§4.3 Scheduling

- block distribution
- cyclic "
- block-cyclic "

} false sharing issue

ex) matrix-vector multiplication

$$m\,\overset{n}{\boxed{A}} * (x) \Rightarrow (b)\}m$$

$$b_i = \sum_{j=0}^{n-1} A_{ij} * x_j \quad , \text{for all } \underline{i \in \{0 \ldots m-1\}}$$

- sequential code

as 1-D array like

```
— void sequential_mult(vector<double>& A,
                       vector<double>& x,
                       vector<double>& b, int m, int n)
  { for(row=φ ~ m-1)
      { accum = φ;
        for(int col=φ ~ n-1)
            accum += A[row*n + col] * x[col];
        b[row] = accum;
      }
  }
```

```
— int main(---)
  { n = ---;    int n=1<<15;
    m = ---;    int m=1<<15;
    vector<double> A(m*n);
    vector<double> x(n);
    vector<double> b(m);
    //initialize A, x here
    sequential_mult(A, x, b, m, n);
    :
  }
```

n=1 then leftshift 15 bits ⇒ 2^{15} = 32768

vector alloc. time overhead — initialize all entries

2.7 sec vs. 1.8×10^{-5} sec.

⇒ less alloc. time way using dynamic array.

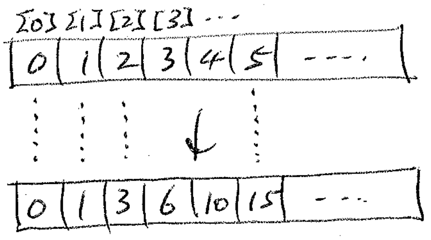
double* A = new double[m*n];

or, vector<no_init_t<double>> A(m*n);

$>g++ -O2 -std=c++11 DMV.cpp ↵

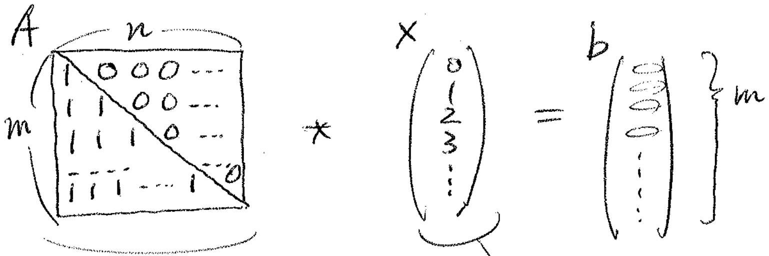
(dense matrix vector product)

— prefix sum computation with matrix-vector mult.

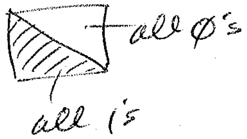


— given values — x vector

— prefix sum — b vector



initialize A matrix as lower triangular



initialize x vector with any given values

```
— for (int col=φ ~ n-1)
      x[col] = col;
```

```
— for (int row=φ ~ m-1)
      for (int col=φ~n-1)
          A[row*n + col] = row ≥ col ? 1 : φ;  //linear storage,
                                                 row-major layout.
```

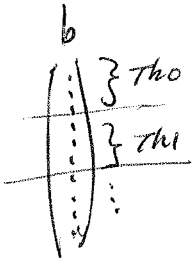
— macro for get_time — inline expansion

```
#include <sys/time.h>
  :
#define GET_TIME(now)\
{ struct timeval t; gettimeofday(&t, NULL);\
  now = t.tv_sec + t.tv_usec/1000000.0;\
}
  :
double start_time; ---
  :
GET_TIME (start_time);      double type
  :
GET_TIME (end_time);    —— cout << end_time - start_time;
```

— block distribution — version1 — main() creates threads.

DMV ex).

```
 ⋮
— void block-parallel_mult (int id, double A[], double x[], double b[],
        #include <cmath>     int m, int n, int num_threads)
{  int chunk = ceil((double) m × num_threads);  // – for not evenly
                                                      divisible case.
                                                  ⌈m/p⌉  assign
   int lower = id * chunk;                               1 more
   int upper = min(lower + chunk, m);                    to earlier
                                                         threads
   for (int row = lower ~ upper-1)
   { double accum = 0.0 ;
      for (int col = φ ~ n-1)
         accum += A[row*n+col] * x[col] ;
      b[row] = accum ;
   }
}
```



```
— int main(---)
 {  ⋮
   double* A = new double[m*n] ;
   double* x = new double[n] ;
   double* b = new double[m] ;
   //initialize A and x here

   vector<thread> threads;
   for (int id = φ ~ num_threads-1)
      threads.emplace_back (block_parallel_mult, id, A, x, b, m, n,
                                                      num_threads);
   for (auto& thread : threads)
      thread.join();  ——— //wait for all threads terminate

   for (int i = φ ~ m-1)
      cout << b[i] << endl;

   delete[] A ; delete[] x ; delete[] b ;
   return φ;
 }
```

— block distribution (DMV) — version2 — using lambda function

lambda func.

```
  ⋮
— void block_parallel_mult(double A[], double x[], double b[],
                           int m, int n, int num_threads)
{ auto block = [&] (int id) -> void
  { int chunk = ceil((double)m/num_threads);
    int lower = id * chunk;
    int upper = min(lower+chunk, m);
    for(int row = lower ~ upper-1)
    { double accum = 0.0;
      for (int col=0 ~ n-1)
         accum += A[row*n + col] * x[col];
      b[row] = accum;
    }
  };
  vector<thread> threads;
  for(int id=0 ~ num_threads-1)
     threads.emplace_back(block, id);
  for(auto& thread : threads)
     thread.join();
}

— int main(--)
{ ⋮
  double* A = new double[m*n];
  double* x = ---
  double* b = ---
  //init A and x here.
  block_parallel_mult(A, x, b, m, n, num_threads);

  for(int i=0 ~ m-1)
     cout << b[i] << endl;
  delete[] A; delete[] x; delete[] b;
  return 0;
}
```

- Lambda function usage

ex)

```
- auto add_one = [] (int& v) {return v+1;};
```

error

```
- int w=1;
  ~~add~~
  auto add_w = [ ] (int& v) {return v+w;};
```

} compile time error

? w is declared out of λ-func.

```
- int w=1;
  auto add_w = [w] (int& v) {return v+w;};
```

capture w by value

```
- int w=1;
  auto add_w = [&w] (int& v) {return v+w;};
```

capture w by reference

```
- auto add_w = [&] (int& v) {return v+w;};
```

capture everything accessed in λ-func. ~~as~~ by reference

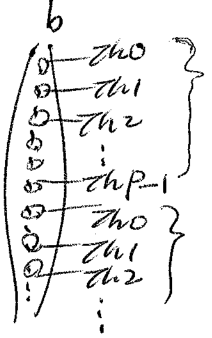
```
- auto add_w = [=] (int& v) {return v+w;};
```

capture everything accessed in λ-func. by value

4-15

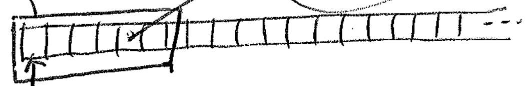
— cyclic distribution (DMV)

```
⋮
— void cyclic_parallel_mult (double A[], double X[], double b[],
                             int m, int n, int num_threads)
 { auto cyclic = [&] (int id) → void
λ-func
    { for (row = id ~ m-1, row += num_threads)
        { accum = 0.0;
          for (col = 0 ~ n-1)
              accum += A[row*n + col] * X[col];
          b[row] = accum;
        }
    };
    vector<thread> threads;
    for (id = 0 ~ num_threads - 1)
        threads.emplace_back (cyclic, id);
    for (auto& thread : threads)
        thread.join();
 }
```

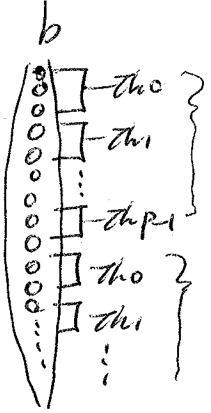


fine-grained cyclic

may cause ⇒ false sharing issue


cache line size
② reading this ele. → read from mem ∵ line is dirty
① 1st ele accessed — this makes the entire cache line dirty

→ sol: block-cyclic distribution



in fact,

block distr — block size = $\frac{m}{p}$

cyclic distr — block size = 1

extreme cases of block-cyclic distr.

— block size should be same as cache line size.

4-16

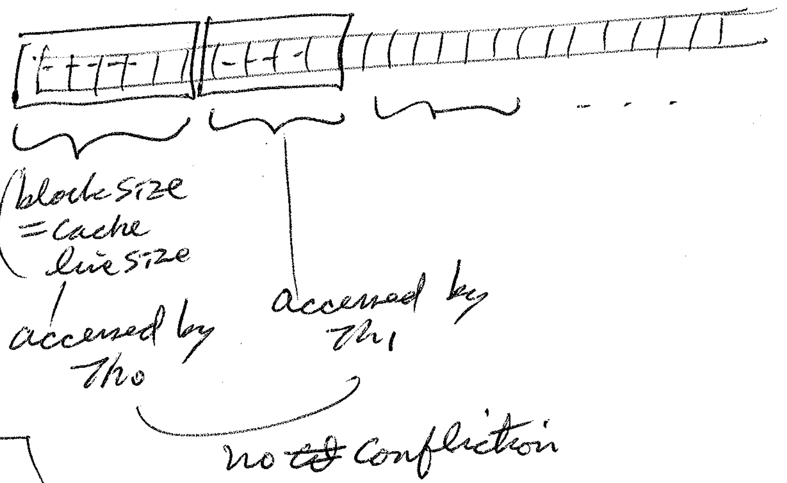
false sharing

```
ex) auto cyclic = [&] (int ~~&~~ id) → void
    { for (int row=id; row<m ; row+= num_threads)
       { b[row]=0;
         for (col=0 ~ n-1)
            b[row] += A[row*n+col] * X[col];
       }
    };
```

→ result is correct, but memory access overheads. (due to false sharing).

ex) b[0] is updated by Th0 on core0 ; — cache line in core0 is dirty

↓ b[1] is read by Th1 on core1 ; — core1 accesses mem for same cache line. (or updated cache)



— block size
- (too) small
  - → false sharing
  - → load balanced
- vs.
- big
  - → cache friendly access
  - → load imbalance

ex) Intel

cache line size 64 bytes

⇒ block size

good (c=16 for int
c=8 for 64bit data

ex) Th_i — local cache update and then, read same line → no problem

cache ~ up to date.

4-17

—block-cyclic distribution (DMV)

```
⋮

—void block_cyclic_parallel_mult(double A[], ... x[], ... b[],
                                 int m, int n, int num_threads, int chunk_size)
{ auto block_cyclic = [&] (int id) -> void
λfunc
  { int offset = id * chunk_size;        // chunksize = 64/8 = 8
                                            (cache line size)  (double type size)
    int stride = num_threads * chunk_size;
          next jump
    for (int lower = offset; lower < m; lower += stride)
    { int upper = min(lower + chunk_size, m);
      for (int row = lower; row < upper; row++)
      { double accum = 0.0;
        for (int col = 0 ~ n-1)
          accum += A[row*n + col] * x[col];
        b[row] = accum;
      }
    }
  }; //auto
  vector<thread> threads;
  for (int id=0 ~ num_threads-1)
    threads.emplace_back(block_cyclic, id);   // call λfunc.
  for (auto& thread : threads)
    thread.join();
}

—int main(...)
{ ⋮
                                                        8
  block_cyclic_parallel_mult(A, x, b, m, n, num_threads, 64/sizeof(double));
  for (i=0 ~ m-1)
       cout << b[i] << endl;
} ⋮
```