核资源,提高运行效率.

1 大整数乘法 SSA 算法简介

1.1 大整数数据结构

大整数一般采用基存储的方式,以 R 为基的 n 位大整数可表示为

$$a=(a_0,a_1,...,a_{n-1})_R=\sum_{i=0}^{n-1}a_iR^i \tag{1}$$

其中,$a_i(0\leqslant i\leqslant n-1)$为 0 或正整数,且 $0\leqslant a_i<R$,最高位 $a_{n-1}\neq 0$.大整数与多项式在形式上基本一致,可将大整数看做是某种特殊类型的多项式 $A(x)$,其中,$x=R$,其值:

$$A(R)=\sum_{i=0}^{n-1}a_iR^i \tag{2}$$

其中,a_i 的条件同公式(1)[1].

大整数数位的数据类型(单位可用 *limb* 表示)和大整数数据结构的定义如下:

```
typedef unsigned long mp_limb_t;        //定义大整数数位的数据类型
typedef struct                          //定义大整数数据结构
{   int _mp_alloc;                      //大整数动态数组的长度
    int _mp_size;                       //大整数动态数组已用的长度和符号
    mp_limb_t* _mp_d;                   //大整数动态数组首地址指针
}mpz_struct;                            //定义存储大整数的数据结构
```

其中,*limb* 的大小一般和处理器位数相关,比如 64 位处理器,1 *limb* 等于 64 位 bit,其最大值的 10 进制数表示为 18 446 744 073 709 551 615.本文涉及的大整数乘法 SSA 算法是基于大整数多项式模型进行分析研究的.

1.2 算法原理

大整数可以用多项式来表示,那么大整数乘法也就是多项式乘法,其实质是求两个向量的卷积.根据卷积定理[20,21],向量卷积的傅里叶变换是向量傅里叶变换的乘积.另外,傅里叶变换有快速算法(fast fourier transformation,简称 FFT),可以将算法的复杂度降低到 $O(n\log n)$,因此能利用 FFT 来求解大整数乘法.但是根据循环卷积和线性卷积的定义,傅里叶变换对应的是循环卷积而不是线性卷积[22],这就需要一些额外操作完成求解.

假设要求解大整数 a 和 b 的乘积,a 的多项式表示为

$$A(R)=(a_0,a_1,...,a_{n-1})_R=\sum_{i=0}^{n-1}a_iR^i \tag{3}$$

其中,a_i 的条件同公式(1),其向量表示为$(a_0,a_1,\ldots,a_{n-1})$.b 的多项式表示为

$$B(R)=(b_0,b_1,...,b_{m-1})_R=\sum_{i=0}^{m-1}b_iR^i \tag{4}$$

其中,$b_i(0\leqslant i\leqslant m-1)$为或正整数,且 $0\leqslant b_i<R$,最高位 $b_{m-1}\neq 0$,a 和 b 的向量表示为$(a_0,a_1,\ldots,a_{n-1})$和$(b_0,b_1,\ldots,b_{m-1})$.众所周知,最高次数为 $n-1$ 的多项式与最高次数为 $m-1$ 的多项式相乘,其结果的最高次数为 $n+m-2$,所以 a 和 b 的乘积 c 是一个最高次数为 $n+m-2$ 的多项式[23].但是根据傅里叶变换的定义,大整数 $a(a_0,a_1,\ldots,a_{n-1})$和 $b(b_0,b_1,\ldots,b_{m-1})$的傅里叶变换结果分别是 n 点序列和 m 点的序列,那么要想应用傅里叶变换求解多项式 a 和 b 的乘积,就需要对 a 和 b 两个向量进行扩充,也即将 a 和 b 两个向量的长度扩充到 $n+m-2$,具体方式见算法 1.

算法 1. 基于 FFT 的大整数乘法算法.

Input: 大整数 a(n 位)和 b(m 位);

Output: $c=a\times b$.

1. $\overline{a}=(a,0_{m-1}),\overline{b}=(b,0_{n-1})$
2. $\hat{a}=FFT(\overline{a}),\hat{b}=FFT(\overline{b})$
3. $\hat{c}=\hat{a}\cdot\hat{b}$
4. $c=IFFT(\hat{c})$

5. 对 c 进行进位操作

算法 1 的第 1 步对 a 和 b 两个向量高阶部分进行补零操作,完成扩充;第 2 步分别对扩充后的两个向量进行 FFT 计算;第 3 步对得到的 FFT 两个序列进行逐点相乘操作;第 4 步对点乘结果做快速傅里叶逆变换 IFFT;最后对第 4 步得到的序列做进位等操作,得到结果 c.

上面的算法已经实现了利用 FFT 计算大整数乘法,但是仍存在两个主要缺陷:其一是对 a 和 b 两个向量需要进行扩充操作,在一定程度上增加了计算量;其二是传统的 FFT 算法是基于复数上的 n 次单位根 $\omega_n=\cos(2\pi/n)+i\sin(2\pi/n)$的,因此必须要进行三角函数类的浮点运算,从而会有一定程度的截断误差.

对于扩充问题,可以利用求循环卷积或者负循环卷积来代替求线性卷积来解决.循环卷积和负循环卷积的定义在文献[24,25]中有比较详细的介绍,下面主要介绍利用 FFT 计算循环卷积和负循环卷积的方法:假设 $a=[a_0,a_1,\ldots,a_{n-1}]^T$ 和 $b=[b_0,b_1,\ldots,b_{n-1}]^T$ 是长度为 n 的列向量.ω是主 n 次单位根,并且$\psi^2=\omega$.假设 n 有一个乘法的逆,那么,

$$a\text{ 和 }b\text{ 的循环卷积}=IFFT(FFT(a)\cdot FFT(b)) \tag{5}$$

$$a\text{ 和 }b\text{ 的负循环卷积}=\psi^{-1}\cdot IFFT(FFT(\psi\cdot a)\cdot FFT(\psi\cdot b)) \tag{6}$$

其中,ψ称为权重因子.由于计算负循环卷积的过程与计算循环卷积的过程类似,因此用循环卷积或负循环卷积代替线性卷积,都能很好地解决扩充问题.

对于截断误差问题,可以利用快速数论变换(fast number theoretic transforms,简称 FNT)技术来解决.数论变换具有与傅里叶变换相似的结构,但其是利用整数根代替复指数单位根,且所有运算按整数模来定义.数论变换实际只是傅里叶变换在有限域的一种表现形式,即:傅里叶变换的插值是取复数域内的 N 次方根,数论变换的插值是取某个剩余系内的 N 次方根,其余的运算和傅叶变换基本一致.数论变换同样具有循环卷积性质,并且在某些情况下可以只利用加法和移位操作来计算,可以显著减少计算量,而且相比于傅里叶变换可以精确地计算卷积而没有舍入误差.

大整数乘法 SSA 算法就是采用上面两种方法对基于傅里叶变换的大整数乘法算法进行优化的.为了实现递归求解,即:在对得到的傅里叶变换后的两个序列进行逐点相乘操作时,可以递归地调用此算法来求解,选取的是求解负循环卷积来解决扩充问题;同时,利用 FNT 技术,在有限域上求解负循环卷积来解决截断误差问题,具体内容见算法 2.

算法 2. 大整数乘法 SSA 算法.

Input: 大整数 a(n 位)和 b(m 位);

Output: $c=a\times b$.

1. $\overline{a}=\psi\cdot a,\overline{b}=\psi\cdot b$
2. $\hat{a}=FNT(\overline{a}),\hat{b}=FNT(\overline{b})$
3. $\hat{c}=\hat{a}\cdot\hat{b}$
4. $\overline{c}=IFNT(\hat{c})$
5. $c=\psi^{-1}\cdot\overline{c}$
6. 对 c 进行进位操作

算法 2 中,第 1 步对 a 和 b 进行分解操作,其实质就是将 a 和 b 两个大整数转化成多项式的过程,并对两个向量乘以权重因子;第 2 步对 a 和 b 两个向量进行快速数论变换;第 3 步对得到的变换后的两个序列进行逐点相乘操作,当点乘的规模很大时,递归调用 SSA 算法;第 4 步对第 3 步中得到的乘积序列做快速数论逆变换(IFNT);第 5 步对第 4 步中得到的结果除以权重因子;最后对第 5 步得到的序列做进位等操作,得到结果 c.

1.3 热点分析

利用 SSA 算法求取负循环卷积的计算过程可以分成 4 个步骤进行计算,分别是分解(decompose)、快速数论转换、点乘(MUL)和快速数论逆转换.运行串行 SSA 算法,对规模为上万 *limbs* 的两个随机生成的大整数进行实验,测试并统计各步骤占总计算时间的比重,见表 1 所示.表中数据显示:点乘操作占比最大超过 60%,这 4 个步

骤占用时间百分比超过 90%.因此,这些是 SSA 算法并行优化的研究热点.

Table 1 Percent of operation time

表 1 运算时间百分比

计算过程	分解	FNT	点乘	IFNT
百分比(%)	3.8	19.4	62.6	9.5

2 大整数乘法 SSA 算法多核并行实现

SSA 算法并行方案是从算法的内部进行多核并行,其核心是需对 SSA 算法的各个核心步骤进行详细地分析,然后根据分析结果分别对每个步骤定制最优的多核并行方案,其优势在于能够充分利用平台的 CPU 资源.

2.1 分解并行方案设计

分解操作的主要工作是根据大整数的规模将其进行分解,以得到大整数的多项式形式,然后对得到的多项式序列乘以权重因子,其采用循环 for 的方式实现.分解过程任务间没有依赖关系,所以可以直接对这部分进行细粒度的多核并行优化.但当规模较小的时候,该并行方式开销较大,需采用粗粒度的优化方案.粗粒度方案是对 SSA 算法中 a 和 b 的分解操作利用 OpenMP 中的 section 并行方式,两个任务并行执行;细粒度方案是利用 OpenMP 中的 for 循环迭代进行多核并行.

2.2 FNT和IFNT并行方案设计

根据规模的大小,采用粗粒度和细粒度两种并行方案进行.对于小规模,采用粗粒度方式,利用 OpenMP 中的 section 并行方式对函数接口内部进行优化,因其实现采用的是时域抽取法,故只对递归的第 1 层进行并行,即函数内部的两次递归调用过程利用 section 方式两个线程并行执行.对于大规模,采用细粒度方式,其采用递归分解的策略进行,如图 1 所示.以 8 线程为例,可将其前两层的递归调用从递归过程中拆分出来,即消除第 1 层递归调用和第 2 层递归调用,当递归调用开始时,直接进入第 3 层递归,然后每次递归调用再按原方式进行.

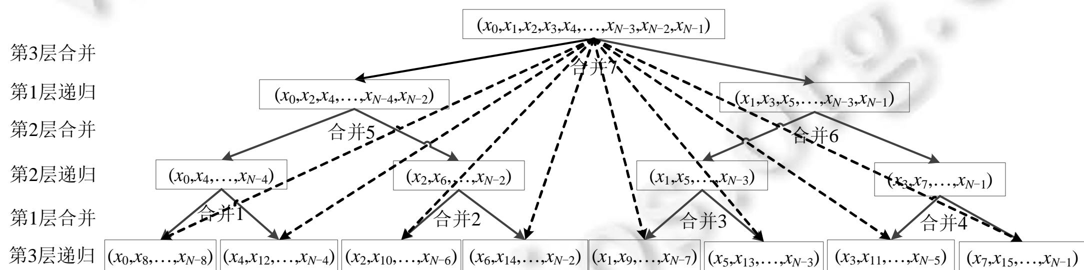


Fig.1 Analysis diagram of the parallelization of the FNT (#threads=8)

图 1 以 8 线程为例,FNT 并行方案分析图

当对递归过程进行如上的改变后,需要进行相应的合并操作以得到正确的结果值,其具体过程如图 1 中的合并 1~合并 7 操作.同时,每层合并操作可多线程并行执行.另外,本方案会根据大整数规模通过调整递归分解策略选择合适的并行方案,以达到最好的并行效果.

IFNT 其并行方案与 FNT 的并行方案相同.

2.3 点乘并行方案设计

点乘部分所需的计算时间最长,并且会随着规模的增加而增大,其主要是利用 OpenMP 中的 for 循环迭代进行多核并行,因任务间没有依赖关系,可以根据线程数直接进行任务划分,然后多个线程分别执行所负责的子任务.但是需要注意的是:SSA 算法本身是递归算法,当进行点乘的数据规模很大时,多个线程内的每对元素的乘

法调用串行 SSA 算法进行.

2.4 SSA算法并行方案设计

通过算法 2 的介绍可知:这 4 步之间存在严格的时间先后关系,并且不存在线程嵌套的情况.根据上面对本方案并行策略的介绍,其多核并行方案流程图如图 2 所示.

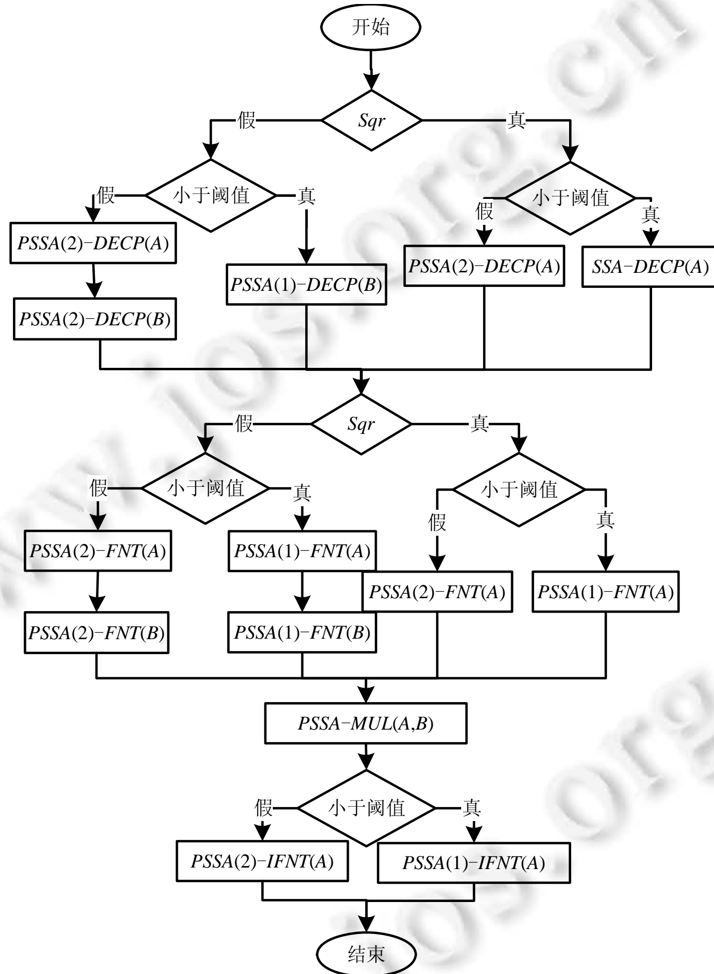


Fig.2 Flow chart of theparallel SSA algorithm

图 2 SSA 算法并行方案流程图

图 2 中,*SSA*–*代表 SSA 算法中的操作,*PSSA*(1)–*代表 SSA 算法并行方案中各操作的粗粒度并行方式,*PSSA*(2)–*代表 SSA 算法并行方案中各操作的细粒度并行方式,*DECP* 表示分解操作.

3 实验结果和分析

3.1 实验准备

实验分别在两个平台进行:平台 1 为某 IntelX86 平台,平台 2 为浪潮 TS850 服务器,信息见表 2.

实验基于大整数运算开源库 GMP-5.1.3 进行算法的正确性验证和性能测试,选取大整数的基 *R* 为 2^{64},大整数的规模以 *unsigned long int*(*limb*)类型为基本单位.选取了 10 组大整数进行实验,每组测试数据都是规模为从

www.jos.org.cn
www.jos.org.cn

几十万到上千万 limbs 的两个大整数,所有测试数据均随机产生.

Table 2 Information of test platform

表 2 测试平台信息

平台名	CPU 型号	CPU 主频	内存	操作系统	编译器
平台 1	Intel(R) Xeon(R) X5550(2 路 4 核)	2.67GHz	64G	Ubuntu 11.04	gcc4.5.2
平台 2	Intel(R) Xeon(R)X7550(8 路 8 核)	2GHz	512G	Ubuntu RDCPS 2.6.35-32-server	gcc4.2.1

3.2 并行算法加速比和可扩展性

实验 1 在某 Intel X86 平台主要测试了 SSA 算法多核并行方案 8 线程时的性能以及 GMP-5.1.3 库相应串行程序的性能,为了与文献[17,18]中方案进行对比,我们还实现了 karatsuba+SSA 的并行方案,其测试结果如图 3 所示.图中 PSSA 代表 SSA 算法并行方案在 8 线程时性能,SSA-karatsuba 代表 karatsuba 算法与 SSA 算法结合的并行方案在 8 线程时性能,大整数规模的单位为百万 limbs.实验结果表明:SSA 算法并行方案得到了较好的加速性能,其性能优于 karatsuba 算法与 SSA 算法结合的并行方案,其平均加速比为 6.41,最大加速比可达到 6.59.

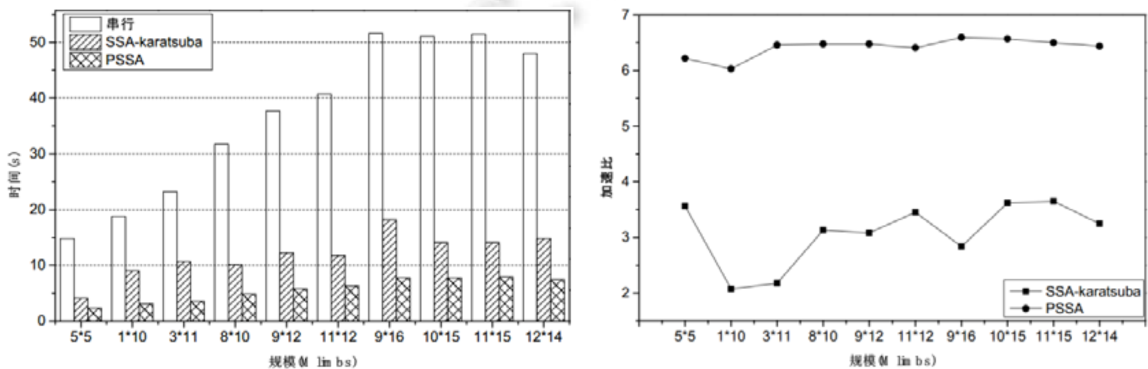


Fig.3 Performance comparison of two parallel schemes for SSA algorithm

图 3 SSA 算法两种并行方案性能对比

另外,本文还测试了加速比与规模的关系,如图 4 所示.当规模较少时,加速比比较低;随着规模的增大,加速比逐渐增大;并且当规模大于 2Mlimbs 时,加速比趋于稳定.

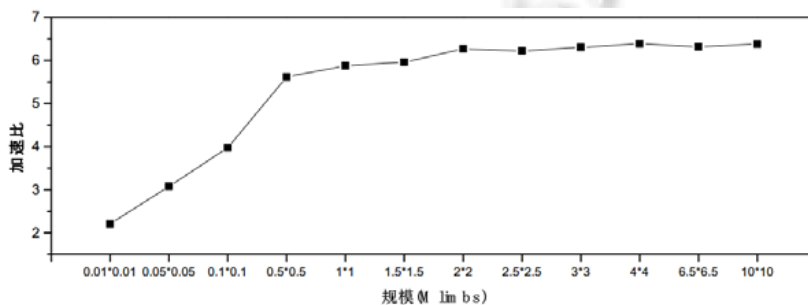


Fig.4 Relationship of speedup and scale

图 4 加速比与规模的关系图

实验 2 在浪潮 TS850 服务器对 SSA 算法并行方案的扩展性进行测试,实验分别以 8 线程、16 线程、32 线程和 64 线程以及 GMP-5.1.3 库中相应串行程序进行,其性能如图 5 所示.图中并行(*)代表 SSA 算法并行方案在 8,16,32,64 线程时的性能,加速比(*)代表此方案在 8,16,32,64 线程时的加速比.实验结果表明:此方案具有一

定的扩展性,加速比会随着线程数的增加而增大,并且 16 线程的加速比最大能达到 11.14,32 线程的加速比最大能达到 16.58,64 线程的加速比最大能达到 21.42.

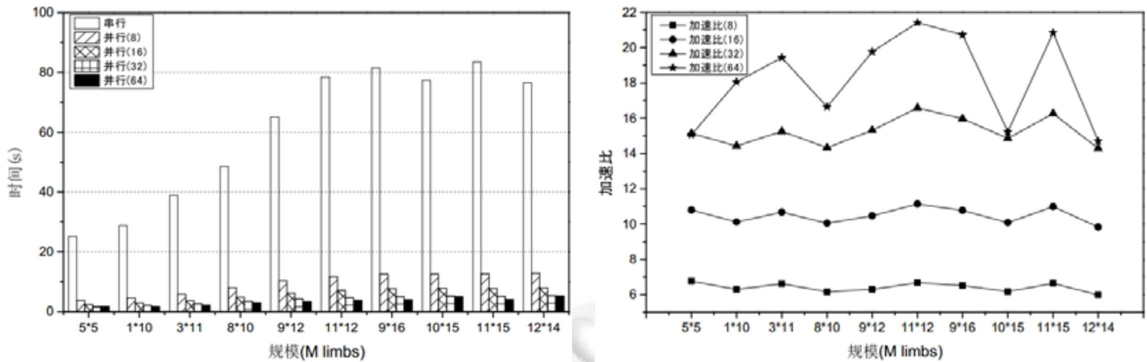


Fig.5 Scalability of the parallel SSA algorithm

图 5 SSA 算法并行方案扩展性

3.3 实验结果分析

采用阿姆达尔定律[26]来分析并行 SSA 算法的加速比.大整数乘法 SSA 算法的基础虽然是 FFT 算法,但是该算法比 FFT 算法更复杂,SSA 算法中的大部分运算都可以并行,但是仍有一小部分计算不能并行.另外,在 FNT 和 IFNT 操作中采用的是递归算法,在并行算法中额外引入了合并过程,这部分开销也需要考虑.整体加速比上限可以用如下公式进行计算:设 SSA 中需要串行计算的部分运行时间为 W_s,可并行计算部分的运行时间为 W_p,p 为线程数,W_o 为并行算法引入的开销,则:

$$Speedup_{SSA}=\frac{W_s+W_p}{W_s+\frac{W_p}{p}+W_o}.$$

不同计算规模 W_s 和 W_o 占总串行时间的比例不同,当计算规模较大时(大于 5Mlimbs),SSA 算法中 W_s 约占总串行时间的 2%,W_o 约占总串行时间的 0.4%,即:

- 当 p=8 时,$Speedup_{SSA}$=1/(0.02+0.98/8+0.004)=6.83;
- 当 p=64 时,$Speedup_{SSA}$=1/(0.02+0.98/64+0.004)=25.44.

本文的 SSA 并行算法 8 线程时最大加速比为 6.59,64 线程时最大加速比为 21.42,考虑到并行区开启的开销,该加速比已经接近加速比上限,所以说本文的加速效果是比较理想的.

3.4 SSA并行算法特点

大整数乘法 SSA 算法的运算流程比较复杂,本文针对其运算流程中的每步精心设计了性能更高的并行方案,如图 2 所示.本文的并行方案有两个主要特点:1) SSA 算法中大量使用了递归,本文根据线程数对递归层进行拆解并行,并采用高效的合并以快速得到正确的结果;2) SSA 算法并行过程中采用自适应的思想,根据运算规模选用不同的并行方案,以取得更好的性能.这两点对大整数运算中其他递归算法的并行优化均有一定的借鉴意义,如大整数分解、GCD 等算法.

本文提出的 SSA 并行方案在 SSA 算法实现本身存在多种递归实现以至于难以并行优化的条件下仍取得较为理想的加速效果,充分说明本文的研究内容具有一定的实际应用意义.

4 结束语

大整数乘法 SSA 算法的并行优化具有重要的研究价值和应用意义.本文详细介绍了该算法的一种并行方案,对 SSA 算法的各个核心步骤进行细粒度的多核并行优化,得到了良好的加速性能.下一步工作拟从以下两个

方面展开:一方面,从算法实现上来进行进一步优化,例如可将算法中的递归实现改成迭代实现,也可以充分利用中国剩余定理,将算法以更利于并行优化的方式实现;另一方面,将本文的技术与众核并行化相结合,在众核平台上综合考虑算法级和代码级两个层次优化来进一步改进算法性能.

References:

[1] Sang B. Research and fast implementation of large integers multiplication algorithm [MS. Thesis]. Guangzhou: South China University of Technology, 2012 (in Chinese with English abstract).

[2] Karatsuba A, Ofman Y. Multiplication of multidigit numbers on automata. Soviet Physics-Doklady, 1963,7:595−596.

[3] Toom AL. The complexity of a scheme of functional elements realizing the multiplication of ISSntegers. Soviet Mathematics Doklady, 1963,3:714−716.

[4] Cooley JW, Tukey JW. An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation, 1965,19: 297−301.

[5] Schönhage A, Strassen V. Schnelle multiplikation großer zahlen. Computing, 1971,7(3-4):281−292.

[6] Gaudry P, Kruppa A, Zimmermann P. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In: Proc. of the 2007 Int'l Symp. on Symbolic and Algebraic Computation. ACM Press, 2007. 167−174.

[7] Sze TW. Schönhage-Strassen algorithm with MapReduce for multiplying terabit integers. In: Proc. of the 2011 Int'l Workshop on Symbolic-Numeric Computation. ACM Press, 2012. 54−62.

[8] Fürer M. Faster integer multiplication. In: Proc. of the 39th ACM Symp. on Theory of Computing. 2007. 57−66.

[9] Fürer M. Faster integer multiplication. SIAM Journal on Computing, 2009,39(3):979−1005.

[10] Keliris A, Maniatakos M. Investigating large integer arithmetic on Intel Xeon Phi SIMD extensions. In: Proc. of the 9th Int'l Conf. on Design & Technology of Integrated Systems in Nanoscale Era (DTIS 2014). Santorini: IEEE, 2014. 1−6.

[11] Tembhurne JV, Sathe SR. Performance evaluation of long integer multiplication using OpenMP and MPI on shared memory architecture. In: Proc. of the 2014 7th Int'l Conf. on Contemporary Computing (IC3). IEEE, 2014. 283−288.

[12] Jebelean T. Using the parallel karatsuba algorithm for long integer multiplication and division. In: Proc. of the Euro-Par'97 Parallel Processing. Berlin, Heidelberg: Springer-Verlag, 1997. 1169−1172.

[13] Mansouri F. On the parallelization of integer polynomial multiplication [MS. Thesis]. University of Western Ontario, 2014.

[14] Xu L, Wang Z. Fast large integer multiplication based on CUDA. Computer Engineering and Applications, 2013,49(16):221−225 (in Chinese with English abstract).

[15] Chmielowiec A. Fast, parallel algorithm for multiplying polynomials with integer coefficients. In: Proc. of the World Congress on Engineering. 2012.

[16] Emeliyanenko P. Efficient multiplication of polynomials on graphics hardware. In: Proc. of the Advanced on Parallel Processing Technologies. Berlin, Heidelberg: Springer-Verlag, 2009. 134−149.

[17] Emmart N, Weems C. High precision integer multiplication with a graphics processing unit. In: Proc. of the 2010 IEEE Int'l Symp. on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). IEEE, 2010. 1−6.

[18] Emmart N, Weems C. High precision integer multiplication with a GPU. In: Proc. of the 2011 IEEE Int'l Symp. on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW). IEEE, 2011. 1781−1787.

[19] Doroz Y, Ozturk E, Sunar B. Evaluating the hardware performance of a million-bit multiplier. In: Proc. of the 2013 Euromicro Conf. on Digital System Design (DSD). IEEE, 2013. 955−962.

[20] Jiang CJ, Jiang Y. Fast Fourier Transform and C Procedures. Hefei: University of Science and Technology of China Press, 2004. (in Chinese).

[21] Nussbaumer HJ. Fast polynomial transform algorithms for digital convolution. IEEE Trans. on Acoustics, Speech, and Signal Processing, 1980,28(2):205−215.

[22] Hu GS. Digital Signal Processing: Theory, Arithmetic and Realization. 2nd ed., Beijing: Tsinghua University Press, 2003. (in Chinese).

[23] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. 3rd ed., Cambridge: The MIT Press, 2009. 900−905.

[24] Proakis JG, Rader CM, Ling F, Nikias CL, Moonen M, Proudler IK. Algorithms for Statistical Signal Processing. London: Prentice Hall, 2002. 63−67.

[25] Crandall R, Fagin B. Discrete weighted transforms and large-integer arithmetic. Mathematics of Computation, 1994,62(205): 305−324.

[26] Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. In: Proc. of the Spring Joint Computer Conf. ACM Press, 1967. 483−485.

附中文参考文献:

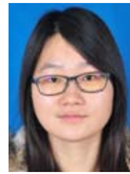
[1] 桑波.大整数乘法算法的研究与快速实现[硕士学位论文].广州:华南理工大学,2012.

[14] 许亮,王震.基于 CUDA 的快速大整数乘法.计算机工程与应用,2013,49(16):221−225.

[20] 蒋长锦,蒋勇.快速傅里叶变换及其 C 程序.合肥:中国科学技术大学出版社,2004.

[22] 胡广书.数字信号处理:理论、算法与实现.第 2 版,北京:清华大学出版社,2003.

赵玉文(1987－),女,河北唐山人,助理研究员,CCF 专业会员,主要研究领域为高性能扩展数学库,并行计算.



蒋丽娟(1990－),女,本科生,CCF 学生会员,主要研究领域为并行计算.

刘芳芳(1982－),女,高级工程师,CCF 专业会员,主要研究领域为高性能扩展数学库,稀疏迭代解法器,异构众核并行.

杨超(1979－),男,博士,教授,博士生导师,主要研究领域为高性能计算,科学与工程计算.

www.jos.org.cn