A Case Study on High Performance Matrix Multiplication

Andrés More - amore@hal.famaf.unc.edu.ar

Abstract

This document reviews a short case study on high performance matrix multiplication. Several algorithms were implemented and contrasted against a commonly used matrix multiplication library. Performance metrics including timings and performance were gathered allowing interesting graphical comparisons of the different approaches. The results showed that carefully optimized implementations are orders of magnitude above straightforward ones.

Contents

1 Introduction **1**
- 1.1 This Work . . . 1
- 1.2 The Subject . . . 1

2 Algorithms **1**
- 2.1 Simple . . . 2
- 2.2 Blocked . . . 2
- 2.3 Transposed . . . 2
- 2.4 BLAS Library . . . 2

3 Results **3**
- 3.1 Timings . . . 3
- 3.2 Performance . . . 4
- 3.3 Memory . . . 5

4 Parallelism **5**
- 4.1 Multi-threading . . . 5
- 4.2 Message Passing Interface . . . 5

5 Conclusions **6**

A The `mm` tool **6**
- A.1 Design . . . 6
- A.2 Usage . . . 6

1 Introduction

This section introduces this work and the topic under study.

1.1 This Work

This work was required to complete *Cluster Programming*, a course offered as part of the *Specialization on High Performance Computing and Grid Technology* offered at *Universidad Nacional de La Plata* (UNLP)[1].

Besides the course class-notes[2], other relevant material was consulted. This included Fernando Tinetti's work [1], a discussion about architecture-aware programming [2] and also performance evaluation on clustered systems [3].

1.2 The Subject

Matrix multiplication routines are widely used in the computational sciences in general, mostly for linear algebra. It is also heavily applied on scientific modeling in particular.

Timing performance is the main roadblock preventing models to became more complex and rich enough to match their real-world counterparts. More than in other areas, architecture-optimized code run orders of magnitude faster than naive implementations.

2 Algorithms

This section surveys several intuitive matrix multiplication algorithms. For each approach, code samples showing their implementation are included as

[1] `http://www.info.unlp.edu.ar`
[2] `http://ftinetti.googlepages.com/postgrado2008`

example for the reader. Square matrices will be assumed to simplify the discussion.

2.1 Simple

The formal definition of matrix multiplication is shown on equation 1. Note that it only implies the final value of each element of the result matrix; nothing is stated about how values are actually calculated.

$$(AB)_{ij} = \sum_{k=1}^{k=n} a_{i1}b_{1j} = a_{i1}b_{1j} + \ldots + a_{in}b_{nj} \quad (1)$$

The previous definition can be translated into the following C implementation.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i * n + j] +=
        a[i * n + k] * b[k * n + j];
```

2.2 Blocked

The first enhancing approach will be to apply the divide-and-conquer principle; it is expected that performing smaller matrix multiplications will optimize the usage of the memory cache hierarchy.

The implementation of the blocking approach is shown below.

```
for (i = 0; i < n; i += bs)
  for (j = 0; j < n; j += bs)
    for (k = 0; k < n; k += bs)
      block(&a[i * n + k],
              &b[k * n + j],
                &c[i * n + j], n, bs);
```

Where the definition of `block` is similar to the preceding approach, only that each block limits should be used.

```
for (i = 0; i < bs; i++)
  for (j = 0; j < bs; j++)
    for (k = 0; k < bs; k++)
      c[i * n + j] +=
        a[i * n + k] * b[k * n + j];
```

Note that the algorithm assumes for simplicity that matrix size should be a multiple of block size. As the best block size depends on the available cache, experimental results will be shown later.

2.3 Transposed

Another interesting approach taken from [2] is to transpose the second matrix. In this way, half of the cache misses when iterating over the columns of the second matrix will be avoided. On this case, each element definition is shown in equation 2.

$$(AB)_{ij} = \sum a_{i1}b_{1j}^T = a_{i1}b_{1j} + a_{i2}b_{2j} + ... + ab \quad (2)$$

The implementation on this case is very similar to our first approach; an additional step to rotate the matrix is added before the actual algorithm.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    tmp[i * n + j] = b[j * n + i];

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i * n + j] +=
        a[i * n + k] * tmp[j * n + k];
```

2.4 BLAS Library

The Basic Linear Algebra Subprograms[3] (BLAS) routines provide standard building blocks for performing basic vector and matrix operations [4]. Among its multiple routines, Level 3 BLAS routines perform general matrix-matrix operations as shown on equation 3. Where A, B, C are matrices and α, β vectors.

$$C \leftarrow \alpha AB + \beta C \quad (3)$$

The Automatically Tuned Linear Algebra Software (ATLAS) is a state-of-the-art, open source implementation of BLAS[4]. The library support single and multi-thread execution modes, feature that enable parallelism on systems with multiple processing units.

[3] `http://www.netlib.org/blas`
[4] `http://math-atlas.sourceforge.net`

```
cblas_dgemm(CblasRowMajor,
              CblasNoTrans,
                CblasNoTrans,
            n, n, n,
            1.0, a, n,
              b, n,
                0.0, c, n);
```

The `dgemm` BLAS routine perform general matrix-matrix multiplication on double floating points, and `cblas_dgemm` is a C language wrapper around the Fortran implementation.

The first three arguments define how the arrays are stored on memory and if any matrix need to be transposed, then the matrix sizes and at last the computation operands are provided.

3 Results

Results were gathered on a system featuring an Intel Core Duo T2400 @ 1.83GHz CPU and also 2GB of DDR2 RAM memory; the software layer included a *Gentoo* system using the 2.6.24 *Linux* kernel and the *GCC* C compiler version 4.1.2.

To get timings results the `gettimeofday` function is used. To get the quantity of the operations, the floating point operations per seconds (FLOPs) unit was used.

3.1 Timings

According to the matrix multiplication definiton, each result matrix element depends on the combination of a row and a column, therefore complexity will be $O(n^3)$. Although the best known theoretical algorithm is only $O(n^{2.58})$ [5]; it is too complex and only useful on very large sizes, not even suitable for computation on current hardware architectures.

Figures 2, 3, 5, 6 show the timings of the simple, block, transp and BLAS algorithms, respectively.

Normalised results are highlighted against the simple approach on figure 7. Taking the 8192×8192 matrix case as example, the performance gains based on the simple definition implementation is shown on table 3.1.

method	**seconds**	**gain**
simple	12000	$\times$ 1
blocked	8000	$\times$ 1.5
transposed	2500	$\times$ 5
blocked	500	$\times$ 25

Figure 1: 8192×8192 timings

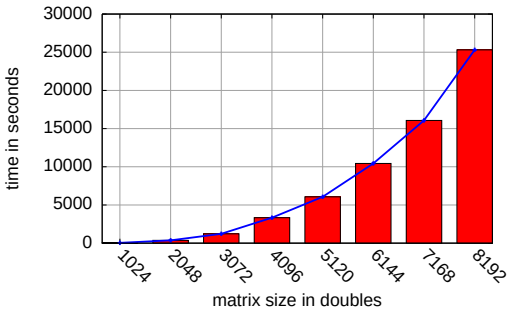

Figure 2: Simple method timings

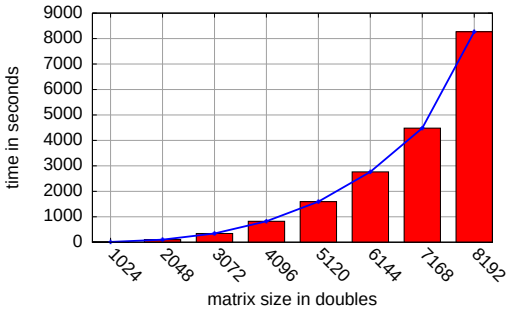

Figure 3: Block method timings

Figure 4 shows experimental results while changing the blocksize. The `sysconf(2)` POSIX interface allows to know this value at run-time, using the following code snippet is enough to find out the best block size.

```
size = sizeof(double);
cls = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);
bs = cls / size;
```

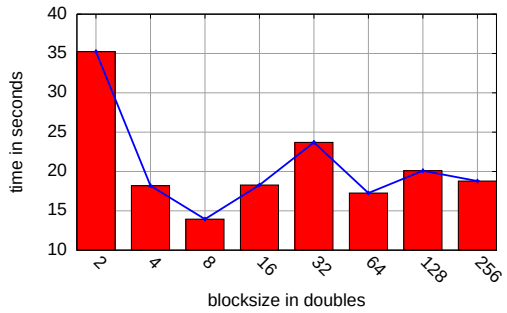

Figure 4: Block Size

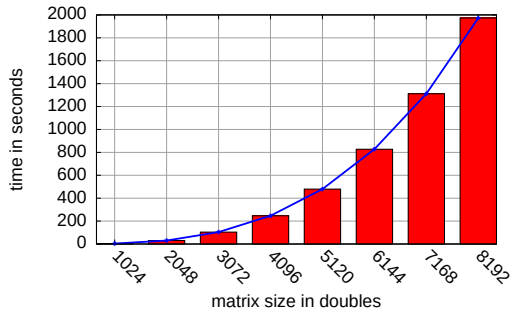

Figure 5: Transposed method timings

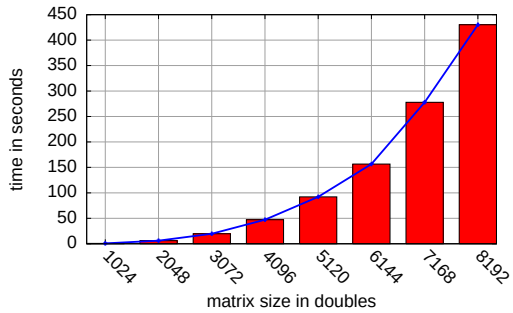

Figure 6: BLAS method timings

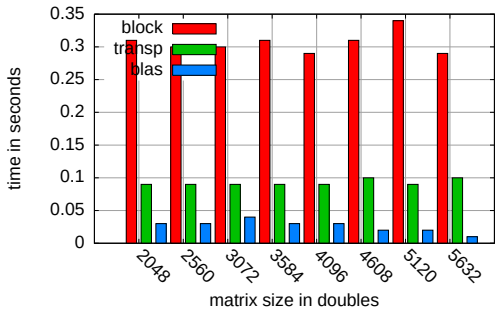

Figure 7: Timings

3.2 Performance

Matrix multiplication operations can be estimated using formula 4, were $n \times n$ is the size of the matrix and t the time used to process it.

$$flops = (n^3 - 2n^2)/t \tag{4}$$

Figure 3.2 compares the performance of the tried approaches. The table below highlights overall gains.

method	mflops	gain
simple	50	$\times$ 1
blocked	150	$\times$ 3
transposed	500	$\times$ 10
blas	1500	$\times$ 30

Figure 8: Average Mflops

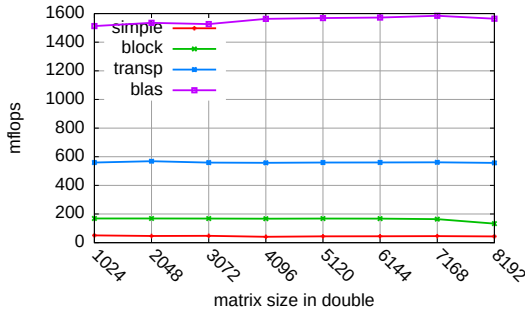

Figure 9: FLOPs Comparison

3.3 Memory

Equation 5. The second matrix used memory is freed after being transposed.

$$memory = 3 \times n^2 \times sizeof(double) \qquad (5)$$

4 Parallelism

Previous approaches were serial, this section details two parallel solutions.

4.1 Multi-threading

The ATLAS library can take advantage of multiple processing units, the multiplication is spread among all available cores in the system.

Figure 10 shows a comparison between serial and multi-threaded BLAS executions.

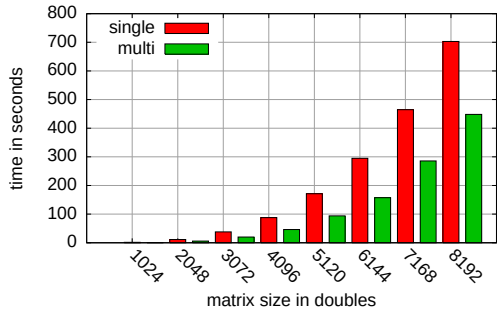

Figure 10: Single vs Multi threaded BLAS

4.2 Message Passing Interface

The Message Passing Interface[5] (MPI) provides point-to-point and collective communication among a set of processes distributed on a parallel system.

Similar to `socket(7)` programming using the `fork(2)` system call, the same binary is executed multiple times, and the programmer is in charge of the data communication protocol to be followed. Besides the actual computing of the matrix product, the code must handle the data distribution and results gathering.

[5] http://www.mpi-forum.org

The data distribution is as follows: each worker process knows which A rows to receive and compute given its own task id, after computation results are sent back to the master which is in charge of the accumulation of the results.

The master process implementation is summarized below.

```
for (i = 1; i < size; i++) {
  MPI_Send(&a[offset * n + 0],rows * n,
             MPI_DOUBLE, i, 1, 0);
  MPI_Send(&b[0], n * n,
             MPI_DOUBLE, i, 1, 0);
  offset += rows;
}

for (i = 1; i < size; i++) {
  offset = rows * (i - 1);
  MPI_Recv(&c[offset * n + 0], rows * n,
             MPI_DOUBLE, i, 2, 0, &status)
}
```

The worker process implementation is summarized below.

```
MPI_Recv (&a[0], rows * n,
          MPI_DOUBLE, 0, 1, 0, &status);
MPI_Recv (&b[0], n * n,
          MPI_DOUBLE, 0, 1, 0, &status);

for (k = 0; k < n; k++)
  for (i = 0; i < rows; i++)
    for (j = 0; j < n; j++)
      c[i * n + k] += a[i * n + j]
                    * b[j * n + k];

MPI_Send (&c[0], rows * n,
            MPI_DOUBLE, 0, 2, 0);
```

Although more intended to other types of systems [3], a naive implementation of MPI executed over the system under test was challenging. Figure 11 contains the timings gathered on such system.

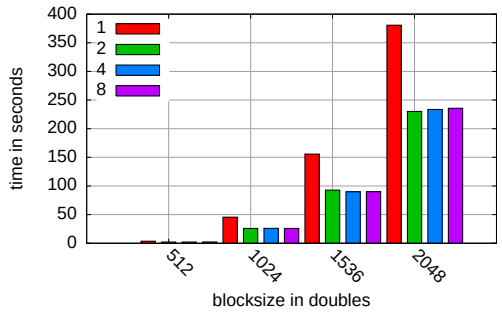

Figure 11: MPI timings according to quantity

5 Conclusions

Libraries are far better optimized, their outstanding performance against mere mortals implementations are unbelievable.

Binaries should be cache aware, as performance is highly dependant on this. Otherwise, recompilation will be needed.

Theorical approaches needs to be reviewed before the actual implementation, as the underlaying architecture can alter common assumptions.

Parallel programming is only hard, not impossible.

A The `mm` tool

A command line tool was implemented from scratch to gather timings and performance metrics of different methods of matrix multiplication[6]. Serial and parallel programming approaches were included in order to analyze scaling and speed up.

The tool used to gather all the results was made available under the GPLv2 license. Hopefully, it will help to understand high performance computing issues.

A.1 Design

The source code is divided into files as shown in table.

Reuse of open source software was done. glibc' Argp for GNU-like argument parsing. The autotools (autoconf, automake and libtool) set for building a portable source code package, easy of extract, compile and install on any supported system.

A.2 Usage

The tool support many method for matrix multiplication, square matrixes are randomly initialized and then multiplied using the requested approach.

The arguments accepted include matrix size and the method to apply, optionally a check using best known implementation can be used, and also an specific block size can be requested.

The output reported is enough to be processed as a comma-separated value (CSV). Each execution reports the following data: `method,size,time,mflops,block,processes`.

```
$ mm 8192 cblas --check
cblas,8192,702.911549,1564.129257,8,1
```

References

[1] F. Tinetti. *Parallel Computing in Local Area Networks*. PhD thesis, Universidad Autonoma de Barcelona, 2004.

[2] Ulrich Deeper. What every programmer should know about memory. Technical report, Red Hat, Inc., November 2007.

[3] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, September 1979.

[5] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 1990.

[6] `http://code.google.com/p/mm-matrixmultiplicationtool`