“Superluminal” FITS File Processing on Multiprocessors: Zero Time Endian Conversion Technique

SATOSHI EGUCHI
Astronomy Data Center, National Astronomical Observatory of Japan, 2-21-1, Osawa, Mitaka, Tokyo 181-8588, Japan; Postdoctoral Fellow of Japanese Virtual Observatory Project; satoshi.eguchi@nao.ac.jp

Received 2013 April 01; accepted 2013 April 18; published 2013 May 6

ABSTRACT. The FITS is the standard file format in astronomy, and it has been extended to meet the astronomical needs of the day. However, astronomical datasets have been inflating year by year. In the case of the ALMA telescope, a ∼TB-scale four-dimensional data cube may be produced for one target. Considering that typical Internet bandwidth is tens of MB/s at most, the original data cubes in FITS format are hosted on a VO server, and the region which a user is interested in should be cut out and transferred to the user (Eguchi et al. 2012). The system will equip a very high-speed disk array to process a TB-scale data cube in 10 s, and disk I/O speed, endian conversion, and data processing speeds will be comparable. Hence, reducing the endian conversion time is one of issues to solve in our system. In this article, I introduce a technique named “just-in-time endian conversion”, which delays the endian conversion for each pixel just before it is really needed, to sweep out the endian conversion time; by applying this method, the FITS processing speed increases 20% for single threading and 40% for multi-threading compared to CFITSIO. The speedup tightly relates to modern CPU architecture to improve the efficiency of instruction pipelines due to break of “causality”, a programmed instruction code sequence.

Online material: color figures

1. INTRODUCTION

The Flexible Image Transport System (FITS) is the standard data format for astronomical observed data even though they are the products of calibration pipelines or other systems. One FITS file can store multiple CCD images and photon event lists as tables, and this feature makes the FITS format prevail from the radio band to the X-ray band. Most archival datasets and source catalogs are currently provided as FITS files.

The original purpose of the FITS format was to transport digital astronomical images from one computer to another with a magnetic tape (Wells et al. 1981). There were no unified standards for computers at that time, and bit size assigned to a character or an integer was quite different from one model to another, even from the same makers. Thus, the authors created a machine-independent and future-expandable image format for data exchange, FITS. Since then the FITS format has been repeatedly extended to agree with astronomical needs of the day (e.g., Greisen & Harten 1981; Grosbol et al. 1988).

However, we will look at the issue of astronomical data inflation, rather than the format, in the years ahead; Atacama Large Millimeter/submillimeter Array (ALMA), which is the largest radio telescope built on the Chajnantor plateau in northern Chile, started observations last year. ALMA is estimated to generate ∼200 TB of observational raw data every year, and the volume of a processed four-dimensional data cube1 for one target may exceed ∼2 TB (Lucas et al. 2004). Furthermore, Large Synoptic Survey Telescope (LSST), a project scheduled for the 2020s, will generate 30 TB data every night.2 We need a system to help astronomers handle such large quantities of data.

Looking at the future, National Astronomical Observatory of Japan has been developing a large data-providing system for ALMA utilizing the technology of Virtual Observatory (VO) to share our outputs with global astronomical communities; all processed datasets (FITS files) are hosted on a VO server, and a user can select a cut-out region to download with a web-based graphical user interface (Eguchi et al. 2012, Paper I hereafter).

A prototype service is already publicly available,3 and I am working on its optimization now. The system must process a TB scale data cube in a few tens of seconds for users’ convenience; thus, it is planned to feature a very high-speed disk array,4 and disk I/O speed and data processing speed will be comparable. All the components of the system consist of Intel platforms, which feature a little endian format, while the FITS format is

1 = (Two-Dimensional Image) ⊗ (Spectrum) ⊗ (Polarization).
2 Please see http://www.lsst.org/lsst/science/development.
3 Please see http://jvo.nao.ac.jp/portal/alma/.
4 A system which consists of 16 striping solid state disks (SSDs) in the consumer products market effectively reaches ∼4 GB/s read/write performance.
big endian. For the interactive TB size FITS file processing system, the endian conversion time is not negligible.

In this article, I introduce a technique to make the endian conversion time apparently disappear, and to make the system much faster by multiprocessing. I describe the hardware and software configuration for evaluation in § 2, and compare endian conversion algorithms and their performance in § 3. In § 4, I examine the best timing for endian conversion, and discuss the performance increase by the conversion timing in § 5. Throughout the article I repeated measurements 100 times for each item and adopted their sample standard deviation (a square root of unbiased variance) as 1-σ statistical error, ignoring any systematic ones.

2. CONFIGURATION AND TEST DATA

Table 1 shows the hardware and software configuration used for verification of the method. I used two types of CPUs, Intel Core i7-2600 (for Machine A) and AMD FX-8350 (for Machine B), to prevent bias due to microarchitecture. Throughout the article, Intel Turbo Boost Technology (the former) and AMD Turbo CORE Technology (the latter) are disabled by BIOS for simplicity. In addition, Intel Hyper-Threading Technology (the former) is also disabled for the same reason. Thus Machine A and B make available 4 and 8 physical processors, respectively. The memory bandwidths and storage speeds were obtained using the following commands: `dd if=/dev/zero of=/dev/null bs=1G count=100`, and `hdparm -t (device)`, respectively.

The same software is installed in both computers: Ubuntu 12.04.1 LTS (amd64), a Debian based 64-bit Linux operating system, GNU Compiler Collection (GCC) version 4.6 for C/C++ compiler (gcc/g++), and CFITSIO version 3.310 for C language FITS library (Pence 2010). I applied the -O2 -pipe -Wall compile options to CFITSIO and other programs used in the article. The Streaming SIMD Extensions 2 (SSE2) code in CFITSIO was enabled since I built the library on a 64-bit Linux system, but the SSSE3 option was disabled since the SSSE3 instruction set is treated as an extension in the amd64 environment.

I used a false color mosaic image of the Carina Nebula obtained with the Hubble Space Telescope for test data. The image is in Tagged Image File Format (TIFF), so I converted it into a grayscale double precision FITS file with the convert command provided by ImageMagick. The size is 29,566 pixels in width and 14,321 pixels in height. The file volume is 3.4 GB (Fig. 1). Through the article, I put this FITS file on a tmpfs mounted on /run/shm to ensure that the file is always in memory for fast access. See Appendix A for the difference between tmpfs and ramdisk.

### 3. ENDIAN CONVERSION ALGORITHMS

3.1. Formalism

Let \( (b_1, b_2, \ldots, b_8) \) be a byte sequence of an internal expression of a 64-bit size value \( a \). The 64-bit endian conversion of \( a \) can be expressed with a permutation \( \sigma \) as

\[
d' = (b_{\sigma(1)}, b_{\sigma(2)}, \ldots, b_{\sigma(8)}),
\]

where

\[
\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}
\]

in Cauchy’s two-line notation, and \( \sigma^2 = 1 \) (Fig. 2).

3.2. Implementation

3.2.1. Byte Shuffle: Straightforward Implementation

A straightforward implementation of equations (1) and (2) can be written as follows:

---

6 There is no way to make the __SSE2__ macro undefined with 64-bit GCC, which switches the codes for SSE2 or otherwise in CFITSIO.
7 Please see http://hubblesite.org/newscenter/archive/releases/2007/16/image/a/.
8 Please see http://www.imagemagick.org/script/index.php.
```c
uint64_t byte_shuffle(uint64_t a)
{
    unsigned char *p = (unsigned char *)&a;
    unsigned char tmp;
    tmp = p[7]; p[7] = p[0]; p[0] = tmp;
    tmp = p[6]; p[6] = p[1]; p[1] = tmp;
    tmp = p[5]; p[5] = p[2]; p[2] = tmp;
    tmp = p[4]; p[4] = p[3]; p[3] = tmp;
    return a;
}
```

I will call this method “byte shuffle”. One will find a short discussion of another implementation of the byte shuffle algorithm in Appendix B.

### 3.2.2. Bit Shift

Another way to perform endian conversion is to use both bit shift and logical operations:

```c
uint64_t bit_shift(uint64_t a)
{
    unsigned char *p = (unsigned char *)&a;
    unsigned char tmp;
    tmp = p[7]; p[7] = p[0]; p[0] = tmp;
    tmp = p[6]; p[6] = p[1]; p[1] = tmp;
    tmp = p[5]; p[5] = p[2]; p[2] = tmp;
    tmp = p[4]; p[4] = p[3]; p[3] = tmp;

    return a;
}
```

Hereafter, I call this method “bit shift”.

### 3.2.3. BSWAP

Intel i486 and later processors have the `BSWAP` instruction, which converts the endian on a given 32-bit register. The instruction is extended in order to accept a 64-bit register in amd64 (Intel 2012). Furthermore, GCC versions 4.3 and later have a helper function to call the instruction, and its prototype is `uint64_t __builtin_bswap64(uint64_t x)`; I’ll call endian conversions utilizing this function “BSWAP”.

### 3.2.4. SSE2

SSE2 is a set of vector instructions for Intel platforms, which became a part of the default instruction set for the amd64 environment. The endian conversion code utilizing SSE2 can process two 64-bit values at once, and is written as follows:

```c
#include <emmintrin.h>
void sse2(uint64_t a[2])
{
    __m128i r0 = _mm_load_si128((__m128i *)a);
    //r0 <- a
}
```

Here is the schematic diagram of a permutation operator $\sigma$ for the endian conversion of a 64-bit value.
__m128i r1 = _mm_srli_epi16(r0, 8);
  //8-bit shifts towards right
__m128i r2 = _mm_slli_epi16(r0, 8);
  //8-bit shifts towards left
r0 = _mm_or_si128(r1, r2);
  //128-bit or operation on r1
  and r2
r0 = _mm_shufflelo_epi16(r0, _MM_SHUFFLE(0, 1, 2, 3));
  //byte shuffle for the lower
half of r0 register
r0 = _mm_shufflehi_epi16(r0, _MM_SHUFFLE(0, 1, 2, 3));
  //byte shuffle for the higher
half of r0 register
_mm_store_si128((__m128i *)a, r0);
  //a <- r0

Very similar codes can be found in CFITSIO and SLLIB/SFITSIO.9 I call these codes simply “SSE2” hereafter.

3.3. SSSE3

Another vector instruction set called “SSSE3” is available for Intel Core series and later CPUs. Utilizing this instruction set, one can perform endian conversion of two 64-bit values at one instruction. An example follows:

```c
#include <tmmintrin.h>
void ssse3(uint64_t a[2])
{
  static const __m128i mask = _mm_set_epi8(8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4, 5, 6, 7);
__m128i r1 = _mm_load_si128((__m128i *)a);
__m128i r2 = _mm_shuffle_epi8(r1, mask);
_mm_store_si128((__m128i *)a, r2);
}
```

There are similar codes in CFITSIO too. I call these codes “SSSE3” hereafter.

3.4. Benchmark

To see which method is fastest and how they behave toward parallelization, I performed a simple benchmark. In the benchmark, I reserved a double-type array whose number of elements was set to 29,566 × 14,321 = 423,414,686, the number of pixels in Figure 1, and filled the array with uniform random numbers of 32-bit resolution on [−1000, 1000] generated with Mersenne Twister (Matsumoto & Nishimura 1998). I adopt 16-byte memory alignment through the benchmark. One will find a short discussion about random alignment case in Appendix D.

3.4.1. Single Thread

The results are summarized in Table 2. For Machine A, all algorithms except for SSSE3 and byte shuffle process the test data in about 410 ms, while SSSE3 takes about 370 ms. On the other hand, for Machine B, all algorithms except for byte shuffle process the test data in about 600 ms, and the SSE2 algorithm is fastest in the all cases. The byte shuffle algorithm is slowest by one order of magnitude compared to the others.

3.4.2. Multi-Thread

I also examined the CPU-scalability of these algorithms. I adopted pthread for parallelization, and simply divided the array containing the test data into equal-sized segments so that the total number of the segments was equal to the number of threads. Then I assigned each thread to each segment.

Figure 3 shows the results. I also list the observed values for detailed comparison of the algorithms in Table 3 (for Machine A) and Table 4 (for Machine B). Except for the byte shuffle algorithm, I observed ≃10% performance gain for Machine A, and ≃40% up to four threads for Machine B with the four algorithms.

It seems strange that the memory bandwidth of Machine B is sufficient for the test data but the four algorithms show performance cutoff at four threads. I performed a detailed hardware benchmark utilizing LMbench,10 and found that context switching time and the latency of L2 cache memory normalized

| Machine | Bit shift (ms) | BSWAP (ms) | SSE2 (ms) | SSSE3 (ms) | Byte shuffle (ms) |
|---------|---------------|------------|-----------|------------|------------------|
| Machine A | 410±2 | 410±2 | 405±2 | 372.4±0.1 | 3190.3±0.6 |
| Machine B | 601.3±0.6 | 605.8±0.7 | 582.3±0.7 | 598±2 | 8056.3±0.3 |

Note: —The endian conversion time of 423,414,686 (=29,566 × 14,321) double-type elements with various algorithms.

### TABLE 2
ENDIAN CONVERSION TIME

---

There are similar codes in CFITSIO too. I call these codes simply “SSE2” hereafter.

---

9 Please see http://www.ir.isas.jaxa.jp/cyamauch/sl/index.html.
10 Please see http://www.bitmover.com/lmbench/.
in CPU cycles of Machine B are 2.4 times and 4.6 times, respectively, larger than those of Machine A. Hence, I conclude that there are some hardware bottlenecks in Machine B, which cause the plateau in Figure 3.

The behaviors of the four algorithms with respect to the number of threads are very similar, and I adopt the bit shift algorithm in the next section because of its compiler portability and identicalness to `BSWAP` (see Appendix C).

### 4. ENDIAN CONVERSION TIMING

A modern CPU has multiple arithmetic logic units (ALUs) and instruction pipelines to boost the operating rates of ALUs. As seen in the previous section, the hardware limitation lies just below the endian conversion time of single thread (Fig. 3, Machine A), preventing CPU scalability. This may lead to many holes (or “no operation” instructions) in the pipelines and reduce performance. If this is the case, shuffling instructions in source codes can produce improvement.

To verify this assumption, I disabled the endian conversion functionality in CFITSIO, I changed the `BYTESWAPPED` macros for i386 and amd64 architectures from `TRUE` into `FALSE` in `fitsio2.h`, and I commented out the code which CFITSIO uses to perform a runtime check to verify whether the machine endian definition by the above macro is consistent with the execution environment in `cfileio.c`, then I rebuilt the library. The patches for those files are shown in Appendix E.

I compare the following two methods:

1. One which loads the full test image (Fig. 1) from `tmpfs` (see § 2) into an array, then converts the endian by the parallelized bit shift algorithm (described in § 3.4.2) and sums up all the elements.

2. One which loads the full test image into an array and sums up all the elements by converting the endian one after another using the bit shift algorithm.

From here, I refer to method (1) as the “on ahead endian conversion method,” and to method (2) as the “just-in-time endian conversion method.”

The on ahead endian conversion method can be written as follows:

```c
{  double *v; // an array to store a FITS image
    size_t len; // the length of the array v
    // load a byte sequence from a FITS file into v here...
    // endian conversion
    for (size_t i = 0; i < len; ++i) {
```

| Number of threads | Bit shift (ms) | BSWAP (ms) | SSE2 (ms) | SSSE3 (ms) | Byte shuffle (ms) |
|-------------------|----------------|------------|-----------|-------------|-------------------|
| 1                 | 410±2          | 411±3      | 404±2     | 372.4±0.2   | 3191.8±0.7        |
| 2                 | 366±3          | 366±3      | 367±2     | 360.4±0.5   | 1605±2            |
| 3                 | 362±3          | 362±3      | 362±3     | 354.9±0.3   | 1082±6            |
| 4                 | 365±4          | 365±4      | 365±4     | 358.8±0.5   | 860±20            |

**Table 3**

The CPU-scalability of the Endian Conversion Algorithms on Machine A.

**Note.**—The CPU-scalability of 423,414,686 (≈29,566 × 14,321) double-type element endian conversion with various algorithms on Machine A.
and the just-in-time endian conversion method can be written as follows:

```c
double *v; // an array to store a FITS image
size_t len; // the length of the array v
// load a byte sequence from a FITS file into v here...
// image processing...
{
    // something...
    // one needs to refer the value of v[i] here
    // endian conversion
    uint64_t *p = (uint64_t *)&v[i];
    uint64_t a = bit_shift(*p);
    double *q = (double *)&a;
    v[i] = *q;
}
// process v here...
}
```

and that of the just-in-time endian conversion method is as follows:

```c
{
    double sum = 0.0;
    for (size_t i = 0; i < len; ++i) {
        // endian conversion
        uint64_t *p = (uint64_t *)&v[i];
        uint64_t a = bit_shift(*p);
        double *q = (double *)&a;
        v[i] = *q;
    }
    // summation
    sum += v[i];
}
```

Table 4 shows the CPU scalability of the endian conversion algorithms on Machine B.

| Number of threads | Bit shift (ms) | BSWAP (ms) | SSE2 (ms) | SSSE3 (ms) | Byte shuffle (ms) |
|-------------------|---------------|------------|----------|-----------|------------------|
| 1                 | 597.8±0.7     | 605.1±0.7  | 579.3±0.9| 597±2     | 8056.2±0.2       |
| 2                 | 484±1         | 499±1      | 484±1    | 495±1     | 4046±3           |
| 3                 | 429.9±0.8     | 445±3      | 432±1    | 440±1     | 2699±4           |
| 4                 | 413±3         | 412±2      | 413±3    | 419±7     | 2031±8           |
| 5                 | 418±3         | 417±1      | 422±2    | 424±2     | 1661±5           |
| 6                 | 412±1         | 411.5±0.8  | 413±1    | 416±1     | 1388±4           |
| 7                 | 412.3±0.9     | 412.2±0.5  | 411.9±0.6| 414.1±0.5 | 1192±4           |
| 8                 | 416.1±0.6     | 415.1±0.7  | 415.5±0.4| 414.5±0.4 | 1043±2           |

Note. — All conditions are same as Table 3.

4.1. Single Thread

I implemented both methods in a single thread and performed a benchmark. The code of the on ahead conversion method is as follows:

```c
for (size_t i = 0; i < len; ++i) {
    uint64_t *p = (uint64_t *)&v[i];
    uint64_t a = bit_shift(*p);
    double *q = (double *)&a;
    double x = *q;
    // use x instead of v[i] below
}
```

where `bit_shift()` is the endian conversion function defined in § 3.2.2.

In this section, I use summing up all the elements in the test image as an example of image processing.

Table 4

| Number of threads | Bit shift (ms) | BSWAP (ms) | SSE2 (ms) | SSSE3 (ms) | Byte shuffle (ms) |
|-------------------|---------------|------------|----------|-----------|------------------|
| 1                 | 597.8±0.7     | 605.1±0.7  | 579.3±0.9| 597±2     | 8056.2±0.2       |
| 2                 | 484±1         | 499±1      | 484±1    | 495±1     | 4046±3           |
| 3                 | 429.9±0.8     | 445±3      | 432±1    | 440±1     | 2699±4           |
| 4                 | 413±3         | 412±2      | 413±3    | 419±7     | 2031±8           |
| 5                 | 418±3         | 417±1      | 422±2    | 424±2     | 1661±5           |
| 6                 | 412±1         | 411.5±0.8  | 413±1    | 416±1     | 1388±4           |
| 7                 | 412.3±0.9     | 412.2±0.5  | 411.9±0.6| 414.1±0.5 | 1192±4           |
| 8                 | 416.1±0.6     | 415.1±0.7  | 415.5±0.4| 414.5±0.4 | 1043±2           |

Note that these codes are identical to those from original CFITSIO.
The results are summarized in Table 5. I obtained slightly faster ($\approx 5\%$) total processing time of $2.22 \pm 0.04$ s and $3.62 \pm 0.04$ s for Machines A and B, respectively, with the on ahead endian conversion method, while processing time with original CFITSIO is $2.38 \pm 0.04$ s and $3.79 \pm 0.03$ s for Machines A and B, respectively.

On the other hand, I obtained a significantly faster time of $1.85 \pm 0.04$ s and $3.05 \pm 0.03$ s for Machines A and B, respectively.

### Table 5

**The Data Processing Times with Two Different Method in Single Thread**

| Method                        | Machine   | FITS read time (s) | Sum up time (s) | Total time (s) |
|-------------------------------|-----------|--------------------|-----------------|----------------|
| On ahead endian conversion    | Machine A | 1.006±0.003       | 0.4198±0.0002   | 2.22±0.04      |
|                               | Machine B | 1.859±0.007       | 0.557±0.001     | 3.62±0.04      |
| Lazy endian conversion        | Machine A | 1.010±0.003       | 0.44003±0.00008 | 1.85±0.04      |
|                               | Machine B | 1.874±0.007       | 0.621±0.002     | 3.06±0.03      |

**Note.**—The total time with original CFITSIO is $2.38 \pm 0.04$ and $3.79 \pm 0.03$ for Machine A and B, respectively.

### Table 6

**The CPU-Scalability of On Ahead and Just-In-Time Endian Conversion Methods on Machine A**

| Method                        | Number of threads | FITS read time (s) | Sum up time (s) | Total time (s) |
|-------------------------------|-------------------|--------------------|-----------------|----------------|
| On ahead endian conversion    | 1                 | 1.445±0.003       | 0.4205±0.0001   | 2.38±0.04      |
|                               | 2                 | 1.404±0.003       | 0.220±0.004     | 2.1±0.1        |
|                               | 3                 | 1.401±0.003       | 0.181±0.002     | 2.11±0.09      |
|                               | 4                 | 1.403±0.003       | 0.177±0.002     | 2.09±0.03      |
|                               | 1                 | 1.048±0.003       | 0.4416±0.0009   | 2.0±0.1        |
|                               | 2                 | 1.051±0.003       | 0.2262±0.0005   | 1.8±0.1        |
| Just-in-time endian conversion | 3                 | 1.051±0.003       | 0.183±0.001     | 1.77±0.02      |
|                               | 4                 | 1.052±0.003       | 0.177±0.002     | 1.76±0.02      |

**Note.**—The endian conversion time is included in the FITS reading time for on ahead endian conversion method.

### Table 7

**The CPU-Scalability of On Ahead and Just-In-Time Endian Conversion Methods on Machine B**

| Method                        | Number of threads | FITS read time (s) | Sum up time (s) | Total time (s) |
|-------------------------------|-------------------|--------------------|-----------------|----------------|
| On ahead endian conversion    | 1                 | 2.526±0.005       | 0.5522±0.0006   | 3.76±0.03      |
|                               | 2                 | 2.403±0.005       | 0.299±0.003     | 3.39±0.01      |
|                               | 3                 | 2.347±0.005       | 0.220±0.002     | 3.253±0.010    |
|                               | 4                 | 2.326±0.005       | 0.193±0.003     | 3.203±0.008    |
|                               | 5                 | 2.334±0.006       | 0.205±0.003     | 3.224±0.010    |
|                               | 6                 | 2.326±0.005       | 0.189±0.003     | 3.20±0.01      |
|                               | 7                 | 2.325±0.006       | 0.178±0.002     | 3.19±0.01      |
|                               | 8                 | 2.334±0.005       | 0.174±0.003     | 3.19±0.01      |
|                               | 1                 | 1.914±0.003       | 0.582±0.002     | 3.19±0.05      |
|                               | 2                 | 1.917±0.003       | 0.328±0.002     | 2.934±0.009    |
|                               | 3                 | 1.916±0.004       | 0.242±0.002     | 2.853±0.008    |
|                               | 4                 | 1.916±0.004       | 0.205±0.002     | 2.812±0.009    |
| Just-in-time endian conversion | 5                 | 1.918±0.003       | 0.205±0.001     | 2.816±0.009    |
|                               | 6                 | 1.918±0.003       | 0.1948±0.0009   | 2.801±0.010    |
|                               | 7                 | 1.917±0.004       | 0.185±0.001     | 2.791±0.010    |
|                               | 8                 | 1.917±0.003       | 0.178±0.003     | 2.781±0.010    |

**Note.**—The endian conversion time is included in the FITS reading time for on ahead endian conversion method.
respectively, which corresponds to \( \approx 25\% \) performance gain, using the just-in-time endian conversion method.

### 4.2. Multi-Thread

I made both methods multithreaded by utilizing OpenMP APIs for its simple implementation. The code for the just-in-time endian conversion method, for example, is below:

```c
double sum = 0.0;
#pragma omp parallel for reduction
(+:sum) schedule (auto)
for (size_t i = 0; i < len; ++i) {
    // endian conversion
    uint64_t *p = (uint64_t *)&v[i];
    uint64_t a = bit_shift(*p);
    double *q = (double *)&a;
    // summation
    sum += *q;
}
```

On the other hand, I could not find the best parameters in the OpenMP APIs for the endian conversion routine for the on ahead conversion method, and hence I applied OpenMP only to the summation routine, and adopted the pthread-based parallelization described in § 3.4.2 for the endian conversion routine in the on ahead conversion method; the number of the threads for OpenMP was set to that for the endian conversion.

The results obtained with these programs are summarized in Table 6 (for Machine A), Table 7 (for Machine B), Figure 4 (for the on ahead endian conversion method), and Figure 5 (for the just-in-time endian conversion method). Note that the endian conversion time of the on ahead endian conversion method is included in the FITS reading time. The total time to perform the same tasks with the original CFITSIO in single thread is superimposed on these figures as a dotted line: 2.38 ± 0.04 s for Machine A and 3.79 ± 0.03 s for Machine B.

For the on ahead endian conversion method, the total time slightly scales the number of threads and becomes faster than original CFITSIO, while the file reading time (including endian conversion time) seems to be less scalable. The scalability of the total time is mostly due to the summation routine, and the parallelization of the endian conversion has little impact due to the hardware limit seen in § 3.4.2.

On the other hand, for the just-in-time endian conversion method, the total time is interestingly smaller than that of original CFITSIO even for single thread. The summation routine seems to be scalable almost in the full range, while the total time scales up to four threads.

## 5. DISCUSSION

### 5.1. Performance Analysis of the Simple Summation Codes

There is a well-known equation to estimate the increase by parallelization, Amdahl’s law (Amdahl 1967):

\[
T_{\text{parallel}} = \left(1 - P\right) + \frac{P}{N + \alpha} T_{\text{single}},
\]

where \( T_{\text{single}} \) and \( T_{\text{parallel}} \) represent processing time in single thread and multi-thread cases, respectively, \( P \) is the ratio of codes which parallelization methods are applied to,\(^{12} \) \( N \) is

---

\(^{11}\) Please see http://openmp.org/wp/.

\(^{12}\) Hardware bottlenecks are included in the \( 1 - P \) term.
the number of threads, and $\alpha$ is the overhead caused by parallelization.

To quantify the performance increase of the on ahead endian conversion method and the just-in-time endian conversion method, I performed a model fitting to the total time of both methods with equation (3). I found that $\alpha \sim O\left(10^{-30}\right)$ during the fitting, thus I fixed $\alpha$ at 0. The results are summarized in Table 8 and Figure 6. The increasing rates of performance compared to original CFITSIO ($T_{\text{single}} = 2.38 \pm 0.04 \text{ s}$ for Machine A and $T_{\text{single}} = 3.79 \pm 0.03 \text{ s}$ for Machine B) are also listed in the table.

The figure shows that the above results are explained well by Amdahl’s law, and that the on ahead endian conversion method for single thread has almost the same performance as original CFITSIO. In fact, these two agree with each other in $\lesssim 5\%$ errors according to the table. The table also suggests that multi-threading boosts this method up to about 20%. Considering the parallelization rate $P \approx 16\%$, one cannot expect further speed up by multi-threading in $N \gtrsim 4$. This suggests that the bottlenecks of other hardware disrupt order in the instruction pipelines and lead to the decrease of operating ratios of ALUs.

On the other hand, the just-in-time endian conversion method is 20% faster than both original CFITSIO and the single thread version of the on ahead method, surprisingly. This seems as if the endian conversion process has disappeared. In the parallelized case, the just-in-time conversion method is 40% faster than the other methods in single thread. However, the performance increase by multi-threading can be expected only in $N \lesssim 4$ since the parallelization rate $P \approx 16\%$, due to the hardware bottlenecks mentioned above.

For further investigation, I fitted the summation time of these methods with equation (3) to investigate the impact of the endian conversion codes in the summation routine on performance; there are endian conversion codes in the summation routine of the just-in-time endian conversion method, but not in that of the on ahead endian conversion method. The results are summarized in Table 9 and Figure 7. I found that the parallelization rate $P \approx 85\%$ in both cases, and that the ratio of $T_{\text{single}}$ of the just-in-time endian conversion method to that of the on ahead method $r = T_{\text{single}}(\text{Just-in-Time})/T_{\text{single}}(\text{On Ahead})$ was equal to $r = 1.02 \pm 0.05$ for Machine A and $r = 1.03 \pm 0.04$ for Machine B. There is no overhead of endian conversion in

| Method                      | Machine  | $T_{\text{single}}$ (s) | $P$    | $\chi^2$ (d.o.f.) | Increase rate of performance single thread | Multi-thread |
|-----------------------------|----------|-------------------------|--------|-------------------|-------------------------------------------|--------------|
| On ahead endian conversion  | Machine A| 2.377 ± 0.009           | 0.164 ± 0.006 | 0.09 (2) | 1.00 ± 0.02 | 1.20 ± 0.02 |
|                             | Machine B| 3.68 ± 0.05            | 0.16 ± 0.02 | 49(6) | 1.03 ± 0.02 | 1.22 ± 0.03 |
|                             | Machine A| 2.02 ± 0.05            | 0.17 ± 0.03 | 0.5 (2) | 1.18 ± 0.04 | 1.42 ± 0.07 |
| Just-in-time endian conversion | Machine B| 3.13 ± 0.02            | 0.130 ± 0.009 | 8.5 (6) | 1.21 ± 0.01 | 1.39 ± 0.02 |

**Table 8**

The fitting results of the total processing time

---

**Table 8**

The fitting results of the total processing time with respect to two different endian conversion methods with Amdahl’s law and their increase rate of performance compared with original CFITSIO ($T_{\text{single}} = 2.38 \pm 0.04 \text{ s}$ for Machine A and $T_{\text{single}} = 3.79 \pm 0.03 \text{ s}$ for Machine B). The errors are 1-$\sigma$ confidence limits for a single parameter.
Since the shift of $r$ from unity is not significant statistically. Thus I conclude that endian conversion is such a simple operation for a modern CPU that the bottlenecks of other hardwares disrupt order in the instruction pipelines; to prevent the disruption, the endian conversion should be done just before a value is referred.

5.2. Application to ALMAWebQL

Until now I only investigated the performance increase of summing up all the elements in a large FITS file by the just-in-time endian conversion method. In this subsection, I apply the method to ALMAWebQL, our interactive web viewer for ALMA data cubes described in Paper I, to obtain more realistic benchmark data. For realistic and fair comparison, the SSE2 boosted endian conversion codes in CFITSIO are enabled for the on ahead endian conversion method, while there is no SSE2 code in the just-in-time endian conversion method. ALMA data cubes do not currently contain information of polarization, and they are simple three-dimensional FITS files (Fig. 8). For image extraction, one must integrate the cube along the spectral direction; for spectrum extraction, one must convolute all spatial information. I measured the time required to complete these computations in single thread with variously-sized data on Machine A. The results for image extraction are summarized in Table 10, and those for spectrum extraction are summarized in Table 11. From these tables, I obtain

$$T_{\text{On Ahead}} = (1.7 \pm 0.1) \times 10^{-3} \left( \frac{V}{\text{MB}} \right) - (0.013 \pm 0.007) \, \text{s},$$

(4)

$$T_{\text{Just-in-Time}} = (1.27 \pm 0.04) \times 10^{-3} \left( \frac{V}{\text{MB}} \right) + (0.021 \pm 0.003) \, \text{s}$$

(5)

for image extraction, and

![Fig. 6.—The fitting results of on ahead and just-in-time endian conversion methods with respect to total processing time with Amdahl’s law. The red dashed and blue dash dotted lines correspond to the law for on ahead and just-in-time endian conversion methods, respectively. See the electronic edition of the PASP for a color version of this figure.](image)

| Method                        | Machine  | $T_{\text{single}}$ (ms) | $P$         | $\chi^2$ (d.o.f.) |
|-------------------------------|----------|--------------------------|-------------|------------------|
| On ahead endian conversion    | Machine A| 420 ± 1                  | 0.82 ± 0.04 | 127.9 (2)        |
|                               | Machine B| 551 ± 6                  | 0.81 ± 0.02 | 491.1 (6)        |
| Just-in-time endian conversion| Machine A| 430 ± 20                 | 0.92 ± 0.06 | 913.4 (2)        |
|                               | Machine B| 570 ± 20                 | 0.80 ± 0.02 | 312.7 (6)        |

Note.—$T_{\text{single}}(\text{Just-in-Time})/T_{\text{single}}(\text{On Ahead}) = 1.02 \pm 0.05$ (for Machine A), 1.03 ± 0.04 (for Machine B).
\[ T(\text{On Ahead}) = (1.252 \pm 0.007) \times 10^{-3} \left( \frac{V}{\text{MB}} \right) + (0.0008 \pm 0.0004) \text{ s}, \]  \hspace{1cm} (6)

\[ T(\text{Just-in-Time}) = (0.85 \pm 0.02) \times 10^{-3} \left( \frac{V}{\text{MB}} \right) + (0.028 \pm 0.002) \text{ s} \]  \hspace{1cm} (7)

for spectrum extraction, where \( T(\text{On Ahead}) \) and \( T(\text{Just-in-Time}) \) represent the time with the on ahead and just-in-time endian conversion methods, respectively, and \( V \) is the file size in the MB unit (Fig. 9). Hence, the just-in-time endian conversion method in single thread is \( \geq 20\% \) faster than the on ahead conversion method, boosted by SSE2 above \( V \geq 200 \text{ MB} \). This demonstrates that the just-in-time endian conversion method can be very powerful when one performs convolution.
and stacking of very large images, which are very common analysis techniques in optical band, obtained with future large telescopes.

5.3. Data Types

In this article, I only treated a double precision FITS file, but one can expect almost the same results for float and LONG data types, which correspond to BITPIX = −32 and 32, respectively; as demonstrated in Appendix C, the bit_shift() function is compiled into the BSWAP instruction. The amd64 architecture can handle both 32 bit and 64 bit operation codes and their operands seamlessly. On the other hand, for byte and short data types, there may be little advantage in applying the just-in-time endian conversion method since the BSWAP instruction cannot take a 16 bit value as its operand, and up-casting into a 32 bit integer always occurs in arithmetic operations in both cases.

6. SUMMARY

The FITS format was originally developed to exchange digital astronomical datasets from one computer to another, but the progress of computation power and software technology enables one to now process FITS files through web browsers. In addition, data size has been inflating year by year, and it will exceed ∼TB in the year ahead. To handle such big FITS files with web applications, the endian conversion time from the FITS native to the machine cannot be negligible, and a solution for this problem is required.

In this article, I compared the features of four typical endian conversion algorithms under a multi-thread environment, and found that the bit shift algorithm was suitable for parallelization. Then I examined the best timing for endian conversion under a multi-thread environment. I found that one should postpone the endian conversion until a value is really referred in a program, because endian conversion is so simple for a modern CPU that the bottlenecks of other hardwares disrupt order in the instruction pipelines, which leads to the decrease of operating ratio of ALUs. In fact, by applying this method to the process of loading a 3.4 GB FITS file and summing up all the elements, performance increased by 20% for single thread and 40% for multi-thread compared to CFITSIO, which corresponded to ∼600 ms, and one can detect the speed-up. No overhead of endian conversion was found on the summation routine; hence one can sweep the endian conversion time out of his/her codes. Note that parallelization of this method peaked out in four threads in the experiment.

CPU vendors introduce various techniques, such as speculative execution and branch prediction, to improve the efficiency of instruction pipelines; an executed instruction code sequence is separate from a programmed one. In this context, modern CPUs partially break “causality”, a programmed instruction code sequence, and gain speed. the just-in-time endian conversion method utilizes such boosting technology. There is nothing new in the method, but it is a small step toward handling the large amounts of astronomical data generated by the next generation of telescopes.

I greatly appreciate Dr. Chisato Yamauchi, who is my colleague and the author of SLLIB/SFITSIO,13 for rewarding discussions.

13 SFITSIO is a lightweight FITS library for C/C++, providing modern APIs.

Fig. 9.—The image (left) and spectrum (right) extraction time with respect to file size of ALMA data cube. The red dashed and blue dash dotted lines correspond to the best fits for on ahead and just-in-time endian conversion method, respectively. See the electronic edition of the PASP for a color version of this figure.
APPENDIX A

TMPFS AND RAMDISK

Both tmpfs and ramdisk are data spaces allocated in memory. One has to specify the size in advance for ramdisk, while one does not necessarily set the size for tmpfs in advance since it is under control of a virtual memory manager and shares swap space.

When an application asks the operating system for memory blocks and when there does not remain sufficient physical memory space, the memory manager first swaps out the files on tmpfs. Tmpfs is an ideal space to put temporary files which one requires very fast access to.

APPENDIX B

ANOTHER IMPLEMENTATION OF BYTE SHUFFLE ALGORITHM

One can also implement the byte shuffle algorithm as follows:

```c
uint64_t byte_shuffle2(uint64_t a)
{
    unsigned char *p = (unsigned char *)&a;
    uint64_t b;
    unsigned char *q = (unsigned char *)&b;

    q[0] = p[7];
    q[1] = p[6];
    q[2] = p[5];
    q[3] = p[4];
    q[4] = p[3];
    q[5] = p[2];
    q[6] = p[1];
    q[7] = p[0];

    return b;
}
```

The number of assignments of the codes \(\frac{8}{8}\) is smaller than that shown in the main part of this article \(\frac{12}{8}\), and one would expect further performance improvement. I disassembled both codes compiled with the \(-02\) option and obtained following:

0000000000000000 <byte_shuffle>:
0: 49 89 fa mov %rdi,%r10
  3: 49 89 f8 mov %rdi,%r8
  6: 89 fe mov %edi,%esi
  8: 48 89 f9 mov %rdi,%rcx
b: 89 fa mov %edi,%edx
d: 48 89 f8 mov %rdi,%rax
10: 40 88 7c 24 ff mov %dil,-0x1(%rsp)
15: 48 c1 e8 20 shr $0x20,%rax
19: 49 c1 ea 38 shr $0x38,%r10
1d: 49 c1 e8 30 shr $0x30,%r8
21: 66 c1 ee 08 shr $0x8,%si

25: 48 c1 e9 28 shr $0x28,%rcx
29: c1 ea 10 shr $0x10,%edx
2c: c1 ef 18 shr $0x18,%edi
2f: 44 88 54 24 f8 mov %r10b,-0x8(%rsp)
34: 40 88 74 24 fe mov %sil,-0x2(%rsp)
39: 44 88 44 24 f9 mov %r8b,-0x7(%rsp)
3e: 88 54 24 fd mov %d1,-0x3(%rsp)
42: 88 4c 24 fa mov %cl,-0x6(%rsp)
46: 40 88 7c 24 fc mov %dil,-0x4(%rsp)
4b: 88 44 24 fb mov %al,-0x5(%rsp)
4f: 48 8b 44 24 f8 mov -0x8(%rsp),%rax
54: c3 retq

0000000000000000 <byte_shuffle2>:
0: 49 89 fa mov %rdi,%r10
  3: 49 89 f9 mov %rdi,%r9
  6: 49 89 f8 mov %rdi,%r8
  9: 48 89 fe mov %rdi,%rsi
c: 89 f9 mov %edi,%ecx
e: 89 fa mov %edi,%edx
10: 89 f8 mov %edi,%eax
12: 49 c1 ea 38 shr $0x38,%r10
16: 49 c1 e9 30 shr $0x30,%r9
1a: 66 c1 e8 08 shr $0x8,%eax
1e: 49 c1 e8 28 shr $0x28,%r8
22: 48 c1 ee 20 shr $0x20,%rsi
26: c1 e9 18 shr $0x18,%ecx
29: c1 ea 10 shr $0x10,%edx
2c: 44 88 54 24 f8 mov %r10b,-0x8(%rsp)
31: 44 88 4c 24 f9 mov %r9b,-0x7(%rsp)
36: 44 88 44 24 fa mov %r8b,-0x6(%rsp)
3b: 40 88 74 24 fb mov %sil,-0x5(%rsp)
40: 88 4c 24 fc mov %cl,-0x4(%rsp)
44: 88 54 24 fd mov %d1,-0x3(%rsp)
48: 88 44 24 fa mov %al,-0x2(%rsp)
4c: 40 88 7c 24 ff mov %dil,-0x1(%rsp)
There are fewer assignments in `byte_shuffle2()` (=8) than in `byte_shuffle()` (=12); however, the former binary codes are longer than the latter ones. Hence one cannot expect more performance gain with the codes.

### APPENDIX C

**BIT SHIFT ALGORITHM AND BSWAP INSTRUCTION**

The bit shift endian conversion code is actually identical to `BSWAP` instruction when compiled with the optimization option of -O2. The disassembled code obtained with the `objdump -d` command is below:

```
0000000000000000 <bit_shift>:
  0: 48 89 f8 mov %rdi,%rax
  3: 48 0f c8 bswap %rax
  6: c3 retq
```

### APPENDIX D

**ENDIAN CONVERSION ALGORITHMS AND MEMORY ALIGNMENT**

It is ensured that the leading memory address (alignment) of an array is always in multiplies of 16 (16-byte alignment) in amd64 architecture. However, if one would like to read a file in multi-thread, he/she must make a copy of the file image in memory. In such a case, the alignment is not always 16-byte. Thus, I performed the benchmark described in § 3.4 (single thread case) but made an alignment of the array random number.

For the benchmark, I modified `_mm_load_si128()` and `_mm_store_si128()` in the SSE2 codes into `_mm_loadu_si128()` and `_mm_storeu_si128()`, respectively, to make the code operable. The results are summarized in Table 12. The trend found in § 3.4.1 is roughly true in this case, although the all algorithms are slightly slower (within a few %) than in the 16-byte alignment case. Hence one does not have to be nervous about memory alignment.

### APPENDIX E

**THE PATCHES FOR CFITSIO**

A1.E.1. `fitsio2.h`

```c
*** fitsio2.h.org 2013-03-08 14:19:49
.560538980 +0900
- fitsio2.h 2013-01-15 14:43:10
 .000000000 +0900
***************
*** 96,102 ****
 #elif defined(__ia64__) \\
 defined(__x86_64__) \\
 /* Intel itanium 64-bit PC, or AMD opteron 64-bit PC */
! #define BYTESWAPPED TRUE
 #define LONGSIZE 64
 #elif defined(__SX) /* Nec SuperUx */
 #elif defined(__ia64__) ||
 defined(__x86_64__) \\
 /* Intel itanium 64-bit PC, or AMD opteron 64-bit PC */
 ! /* #define BYTESWAPPED TRUE */
 #define BYTESWAPPED FALSE
 #define LONGSIZE 64
 #elif defined(__SX) /* Nec SuperUx */
***************
*** 169,175 ****
 /* generic 32-bit IBM PC */
 #define MACHINE IBMPC
! #define BYTESWAPPED TRUE
```

Note: The endian conversion time of 423,414,686 (~29,566 x 14,321) double-type elements with various algorithms in case that memory alignment is random.

**TABLE 12**

| Machine    | Bit shift (ms) | BSWAP (ms) | SSE2 (ms) | SSSE3 (ms) | Byte shuffle (ms) |
|------------|---------------|------------|-----------|------------|------------------|
| Machine A  | 413±2         | 416±3      | 416±5     | 377±2      | 3220±10          |
| Machine B  | 624±7         | 641±9      | 602±9     | 590±3      | 8320±90          |

Note: The endian conversion time of 423,414,686 (~29,566 x 14,321) double-type elements with various algorithms in case that memory alignment is random.
#elif defined(__arm__)
— 170,178 —
/* generic 32-bit IBM PC */
#define MACHINE IBMPC
! /* #define BYTESWAPPED TRUE */
! #define BYTESWAPPED FALSE
! #elif defined(__arm__)

A2.E.2. cfileio.c

*** cfileio.c.org 2013-03-08 14:20:09
.052539296 +0900
— cfileio.c 2013-01-16 19:57:46.000000000
+0900
**************
*** 3763,3769 ****
}
/* test for correct byteswapping. */
!
! u.ival = 1;
if ((BYTESWAPPED && u.cval[0] != 1) ||
(BYTEWAPPED == FALSE && u.cval[1] != 1))
**************
*** 3776,3782 ****
PFUNLOCK;
return(1);
}
! /* test that LONGLONG is an 8 byte integer */

REFERENCES
Amdahl, G. M. 1967, AFIPS ’67 (Spring), 483
Eguchi, S., Kawasaki, W., Shirasaki, Y., et al. 2012, preprint (arXiv:1211.3790)
Greisen, E. W., & Harten, R. H. 1981, ApJS, 44, 371
Grosbøl, P., Harten, R. H., Greisen, E. W., & Wells, D. C. 1988, ApJS, 73, 359
Intel Corporation 2012, Intel 64 and IA-32 Architectures Software Developer’s Manuals, Volume 2A, 3–78
Lucas, R., Richer, J., Shepherd, D., Testi, L., Wright, M., & Wilson, C.
2004, ALMA Memo, 501
Matsumoto, M., & Nishimura, T. 1998, ACM Trans. Model. Comp. Simul., 8, 3
Pence, W. D. 2010, Astrophysics Source Code Library, 10001
Rohland, C. 2001, tmpfs Documentation, http://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt
Wells, D. C., Greisen, E. W., & Harten, R. H. 1981, ApJS, 44, 363

2013 PASP, 125:565–579