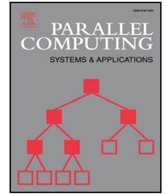
Parallel Computing 75 (2018) 1–10

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco



Computation of the 100 quadrillionth hexadecimal digit of π on a cluster of Intel Xeon Phi processors

Daisuke Takahashi

Center for Computational Sciences, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki, 305-8577, Japan

ARTICLE INFO

Article history:
Received 18 December 2016
Revised 5 December 2017
Accepted 16 February 2018
Available online 17 February 2018

Keywords:
BBP-type formula
Modular exponentiation
Cluster of Intel Xeon Phi processors

ABSTRACT

This paper presents the computation of a specific hexadecimal digit of π by using a Bailey–Borwein–Plouffe (BBP)-type formula on a cluster of Intel Xeon Phi processors. The BBP-type formula can be computed using modular exponentiation. We use Montgomery multiplication for the modular multiplication, which is the most time-consuming part of the modular exponentiation. We vectorize multiple modular exponentiations and multiple integer divisions by using Intel Advanced Vector Extensions 512 (Intel AVX-512) instructions. A parallel implementation of the BBP-type formula is presented. The 100 quadrillionth hexadecimal digit of π was computed on a 512-node cluster of Intel Xeon Phi processors with an elapsed time of 641 h 29 min that includes the time required for verification.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Many computations of mathematical constants (e.g., π and e) have been performed with high precision [1–4]. Mathematical constants are computed from their series expansion, such as:

$$\pi = 16 \arctan \frac{1}{5} - 4 \arctan \frac{1}{239}, \quad \arctan \frac{1}{q} = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)q^{2k+1}}. \tag{1}$$

Brent [5] and Salamin [6] independently discovered an algorithm to compute π. This algorithm has quadratic convergence. Borweins discovered cubic and higher order algorithms for π [7,8].

In 2009, Bellard computed π up to about 2.7 trillion decimal digits in about 131 days using the following Chudnovsky's formula [9] and an Intel Core i7 processor [1].

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!\,(13591409 + 545140134k)}{(3k)!\,(k!)^3\, 640320^{3k+3/2}}. \tag{2}$$

In 2013, Yee and Kondo computed π up to 12.1 trillion decimal digits in about 94 days using Chudnovsky's formula and dual Intel Xeon E5-2690 processors [3]. In 2016, Trueb computed π up to about 22.4 trillion decimal digits in about 105 days using Yee's program and quad Intel Xeon E7-8890 v3 processors [4].

An algorithm for the computation of a specific hexadecimal digit of π was discovered by Bailey, Borwein, and Plouffe in 1995 (hereafter called the BBP formula) [10,11]. The BBP formula enables computation of a specific bit in π without computing all the previous bits. PiHex [12] was a distributed computing project that used Bellard's BBP-type formula to

E-mail address: daisuke@cs.tsukuba.ac.jp

https://doi.org/10.1016/j.parco.2018.02.002
0167-8191/© 2018 Elsevier B.V. All rights reserved.

compute the quadrillionth bit of π. This required 250 CPU-years and used 1734 computers from 56 different countries. Sze computed the two quadrillionth bit of π in 23 days using Bellard's formula and a 1000-node cluster [13]. Karrels computed the ten quadrillionth hexadecimal digit of π in 88 days using 51 machines with GPUs [14].

Bailey et al. [15] stated that the main motivation for computing and analyzing π and other mathematical constants is to explore whether and why these sequences are random numbers. Even just storing the values of 100 quadrillion ($= 10^{17}$) hexadecimal digits of π requires a storage capacity of 50 PB. As of November 2017, the total storage capacity of Sunway TaihuLight [16], is ranked first in the TOP500 list [17], is 20 PB. Thus, in order to know the 100 quadrillionth hexadecimal digit of π, we have no other choice than to compute a few hexadecimal digits of π starting at position 10^{17} by using the BBP-type formula.

The Intel Many Integrated Core Architecture (Intel MIC Architecture) has emerged as an important computational accelerator in high-performance computing systems. The Knights Landing processor [18] is the second-generation Intel Xeon Phi product. To best of our knowledge, an implementation of the BBP-type formula on a cluster of Intel Xeon Phi processors has not yet been reported. In this paper, we present the use of a BBP-type formula on a cluster of Intel Xeon Phi processors to compute a specific hexadecimal digit of π.

The remainder of this paper is organized as follows. Section 2 presents the BBP-type formula that we use. Section 3 describes modular exponentiation and Montgomery multiplication. In Section 4, we propose an implementation of the BBP-type formula on a cluster of Intel Xeon Phi processors. The performance results are then presented in Section 5. Section 6 presents the computation of the 100 quadrillionth hexadecimal digit of π on a 512-node cluster of Intel Xeon Phi processors. Finally, Section 7 presents some concluding remarks.

2. BBP-type formula

The BBP formula [10,11] is as follows:

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right). \tag{3}$$

Bellard's formula [19] is approximately 43% faster than the BBP formula, and it is as follows:

$$\pi = \frac{1}{2^6}\sum_{k=0}^{\infty} \frac{(-1)^k}{2^{10k}}\left(-\frac{2^5}{4k+1} - \frac{1}{4k+3} + \frac{2^8}{10k+1} - \frac{2^6}{10k+3} - \frac{2^2}{10k+5} - \frac{2^2}{10k+7} + \frac{1}{10k+9}\right). \tag{4}$$

Consider computing a few hexadecimal digits of π starting at position $n+1$ for a positive integer n. Note that this is equivalent to computing $\{16^n\pi\}$, where $\{\cdot\}$ denotes the fractional part [11].

From Eq. (4), we have

$$\begin{aligned}\{16^n\pi\} = \{&-\{16^nS(4,1,-1)\} - \{16^nS(4,3,-6)\} + \{16^nS(10,1,2)\} - \{16^nS(10,3,0)\} \\ &-\{16^nS(10,5,-4)\} - \{16^nS(10,7,-4)\} + \{16^nS(10,9,-6)\}\},\end{aligned} \tag{5}$$

where

$$S(m,j,l) = \sum_{k=0}^{\infty}(-1)^k \frac{2^l}{2^{10k}(mk+j)}. \tag{6}$$

We note that

$$\begin{aligned}\{16^nS(m,j,l)\} &= \left\{\left\{\sum_{k=0}^{\lfloor(4n+l)/10\rfloor}(-1)^k\frac{2^{4n+l-10k}}{mk+j}\right\} + \sum_{k=\lfloor(4n+l)/10\rfloor+1}^{\infty}(-1)^k\frac{2^{4n+l-10k}}{mk+j}\right\} \\ &= \left\{\left\{\sum_{k=0}^{\lfloor(4n+l)/10\rfloor}(-1)^k\frac{2^{4n+l-10k} \bmod (mk+j)}{mk+j}\right\} + \sum_{k=\lfloor(4n+l)/10\rfloor+1}^{\infty}(-1)^k\frac{2^{4n+l-10k}}{mk+j}\right\}.\end{aligned} \tag{7}$$

The BBP-type formula requires a bit complexity of $O(n \log nM(\log n))$ where $M(d)$ is the complexity of multiplying d-bit integers [10].

3. Modular exponentiation and Montgomery multiplication

A key operation of the BBP-type formula is the modular exponentiation $2^{4n+l-10k} \bmod (mk+j)$ in the numerator of the first summation in Eq. (7). Many algorithms for modular exponentiation have been proposed [20,21]. Algorithm 1 shows the left-to-right binary modular exponentiation for $x = a^e \bmod N$ [20]. This algorithm consists of the modular squaring $x^2 \bmod N$ and the modular multiplication $ax \bmod N$. For evaluating the numerator of the first summation in equation (7), we only have to consider the case of $x = 2^e \bmod N$. In this case, the modular multiplication $ax \bmod N$ in line 6 of Algorithm 1 can be replaced by the left shift $x \ll 1$ and the conditional subtraction $x - N$ when $x \geq N$. Algorithm 2 shows the left-to-right binary modular exponentiation for $x = 2^e \bmod N$. The m-ary method [20,21] and the sliding window method [20] are known

Algorithm 1 Left-to-right binary modular exponentiation for $x = a^e \bmod N$ [20].

Input: a, e, N positive integers
Output: $x = a^e \bmod N$
1: let $(e_l e_{l-1} \ldots e_1 e_0)$ be the binary representationof e, with $e_l = 1$
2: $x \leftarrow a$
3: **for** i **from** $l - 1$ **downto** 0 **do**
4: $x \leftarrow x^2 \bmod N$
5: **if** $e_i = 1$ **then**
6: $x \leftarrow ax \bmod N$
7: **return** x.

Algorithm 2 Left-to-right binary modular exponentiation for $x = 2^e \bmod N$.

Input: e, N positive integers
Output: $x = 2^e \bmod N$
1: let $(e_l e_{l-1} \ldots e_1 e_0)$ be the binary representationof e, with $e_l = 1$
2: $x \leftarrow 2$
3: **for** i **from** $l - 1$ **downto** 0 **do**
4: $x \leftarrow x^2 \bmod N$
5: **if** $e_i = 1$ **then**
6: $x \leftarrow x \ll 1$
7: **if** $x \geq N$ **then**
8: $x \leftarrow x - N$
9: **return** x.

Algorithm 3 Montgomery multiplication algorithm [22].

Input: A, B, N such that $0 \leq A, B < N,\ \beta > N,\ \gcd(\beta, N) = 1,$
$\mu = -N^{-1} \bmod \beta$
Output: $C = AB\beta^{-1} \bmod N$ such that $0 \leq C < N$
1: $C \leftarrow AB$
2: $q \leftarrow \mu C \bmod \beta$
3: $C \leftarrow (C + qN)/\beta$
4: **if** $C \geq N$ **then**
5: $C \leftarrow C - N$
6: **return** C.

to reduce the number of modular multiplications for the modular exponentiation $x = a^e \bmod N$. However, the number of modular squaring operations for these methods is equal to that for the left-to-right binary modular exponentiation. Thus, the left-to-right binary modular exponentiation is sufficiently for computing $x = 2^e \bmod N$ in Algorithm 2.

The modular exponentiation $2^{4n+l-10k} \bmod (mk + j)$ in the numerator of the first summation in Eq. (7) must be performed exactly. The upper limit of the hexadecimal digit n is determined by $(10\lfloor(4n+2)/10\rfloor + 9)^2 < 2^{113}$ when IEEE 754 128-bit floating-point arithmetic is used. In this case, the upper limit of the hexadecimal digit n is $\lfloor\sqrt{2} \cdot 2^{54}\rfloor - 1 \approx 2.55 \times 10^{16}$, and thus it is not sufficiently precise for computing the 100 quadrillionth ($= 10^{17}$th) hexadecimal digit of π. On the other hand, the upper limit of the hexadecimal digit n is determined by $(10\lfloor(4n+2)/10\rfloor + 9)^2 < 2^{128}$ when 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication is used. In this case, the upper limit of the hexadecimal digit n is $2^{62} - 3 \approx 4.61 \times 10^{18}$. Thus, we use the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication in the modular exponentiation.

The most time-consuming part in Algorithm 2 is the modular squaring $x^2 \bmod N$. It includes modulo operations, which are slow due to the integer division process. However, Montgomery multiplication [22], shown as Algorithm 3, is known to avoid this. In Montgomery multiplication, it is necessary that $\gcd(\beta, N) = 1$. Here, since β is a positive power of two integer and all denominators of Eq. (4) are odd numbers, we can use Montgomery multiplication in the modular exponentiation. We note that Sze [13] and Karrels [23] also used Montgomery multiplication for the modular exponentiation when using Bellard's formula to compute the two quadrillionth bit of π and the quadrillionth hexadecimal digit of π, respectively.

Let **MontgomeryMul**(A, B) be the Montgomery multiplication, as in Algorithm 3. The result of a Montgomery multiplication **MontgomeryMul**(A, B) is not $AB \bmod N$ but rather $AB\beta^{-1} \bmod N$ [24]. To obtain a correct result at the end of the modular exponentiation, we need to make a pre-multiplication **MontgomeryMul**(A, β^2) and a post-multiplication **MontgomeryMul**$(A^e, 1)$ [24]. The post-multiplication is equivalent to computing $A^e\beta^{-1} \bmod N$. The modular exponentiation $x = 2^e \bmod N$ can be transformed into $x = 2^{e-\log_2 \beta}\beta \bmod N$ when β is a positive power of two integer. Thus, the post-multiplication can be avoided by replacing e with $e - \log_2 \beta$ for the modular exponentiation $x = 2^e \bmod N$ when $e > \log_2 \beta$.

Algorithm 4 Newton's method for the modular multiplicative inverse N^{-1} mod 2^{64} [25].

Input: N such that $0 < N < 2^{64},\ 2 \nmid N$
Output: $\mu = N^{-1}$ mod 2^{64}
1: $\mu \leftarrow \{(3N) \oplus 2\}$ mod 2^{64}
2: **for** i **from** 1 **to** 4 **do**
3: $\quad \mu \leftarrow \mu(2 - N\mu)$ mod 2^{64}
4: **return** μ.

Algorithm 5 Modular exponentiation for $x = 2^e$ mod N with Montgomery multiplication on 64-bit processors.

Input: e, N such that $0 < e < 2^{64},\ 0 < N < 2^{63},\ 2 \nmid N$
Output: $x = 2^e$ mod N
1: **if** $e < 65$ **then**
2: $\quad x \leftarrow 2^e$ mod N
3: $\quad$ **return** x
4: $e \leftarrow e - 64$
5: let $(e_l e_{l-1} \ldots e_1 e_0)$ be the binary representationof e, with $e_l = 1$
6: $x \leftarrow 2^{65}$ mod N
7: **for** i **from** $l - 1$ **downto** 0 **do**
8: $\quad x \leftarrow$ **MontgomeryMul**(x, x)
9: $\quad$ **if** $e_i = 1$ **then**
10: $\quad\quad x \leftarrow x \ll 1$
11: $\quad\quad$ **if** $x \geq N$ **then**
12: $\quad\quad\quad x \leftarrow x - N$
13: **return** x.

In Algorithm 3, the modular multiplicative inverse $\mu = -N^{-1}$ mod β is precomputed. Although the modular multiplicative inverse can be computed by the extended Euclidean algorithm, Newton's method is more efficient when β is a power of two [20,25]. Algorithm 4 shows Newton's method for the modular multiplicative inverse N^{-1} mod 2^{64} [25]. Here, $(3N)\oplus 2$ is the correct multiplicative inverse modulo 2^5 (5 bits) [25], where $\oplus$ denotes the exclusive or operation. Since Newton's method has quadratic convergence, four iterations are sufficient to obtain N^{-1} mod 2^{64}. Algorithm 5 shows the modular exponentiation for $x = 2^e$ mod N with the Montgomery multiplication on 64-bit processors.

4. Implementation of the BBP-type formula on a cluster of Intel Xeon Phi processors

Montgomery multiplication algorithms using vector instructions have been proposed [26,27]. Another approach is to use the SIMD instructions to compute multiple Montgomery multiplications in parallel [27]. We vectorized the multiple Montgomery squaring operations with Intel Advanced Vector Extensions 512 (Intel AVX-512) instructions [28]. In this scheme, multiple numerators in the first summation of equation (7) can be computed in parallel.

Although the x86_64 `mulq` instruction performs the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication, the Intel AVX-512 instruction set only supports `vpmuludq` instruction, which performs 32-bit $\times$ 32-bit$\rightarrow$ 64-bit unsigned integer multiplication. Thus, we use the radix-β interleaved Montgomery multiplication algorithm [22,27], which is shown as Algorithm 6. In the radix-2^{32} interleaved Montgomery multiplication, there is some overflow in the 64-bit unsigned

Algorithm 6 The radix-β interleaved Montgomery multiplication algorithm [22,27].

Input: A, B, N, μ such that$A = \sum_{i=0}^{m-1} a_i \beta^i,\ 0 \leq a_i < \beta,\ 0 \leq A, B < N,$
$\beta^{m-1} \leq N < \beta^m,\ \gcd(\beta, N) = 1,\ \mu = -N^{-1}$ mod β
Output: $C = AB\beta^{-m}$ mod N such that $0 \leq C < N$
1: $C \leftarrow 0$
2: **for** i **from** 0 **to** $m - 1$ **do**
3: $\quad C \leftarrow C + a_i B$
4: $\quad q \leftarrow \mu C$ mod β
5: $\quad C \leftarrow (C + qN)/\beta$
6: **if** $C \geq N$ **then**
7: $\quad C \leftarrow C - N$
8: **return** C.

integer addition. There are no carry bits for the 512-bit wide SIMD registers (ZMM0–ZMM31) on the Intel AVX-512 [28]. Although it is possible to detect the overflow by using branches, there will be performance degradation on processors that

```
void vsqrmod(uint64_t *c, uint64_t *a, uint64_t *N, uint32_t *mu)
/*  Compute c[:] = (a[:] * a[:] * 2^-62) mod N[:].
    We need mu[:] = -N[:]^-1 mod 2^31. */
{
  uint64_t t0, t1, t2;
  uint32_t a0, a1, N0, N1, q;
  int i;

#pragma ivdep
#pragma vector aligned
  for (i = 0; i < VLEN; i++) {
    a0 = a[i] & 0x7FFFFFFF;
    a1 = a[i] >> 31;
    N0 = N[i] & 0x7FFFFFFF;
    N1 = N[i] >> 31;
    t0 = (uint64_t) a0 * a0;
    t1 = (uint64_t) a0 * a1;
    t2 = (uint64_t) a1 * a1;
    q = ((uint32_t) t0 * mu[i]) & 0x7FFFFFFF;
    t0 = ((t0 + (uint64_t) q * N0) >> 31) + (t1 + (uint64_t) q * N1);
    t1 += t0 & 0x7FFFFFFF;
    t2 += t0 >> 31;
    q = ((uint32_t) t1 * mu[i]) & 0x7FFFFFFF;
    t1 = ((t1 + (uint64_t) q * N0) >> 31) + (t2 + (uint64_t) q * N1);
    c[i] = min(t1, t1 - N[i]);
  }
}
```

Fig. 1. Vectorized multiple Montgomery squaring operations of 62-bit integers.

have SIMD instructions. Thus, we use the radix $\beta = 2^{31}$ of Algorithm 6 to avoid overflow. In this case, the upper limit of the hexadecimal digit n is $2^{60} - 2 \approx 1.15 \times 10^{18}$.

In lines 6 and 7 of Algorithm 6, the performance is also degraded by the conditional subtraction $C - N$ when $C \geq N$. For multiple Montgomery multiplications, such conditional subtractions can be vectorized with Intel AVX-512 `vmovups`, `vpcmpuq`, `vmovdqu64`, and `vpsubq` instructions by the Intel C Compiler. On the other hand, min/max operations are effective for avoiding branches. The conditional subtraction can be replaced by the operation $\min(C, C - N)$ for 64-bit unsigned integer values C and N with the wrap-around two's complement arithmetic. Although the Intel Advanced Vector Extensions 2 (AVX2) instruction set [29] does not support the min instruction for 64-bit unsigned integers, the Intel AVX-512 instruction set supports the `vpminuq` instruction for 64-bit unsigned integers. This scheme is faster than conditional subtraction on Intel Xeon Phi processors.

Fig. 1 shows the vectorized multiple Montgomery squaring operations for 62-bit integers. This corresponds to $A = B$, $\beta = 2^{31}$, and $m = 2$ in Algorithm 6. In Fig. 1, `#pragma ivdep` instructs the compiler to ignore assumed vector dependencies, and `#pragma vector aligned` instructs the compiler to use aligned data movement instructions for all array references when vectorizing. The performance of the vectorized multiple Montgomery squaring operations in Fig. 1 depends on the vector length. According to preliminary experimental results, the vector length `VLEN` in Fig. 1 is determined to be equal to 40 on Intel Xeon Phi processors. In this case, the vectorized multiple Montgomery squaring operations can be performed by using only the 512-bit wide SIMD registers except for memory access for input/output arrays. The vectorized multiple Montgomery squaring operations can be further optimized using the Intel AVX-512 intrinsic functions [30].

When vectorizing multiple modular exponentiations for $x = 2^e \bmod N$, multiple modulo operations in line 6 of Algorithm 5 can be vectorized using the `_mm512_rem_epu64()` intrinsic function in the Short Vector Math Library (SVML) [30]. The number of iterations l in line 7 of Algorithm 5 may be different for multiple exponents, such as $e = (e_l e_{l-1} \ldots e_1 e_0)_2$. The exponent $4n + l - 10k$ of the modular exponentiation in equation (7) monotonically decreases. Thus, if the number of iterations l for the first element of the exponent vector is greater than that for the last element of the exponent vector, the scalar version of the modular exponentiation is performed. Since the number of calls for the scalar version for the nth hexadecimal digit of π is $O(\log n)$ at most, the overhead for scalar processing is almost negligible. In lines 9 and 10 of Algorithm 5, the statement "**if** $e_i = 1$ **then** $x \leftarrow x \ll 1$" degrades the performance because it introduces a branch. However, because e_i is 0 or 1, this branch can be omitted by performing the left shift $x \leftarrow x \ll e_i$. Such multiple left shifts can be vectorized with Intel AVX-512 `vpsllvq` instruction by the Intel C Compiler. Also, in lines 11 and 12 of Algorithm 5, the conditional subtraction $x - N$ when $x \geq N$ can be replaced by the operation $\min(x, x - N)$, similar to what was done for the vectorized multiple Montgomery squaring operations shown in Fig. 1.

The range of the absolute value of each fraction in equation (7) is [0, 1). Thus, the division and summation of equation (7) can be performed by using fixed-point arithmetic. In our implementation, we used 128-bit unsigned fixed-point

Algorithm 7 192-bit by 64-bit unsigned integer division based on the exact division algorithm.

Input: x, N, r, μ such that$0 \le x < N$, $0 < N < 2^{64}$, $2 \nmid N$,
$r = (2^{128} \cdot x) \bmod N$, $\mu = N^{-1} \bmod 2^{64}$
Output: $q = \lfloor (2^{128} \cdot x)/N \rfloor$
1: **if** $r = 0$ **then**
2: **return** 0
3: $q_0 \leftarrow (-r \cdot \mu) \bmod 2^{64}$
4: $q_1 \leftarrow [\{(2^{64} - 1) -$ **umulh**$(N, q_0)\} \cdot \mu] \bmod 2^{64}$
5: $q \leftarrow q_1 \cdot 2^{64} + q_0$
6: **return** q.

arithmetic. According to the Q format [31], a UQ128 number has 128 fractional bits, and its range is $[0, 1 - 2^{-128}]$. UQ128 fixed-point arithmetic can be implemented using 128-bit unsigned integer arithmetic. Both GCC [32] and Clang [33] provide the `__uint128_t` extension for 128-bit unsigned integer arithmetic. Although the Intel C compiler also supports the `__uint128_t` extension, a statement which contains the `__uint128_t` variables cannot be automatically vectorized. Thus, the summation of Eq. (7) is only performed with scalar processing. For negative values, we can use the two's complement representation. In this scheme, neither floating-point arithmetic nor the extraction of the fractional part of a floating-point number is necessary. Also, it is not necessary to convert between the fraction and its hexadecimal form [10]. Furthermore, when using 128-bit unsigned fixed-point arithmetic, the result does not depend on the computation order. With Bellard's formula, it correctly yields the first 25 hexadecimal digits for the ten quadrillionth hexadecimal digit of π.

For evaluating Eq. (7) with 128-bit unsigned fixed-point arithmetic, we need to compute 192-bit by 64-bit unsigned integer division $\lfloor (2^{128} \cdot x)/N \rfloor$, where $x = 2^e \bmod N$, and e, and N are positive integers such that $0 < N < 2^{64}$. Since the x86_64 `divq` instruction performs 128-bit by 64-bit unsigned integer division, the 192-bit by 64-bit unsigned integer division can be implemented by the x86_64 `divq` instruction twice. However, the x86_64 `divq` instruction is a slow operation, and the Intel AVX-512 instruction set does not support 128-bit by 64-bit unsigned integer division. Although Karrels used Newton's method for 192-bit by 64-bit integer division, the division and summation of Eq. (7) dominated 24% of the runtime for the quadrillionth hexadecimal digit of π [23].

If we know the remainder $(2^{128} \cdot x) \bmod N$ in advance, we can use the exact division algorithm [34] for the 192-bit by 64-bit unsigned integer division. This remainder can be easily computed by replacing e with $e + 128$ for $x = 2^e \bmod N$. Since the value of e is $4n + l - 10k$ in Eq. (7), the additional cost for precomputing $(2^{128} \cdot x) \bmod N$ is almost negligible when e is sufficiently large. Algorithm 7 shows 192-bit by 64-bit unsigned integer division based on the exact division algorithm. In Algorithm 7, the modular multiplicative inverse $\mu = N^{-1} \bmod 2^{64}$ is precomputed. The modular multiplicative inverse can be computed by using Algorithm 4. In Algorithm 7, the function **umulh** returns the upper 64-bit half of the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication. Multiple 192-bit by 64-bit unsigned integer divisions based on the exact division algorithm can be vectorized by using the Intel AVX-512 instructions. The 128-bit quotient of the 192-bit by 64-bit unsigned integer division is stored in a `__uint128_t` datatype variable. By using this scheme, the division and summation of Eq. (7) dominate only about 7% of the runtime for the quadrillionth hexadecimal digit of π. On the other hand, the modular exponentiation in the numerator of the first summation in Eq. (7) dominates about 92% of the runtime for the quadrillionth hexadecimal digit of π.

The BBP-type formula is embarrassingly parallel except for the final summation of results. Thus, it can be easily parallelized by using both OpenMP and MPI. By using the OpenMP `schedule(guided)` clause for main loop scheduling, the chunk sizes are initially large, and they then decrease in order to better handle load imbalances between iterations. The partial sum of each MPI process is computed using the OpenMP `reduction` clause. The total sum is then computed using the MPI reduce operation in a block-cyclic distribution.

5. Performance results

In order to evaluate the implemented parallel computation of a specific hexadecimal digit of π, we measured both the single-node performance and the multi-node performance. We averaged the elapsed times obtained from 10 executions of the nth hexadecimal digit of π by using Bellard's formula.

5.1. Single-node performance

The performance was measured on an Intel Xeon E5-2690 v4, an Intel Xeon Phi 5110P, and an Intel Xeon Phi 7250. Both scalar and vector versions were implemented. The original programs were written in C with OpenMP. The scalar version uses the Montgomery squaring routine with x86_64 inline assembly. The vector version uses a routine with multiple Montgomery squaring operations with the Intel AVX2, Intel Initial Many Core Instructions (Intel IMCI) [35], and Intel AVX-512 intrinsic functions on the Intel Xeon E5-2690 v4, the Intel Xeon Phi 5110P, and the Intel Xeon Phi 7250, respectively. The specifications

Table 1
Specification of the machines.

	Intel Xeon processor	Intel Xeon Phi coprocessor	Intel Xeon Phi processor
Number of cores	14	60	68
Number of threads	28	240	272
CPU	Intel Xeon E5-2690 v4 Broadwell-EP 2.6 GHz	Intel Xeon Phi 5110P Knights Corner 1.053 GHz	Intel Xeon Phi 7250 Knights Landing 1.4 GHz
L1 Cache (per core)	I-Cache: 32 KB D-Cache: 32 KB	I-Cache: 32 KB D-Cache: 32 KB	I-Cache: 32 KB D-Cache: 32 KB
L2 Cache	256 KB (per core)	512 KB (per core)	1 MB (shared between two cores)
L3 Cache (shared)	35 MB	N/A	N/A
Main Memory	DDR4-2400 256 GB	GDDR5 8 GB	MCDRAM 16 GB + DDR4-2400 96 GB
OS	Linux 3.10.0–327.36.3. el7.x86_64	Linux 2.6.38.8+mpss3.6	Linux 3.10.0–327.22.2.el7. xppsl_1.4.1.3272.x86_64
C compiler	Intel C Compiler Version 17.0.1.132	Intel C Compiler Version 17.0.1.132	Intel C Compiler Version 17.0.1.132

Table 2
Execution time to compute the 10^8th hexadecimal digit of π.

	Theoretical peak performance			Time	
	FP64 (TFlops)	FP32 (TFlops)	INT32 (Tops)	vector (sec)	scalar (sec)
Intel Xeon E5-2690 v4	0.582	1.165	0.582	1.251	2.031
Intel Xeon Phi 5150P	1.011	2.022	1.011	2.224	8.690
Intel Xeon Phi 7250	3.046	6.093	3.046	0.344	1.707
NVIDIA GeForce GTX 680	0.129	3.090	0.515	1.57 [23]	

for these three platforms are shown in Table 1. We note that Hyper-Threading [36] was enabled on each of these three platforms.

For the Intel Xeon E5-2690 v4, the Intel Xeon Phi 5110P, and the Intel Xeon Phi 7250, the compiler options were `icc -O3 -xHOST -qopenmp`, `icc -O3 -mmic -qopenmp`, and `icc -O3 -xMIC-AVX512 -qopenmp`, respectively. The compiler option `-O3` specifies to optimize for maximum speed and enable more aggressive optimizations, and `-xHOST` specifies to generate instructions for the highest instruction set and processor available on the compilation host machine. The compiler option `-mmic` specifies to build an application that runs natively on Intel MIC Architecture. The compiler option `-xMIC-AVX512` specifies to generate Intel AVX-512 Foundation instructions, Intel AVX-512 Conflict Detection instructions, Intel AVX-512 Exponential and Reciprocal instructions, and Intel AVX-512 Prefetch instructions. The compiler option `-qopenmp` specifies to enable the compiler to generate multi-threaded code based on the OpenMP directives. The executions on the Intel Xeon Phi 5110P were performed in “native mode”. The executions on the Intel Xeon Phi 7250 were performed in “flat mode” and “quadrant mode”. On the Intel Xeon Phi 5110P and the Intel Xeon Phi 7250, the environment variable `KMP_AFFINITY=granularity=fine,balanced` was specified.

Table 2 lists the single-node execution time required to compute the 10^8th hexadecimal digit of π on the Intel Xeon E5-2690 v4, the Intel Xeon Phi 5150P, the Intel Xeon Phi 7250, and Karrels’s result using the NVIDIA GeForce GTX 680 [23]. We note that the theoretical peak INT32 performances in Table 2 are based on the multiply-add operation for 32-bit integers.

The theoretical peak performance of the Intel Xeon Phi 7250 is about 3.01 times faster than that of the Intel Xeon Phi 5150P. With the Intel Xeon Phi 5150P, the Intel IMCI does not support the Intel AVX-512 `vpaddq` instruction for the 64-bit integer addition or the `vpmuludq` instruction for the 32-bit $\times$ 32-bit $\rightarrow$ 64-bit unsigned integer multiplication. The Intel C Compiler can vectorize the 64-bit integer addition with the IMCI `vpadcd` and `vpaddsetcd` instructions. We implemented a wrapper function for the 32-bit $\times$ 32-bit $\rightarrow$ 64-bit unsigned integer multiplication by using the Intel IMCI `_mm512_mulhi_epu32()`, `_mm512_mullo_epi32()`, and `_mm512_mask_shuffle_epi32()` intrinsic functions. This is why the Intel Xeon Phi 7250 (vector version) is about 6.47 times faster than the Intel Xeon Phi 5150P (vector version). Since the scalar version uses the x86_64 `mulq` instruction, which performs the 64-bit $\times$ 64-bit $\rightarrow$ 128-bit unsigned integer multiplication for the Montgomery squaring, it has an advantage in that there is no need to use the interleaved Montgomery multiplication in Algorithm 6. Nevertheless, on the Intel Xeon Phi 7250, the vector version is about 4.96 times faster than the scalar version.

Fig. 2 shows the speedup for computing the nth hexadecimal digit of π (vector version) on the Intel Xeon Phi 7250 when from 1 to 272 threads are used. We note that the Intel Xeon Phi 7250 has 68 cores. The results indicate that hyper-threading is effective for $n \geq 10^8$.

5.2. Multi-node performance

The performance was measured on the Fujitsu PRIMERGY CX1640 M1 cluster at the Joint Center for Advanced High Performance Computing (JCAHPC), which the University of Tokyo and University of Tsukuba jointly operate. The original program was written in C with OpenMP and MPI. We used the vector version described in Section 5.1. The specification

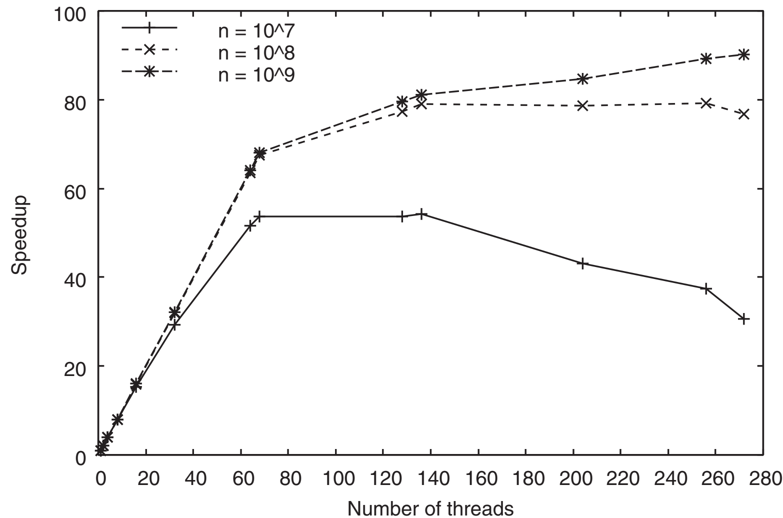


Fig. 2. Speedup for computing the nth hexadecimal digit of π (vector version) on the Intel Xeon Phi 7250.

Table 3
Specification of the Fujitsu PRIMERGY CX1640 M1 cluster.

Number of nodes	8208
CPU	Intel Xeon Phi 7250 (68-core, 1.4 GHz)
Main memory	MCDRAM 16 GB + DDR4-2400 96 GB
Theoretical peak performance	25.004 PFlops
Total main memory size	897.75 TB
Interconnect	Intel Omni-Path Architecture
Network topology	Fat-tree
OS	Linux 3.10.0–327.22.2.el7.xppsl_1.4.1.3272.x86_64
C compiler	Intel C Compiler Version 17.0.1.132
MPI library	Intel MPI 5.1.3.258

of the Fujitsu PRIMERGY CX1640 M1 cluster is shown in Table 3. The experiments used from 1 to 512 nodes. The compiler options were specified as `mpiicc -O3 -xMIC-AVX512 -qopenmp`. The executions on the Intel Xeon Phi 7250 were performed using “flat mode” and “quadrant mode”. With the Intel Xeon Phi 7250 cluster, each processor has 1 MPI process, and 268 threads per processor were used. The environment variable `KMP_AFFINITY=granularity=fine,balanced` was specified.

Fig. 3 shows the average execution time required to compute the n-th hexadecimal digit of π on the Fujitsu PRIMERGY CX1640 M1 cluster. For $n = 10^9$ on 512 nodes, the parallelization overhead dominates the execution time, as shown in Fig. 3. On the other hand, we can see that the speedup of the parallel implementation is nearly linear for $n = 10^{11}$ on 512 nodes.

6. The computation of the 100 quadrillionth hexadecimal digit of π

We have computed the 100 quadrillionth ($= 10^{17}$th) hexadecimal digit of π by using Bellard’s formula on the Fujitsu PRIMERGY CX1640 M1 cluster at the Joint Center for Advanced High Performance Computing (JCAHPC). The computation was performed during the test operation period. The main run and the verification run were each performed on 512 nodes. Due to the runtime limit for jobs, the main run and the verification run were each performed as 200 separate jobs. The elapsed times of the main run and the verification run were 320 h 31 min and 320 h 57 min, respectively.

The main run computed 32 hexadecimal digits of π starting at position 10^{17}, and the verification run computed 32 hexadecimal digits of π starting at position $10^{17} - 1$. A comparison of these results showed that the hexadecimal digits of π from the 10^{17}th to the $10^{17} + 22$nd digits were consistent. Table 4 shows the computed hexadecimal digits of π. Computation of the ten quadrillionth ($= 10^{16}$th) hexadecimal digit of π by Karrels [14,37] has been verified with our computed 25 hexadecimal digits of π starting at position 10^{16}.

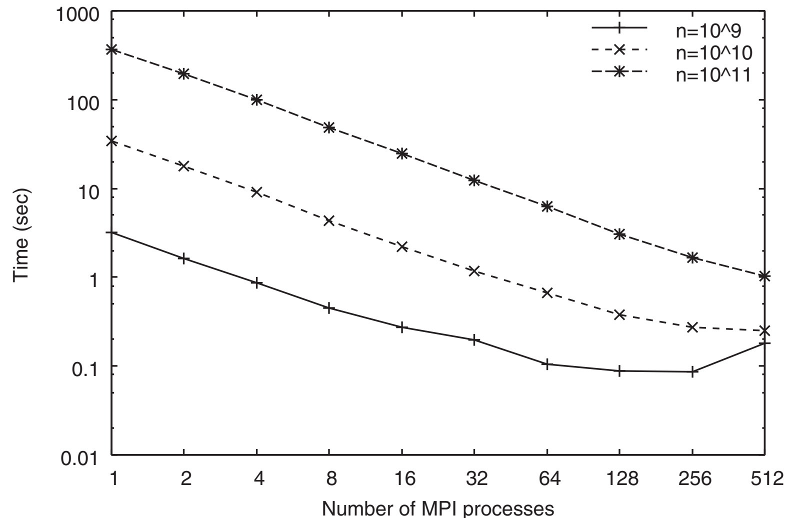


Fig. 3. Execution time for computing the nth hexadecimal digit of π on the Fujitsu PRIMERGY CX1640 M1 cluster.

Table 4
Computed hexadecimal digits of π.

Position	Hexadecimal digits starting at this position
10^6	26C65E52CB459350050E4BB17
10^7	17AF5863EFED8DE97033CD0F6
10^8	ECB840E21926EC5AE0D2F3405
10^9	85895585A0428B564084E74A2
10^{10}	921C73C6838FB2B6223630F51
10^{11}	C9C381872D27596F81D0E48B9
10^{12}	5B4466E8D215388C4E014CEC5
10^{13}	A0F9FF371D17593E0D06D5892
10^{14}	0D39BABA1B8FED53DD5F8BDE8
10^{15} [14]	8353CB3F7F0C9ACCFA9AA215F
10^{16}	9077E0164B9C613FD6C7F170C
10^{17}	A937EB59439E485E

7. Conclusion

This paper presented the use of a BBP-type formula on a cluster of Intel Xeon Phi processors to compute a specific hexadecimal digit of π. The BBP-type formula can be computed using modular exponentiation. We used Montgomery multiplication for the modular multiplication, which is the most time-consuming part of the modular exponentiation. We vectorized the multiple modular exponentiations and the multiple integer divisions by using the Intel AVX-512 instructions. The parallel implementation of the BBP-type formula was presented. The 100 quadrillionth hexadecimal digit of π was computed on a 512-node cluster of Intel Xeon Phi processors with an elapsed time of 641 h 29 min that includes the time required for verification.

Acknowledgment

This research was conducted using the Fujitsu PRIMERGY CX1640 M1 cluster (Oakforest-PACS) in the Joint Center for Advanced High Performance Computing (JCAHPC). This research was partially supported by JSPS KAKENHI Grant Number JP16K00168.

References

[1] F. Bellard, Computation of 2700 billion decimal digits of pi using a desktop computer, 2010, http://bellard.org/pi/pi2700e9/pipcrecord.pdf.
[2] D. Takahashi, Parallel implementation of multiple-precision arithmetic and 2,576,980,370,000 decimal digits of π calculation, Parallel Comput. 36 (2010) 439–448.
[3] A.J. Yee, S. Kondo, 12.1 trillion digits of pi, 2013, http://www.numberworld.org/miscruns/pi-12t/.
[4] P. Trueb, π^e trillion digit of π, 2016, http://www.pi2e.ch/.
[5] R.P. Brent, Fast multiple-precision evaluation of elementary functions, J. ACM 23 (1976) 242–251.
[6] E. Salamin, Computation of π using arithmetic-geometric mean, Math. Comput. 30 (1976) 565–570.
[7] J.M. Borwein, P.B. Borwein, Cubic and higher order algorithms for π, Can. Math. Bull. 27 (1984) 436–443.

[8] J.M. Borwein, P.B. Borwein, Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity, John Wiley & Sons, 1987.
[9] D.V. Chudnovsky, G.V. Chudnovsky, Approximations and complex multiplication according to Ramanujan, Ramanujan Revisited, Academic Press, 1988. 375–396 and 468–472
[10] D. Bailey, P. Borwein, S. Plouffe, On the rapid computation of various polylogarithmic constants, Math. Comput. 66 (1997) 903–913.
[11] D.H. Bailey, The BBP algorithm for pi, 2006, http://www.davidhbailey.com/dhbpapers/bbp-alg.pdf.
[12] C. Percival, PiHex A distributed effort to calculate Pi, 2000, http://wayback.cecm.sfu.ca/projects/pihex/.
[13] T.-W. Sze, The two quadrillionth bit of pi is 0! distributed computation of pi with Apache Hadoop, in: Proc. 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom 2010), 2010, pp. 727–732.
[14] E. Karrels, Computing digits of π with CUDA, 2017, http://www.karrels.org/pi/.
[15] D.H. Bailey, J.M. Borwein, A. Mattingly, G. Wightwick, The computation of previously inaccessible digits of π^2 and catalan's constant, Not. Am. Math. Soc. 60 (2013) 844–854.
[16] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, G. Yang, The sunway taihulight supercomputer: system and applications, Sci. China Inform. Sci. 59 (2016) 072001:1–072001:16.
[17] TOP500 Supercomputer Sites, http://www.top500.org/.
[18] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y.-C. Liu, Knights landing: second-generation intel xeon phi product, IEEE Micro. 36 (2016) 34–46.
[19] F. Bellard, A new formula to compute the n'th binary digit of pi, 1997, http://bellard.org/pi/pi_bin.pdf.
[20] R. Brent, P. Zimmermann, Modern Computer Arithmetic, Cambridge University Press, 2010.
[21] D.E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third ed., Addison-Wesley, 1997.
[22] P.L. Montgomery, Modular multiplication without trial division, Math. Comput. 44 (1985) 519–521.
[23] E. Karrels, Computing the quadrillionth digit of π, 2013, http://on-demand.gputechconf.com/gtc/2013/presentations/S3071-Computing-the-Quadrillionth-Digit-of-Pi.pdf.
[24] G. Hachez, J.-J. Quisquater, Montgomery exponentiation with no final subtractions: Improved results, in: Proc. Second International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000), in: Lecture Notes in Computer Science, 1965, Springer-Verlag, 2000, pp. 293–301.
[25] E.W. Mayer, Efficient long division via montgomery multiply, Comput. Res. Reposito. (2016). abs/1303.0328. http://arxiv.org/abs/1303.0328v6.
[26] S. Gueron, V. Krasnov, Software implementation of modular exponentiation, using advanced vector instructions architectures, in: Proc. 4th International Workshop on the Arithmetic of Finite Fields (WAIFI 2012), in: Lecture Notes in Computer Science, 7369, Springer-Verlag, 2012, pp. 119–135.
[27] J.W. Bos, P.L. Montgomery, D. Shumow, G.M. Zaverucha, Montgomery multiplication using vector instructions, in: Proc. Selected Areas in Cryptography 2013 (SAC 2013), in: Lecture Notes in Computer Science, 8282, Springer-Verlag, 2014, pp. 471–489.
[28] Intel Corporation, Intel architecture instruction set extensions programming reference, 2016a, https://software.intel.com/sites/default/files/managed/26/40/319433-026.pdf.
[29] Intel Corporation, Intel 64 and IA-32 architectures software developer's manual, volume 1: Basic architecture, 2016b, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf.
[30] Intel Corporation, Intel C++ compiler 17.0 developer guide and reference, 2016c, https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide-pdf.
[31] Texas Instruments Incorporated, TMS320C64x DSP library programmer's reference, 2003, http://www.ti.com/lit/ug/spru565b/spru565b.pdf.
[32] Free Software Foundation, Inc., GCC, the GNU Compiler Collection, https://gcc.gnu.org/.
[33] The Clang Team, clang: a C language family frontend for LLVM, http://clang.llvm.org/.
[34] T. Jebelean, An algorithm for exact division, J. Symb. Comput. 15 (1993) 169–180.
[35] Intel Corporation, Intel Xeon Phi coprocessor instruction set architecture reference manual, 2012, https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf.
[36] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture, Intel Technol. J. 6 (2002) 1–11.
[37] A. Bellos, Pi day 2015: a sweet treat for maths fans, 2015, https://www.theguardian.com/science/2015/mar/13/pi-day-celebration-maths-fans-language-memory-contests.