memory layouts – just go fast!
  - May want to play with SSE intrinsics, compiler flags, etc.
  - Deserves its own timing rig (see `mm_kernel`)
- Use blocking based on kernel to improve access pattern

Simple model

Consider two types of memory (fast and slow) over which we have complete control.

- m = words read from slow memory
- t_m = slow memory op time
- f = number of flops
- t_f = time per flop
- $q = f/m$ = average flops / slow memory access

Time:

$$ft_f + mt_m = ft_f \left(1 + \frac{t_m/t_f}{q}\right)$$

Larger q means better time.

How big can q be?

1. Dot product: n data, $2n$ flops
2. Matrix-vector multiply: n^2 data, $2n^2$ flops
3. Matrix-matrix multiply: $2n^2$ data, $2n^3$ flops

These are examples of level 1, 2, and 3 routines in *Basic Linear Algebra Subroutines* (BLAS). We like building things on level 3 BLAS routines.

q for naive matrix multiply

$q \approx 2$ (on board)

Better locality through blocking

Basic idea: rearrange for smaller working set.

```
for (I = 0; I < n; I += bs) {
  for (J = 0; J < n; J += bs) {
    block_clear(&(C(I,J)), bs, n);
    for (K = 0; K < n; K += bs)
      block_mul(&(C(I,J)), &(A(I,K)), &(B(K,J)),
                bs, n);
  }
}
```

Q: What do we do with “fringe” blocks?

q for naive matrix multiply

$q \approx b$ (on board). If M_f words of fast memory, $b \approx \sqrt{M_f/3}$.

Th: (Hong/Kung 1984, Irony/Tishkin/Toledo 2004): Any reorganization of this algorithm that uses only associativity and commutativity of addition is limited to $q = O(\sqrt{M_f})$

Note: Strassen uses distributivity...

Mission tedious-but-addictive

HW 1: You will optimize matrix multiply yourself!

- Find partners from different backgrounds
- Read the background material (tuning notes, etc)
- Use version control, automation, and testing wisely
- Get started early, and try not to over-do it!

Some predictions:

- You will make no progress without addressing memory.
- It will take you longer than you think.
- Your code will be rather complicated.
- Few will get anywhere close to the vendor.
- Some of you will be sold anew on using libraries!

Not all assignments will be this low-level.

Performance BLAS

Fastest way to good performance: use tuned libraries!

- DGEMM is one of the *Basic Linear Algebra Subroutines*
- It is "level 3" ($O(n^3)$ work on $O(n^2)$ data)
- Possible to make it fast, though not trivial
- Makes a good building block for higher-level operations!

Several fast BLAS options: OpenBLAS, ATLAS, MKL.

A little perspective

> *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."*
>
> *– C.A.R. Hoare (quoted by Donald Knuth)*

- Best case: good algorithm, efficient design, *obvious code*
- Speed vs readability, debuggability, maintainability?
- A sense of balance:
  - Only optimize when needed
  - Measure before optimizing
  - Low-hanging fruit: data layouts, libraries, compiler flags
  - Concentrate on the bottleneck
  - Concentrate on inner loops
  - Get correctness (and a test framework) first

Moving onward...

Class cluster basics

- Compute nodes are dual quad-core Intel Xeon E5504
- Nominal peak per core:

  2 SSE instruction/cycle $\times$

  2 flops/instruction $\times$

  2 GHz = 8 GFlop/s per core
- Caches:
  1. L1 is 32 KB, 4-way
  2. L2 is 256 KB (unshared) per core, 8-way
  3. L3 is 4 MB (shared), 16-way associative

  L1 is relatively slow, L2 is relatively fast.
- Inter-node communication is switched gigabit Ethernet

Cluster structure

Consider:

- Each core has vector parallelism
- Each chip has four cores, shares memory with others
- Each box has two chips, shares memory
- Five instructional nodes, communicate via Ethernet

How did we get here? Why this type of structure? And how does the programming model match the hardware?

Parallel computer hardware

Physical machine has *processors*, *memory*, *interconnect*.

- Where is memory physically?
- Is it attached to processors?
- What is the network connectivity?