Scalability of Algorithms for Arithmetic Operations in Radix Notation*

Anatoly V. Panyukov †

Department of Computational Mathematics
and Informatics, South Ural State University,
Chelyabinsk, Russia
paniukovav@susu.ac.ru

Abstract

We consider precise rational-fractional calculations for distributed computing environments with an MPI interface for the algorithmic analysis of large-scale problems sensitive to rounding errors in their software implementation. We can achieve additional software efficacy through applying heterogeneous computer systems that execute, in parallel, local arithmetic operations with large numbers on several threads. Also, we investigate scalability of the algorithms for basic arithmetic operations and methods for increasing their speed.

We demonstrate that increased efficacy can be achieved of software for integer arithmetic operations by applying mass parallelism in heterogeneous computational environments. We propose a redundant radix notation for the construction of well-scaled algorithms for executing basic integer arithmetic operations. Scalability of the algorithms for integer arithmetic operations in the radix notation is easily extended to rational-fractional arithmetic.

Keywords: integer computer arithmetic, heterogeneous computer system, radix notation, massive parallelism

AMS subject classifications: 68W10

1 Introduction

Verified computations have become indispensable tools for algorithmic analysis of large scale unstable problems (see e.g., [1, 4, 5, 7, 8, 9, 10]). Such computations require specific software tools; in this connection, we mention that our library "Exact computation" [11] provides appropriate instruments for implementation of such computations

*Submitted: January 27, 2013; Revised: August 17, 2014; Accepted: November 7, 2014.

†The author was supported by the Federal special-purpose program "Scientific and scientific-pedagogical personnel of innovation Russia", project 14.B37.21.0395.

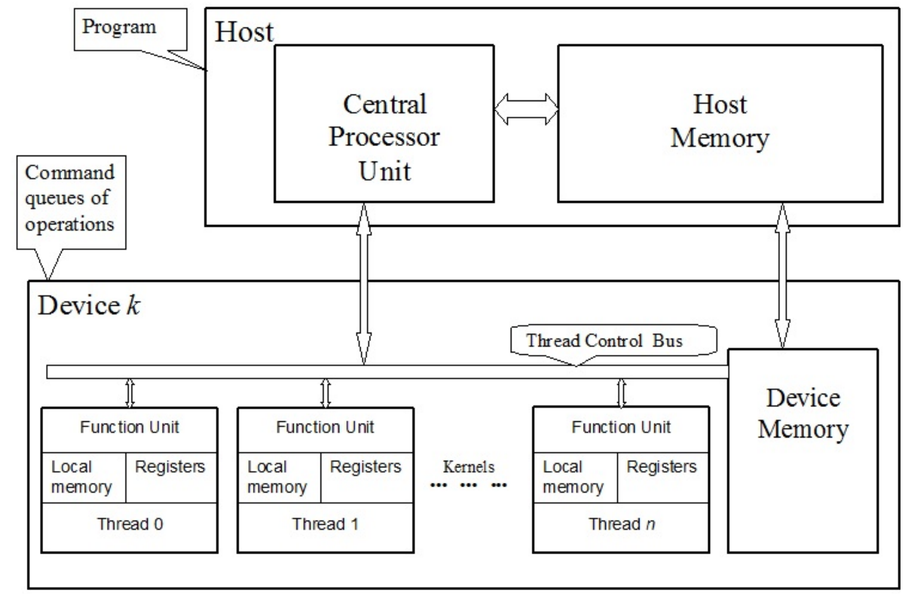

Figure 1: Fragment of a heterogenous system architecture

in a distributed computing environment. Further increases of efficacy are possible by involving heterogeneous computing environments that allow one to parallelize execution of local arithmetic operations through a large number of processes.

Let several processes, numbered $k = 0, 1, \ldots, n$, execute an operation ρ and require execution times t_k^ρ, respectively. Then the time required for the entire operation to be completed is $t^\rho = \max\{t_k^\rho : \ k = 0, 1, \ldots, n\}$. The key algorithm property which determines the efficiency of the operation execution in parallel is scalability. If a sufficient number of processes for execution of an algorithm is available, an algorithm is called *completely scalable* if its execution time does not depend on the length of operands, and an algorithm is called *well-scalable* if its execution time is $O(\log_2 n)$, where n is the maximal length of the operands.

In this paper, we investigate scalability of the basic algorithms implementing arithmetic operations, and we develop completely and well-scalable algorithms for these operations. We demonstrate that redundant positional notation produces completely scalable addition/subtraction algorithms and well-scalable algorithms for the remaining basic arithmetical operations. We present here the results on scalable algorithms for integer arithmetic announced at the conferences [2, 3, 8, 12, 13].

2 Heterogenous Computational Systems

Figure 1 presents the structure of a typical heterogenous computational system consisting of the managing host unit containing the CPU and a set of devices. The CPU runs programs and provides operating system connections. The Device block provides parallel execution of basic operations with the objects of the program.

Inter-block data exchange on the Thread Control Bus connects device memory and host memory via the direct memory access bus (DMA) of the host. All local device interprocess communications ("Point to Point") can be executed in parallel and asynchronously. Sending the data from process k to many processes can be carried out in two steps. First, process k positions the data to be sent on a shared thread or a shared device memory. Second, recipients processes knowing the sender process k read transmitted data. Reading the message from the shared memory may be performed by all the devices simultaneously.

In summary, such a system offers low-cost, low-powered, high-performance computing. However, the transfer speed between the host CPU and the multi-kernel device can become a bottleneck, making it unfit for applications that require frequent data exchange.

Programs for heterogenous computational systems contain `Host` and `Device` modules. `Host` modules are similar to programs for homogenous systems. Their functionality includes transmissions of the operation code, address and word length of operands to the device modules, and initialization of the required number of the processes. Let us use the prefix `_global_` for the names of host procedures. The `Device` modules include a wide variety of devices with high demands for scalability of the algorithms executed on them because of the large numbers of device kernels that may have more time steps than the central processing unit.

In this paper, we offer Pascal-like pseudo-code for the algorithms.

3 Parallel Algorithms for Integer Arithmetic Operations in the Classical Radix Notation

3.1 Addition of nonnegative numbers

A possible method of parallel execution of the classical addition algorithm for n-digit numbers is the parallel digit-by-digit summation. As the result, a part of the digits appear as binary carries, and we assume their number to be l. After that, we can form l parallel processes for binary carry propagation. Algorithm 1 describes the procedures Digit_Addition, Carry_Propagation, and Add_Process for a local digit process, and the procedure _global_Add that defines the width of summands and creates the required quantity of the parallel processes. Further, it is necessary to call the procedure _global_Add to get the result of the addition $(a_{n-1} \ldots a_0)_R + (b_{m-1} \ldots b_0)_R$.

Let us estimate the time expenditures for the execution of Algorithm 1 as well as a possible gain from executing it in parallel. Numbers in a computer's memory are stored in a binary format. If r is the word length, the base of the radix notation is $R = 2^r$. Most modern processors use either 32 bit words or 64 bit words. Below, we use the noted values of R and r.

For each of the parallel processes, the procedure Digit_Addition requires time for elementary addition on the register actually equal to the time of sending s. The execution time of the procedures Carry_Propagation and Add_Process can vary within the bounds $[0,\ n \cdot s]$, that is, the execution time of Algorithms 1 can change in the above interval depending on the initial terms. The execution time of the addition executed by a strictly sequential algorithm (i.e., digit after digit) is evaluated as $3n \cdot s$, if the above terms are accepted.

Let us estimate the mean execution time of Algorithm 1, assuming that the digits of the summands are random uniformly distributed quantities. To simplify our ac-

Algorithm 1 Addition

Requires: $R = 2^r$, $n \ge m$, $a_i = \left(a_i^{r-1} \ldots a_i^0\right)_2$, $i = 0, 1, 2, \ldots, n-1$, and $b_j = \left(b_j^{r-1} \ldots b_j^0\right)_2$, $j = 0, 1, 2, \ldots, m-1$;

Produces: $t = (t_n, \ldots, t_0)_R = (a_{n-1} \ldots a_0)_R + (b_{m-1} \ldots b_0)_R$, $t_n \in \{0, 1\}$.

procedure DIGIT_ADDITION(In: a, b, i, Out: c, t)
  $\left(s_i^r\, s_i^{r-1} \ldots s_i^1\, s_i^0\right)_2 \leftarrow \left(a_i^{r-1}\, a_i^{r-2} \ldots a_i^1\, a_i^0\right)_2 + \left(b_i^{r-1}\, b_i^{r-2} \ldots b_i^1\, b_i^0\right)_2$;
  $t_i \leftarrow \left(s_i^{r-1} \ldots s_i^1\, s_i^0\right)_2$ ▷ i-th digit before carry propagation
  $c \leftarrow s_i^r$ ▷ carry value to $(i+1)$-th digit
end procedure

procedure CARRY_PROPAGATION(In: n, i, InOut: c, t)
  while $c \neq 0$ **do** ▷ there is not carry if $c = 0$
    $i \leftarrow i + 1$;
    $\left(s_i^r\, s_i^{r-1} \ldots s_i^1\, s_i^0\right)_2 \leftarrow t_i + c$
    $t_i \leftarrow \left(s_i^{r-1} \ldots s_i^1\, s_i^0\right)_2$ ▷ i-th digit after carry propagation
    $c \leftarrow s_i^r$ ▷ carry value to next digit
  end while
  Terminate process
end procedure

procedure ADD_PROCESS(In: a, b, i, Out: t)
  var c ▷ for carry of this local process
  DIGIT_ADDITION(a, b, i, c, t)
  CARRY_PROPAGATION(n, i, c, t)
end procedure

procedure _GLOBAL_ADD(In: a, b, Out: n, m, t) ▷ exec add in parallel
  $n \leftarrow \max\{\text{sizeof }(a), \text{sizeof }(b)\}$
  $m \leftarrow \min\{\text{sizeof }(a), \text{sizeof }(b)\}$
  for all $i = 0, 1, \ldots, m-1$ **do**
    ExecInParallel ADD_PROCESS(a, b, i, t)
  end for
end procedure

counting, we assume $n = m$. The probability that a carry takes place in at least one of the digits during the summation is

$$p = \mathrm{P}\{a_i + b_i \geq 2^r\} = \sum_{l=1}^{2^r-1} \mathrm{P}\{a_i = l\}\,\mathrm{P}\{b_i \geq 2^r - l\} = \sum_{l=1}^{2^r-1} \frac{1}{2^r} \cdot \frac{l}{2^r} = \frac{1}{2}\left(1 - \frac{1}{2^r}\right).$$

The probability of obtaining the value $2^r - 1$ as the result of the summation is

$$q = \mathrm{P}\{a_i + b_i = 2^r - 1\} = \sum_{l=0}^{2^r-1} \mathrm{P}\{a_i = l\}\,\mathrm{P}\{b_i = 2^r - 1 - l\} = \sum_{l=0}^{2^r-1} \frac{1}{2^r} \cdot \frac{1}{2^r} = \frac{1}{2^r}.$$

The probability of chaining the carry with the length $k \geq l$ is

$$P_l = \mathrm{P}\left\{ \bigcup_{i=0}^{m-l-1} \left(\left(a_i + b_i \geq 2^r - 1\right) \bigcap_{j=1}^{l} (a_{i+j} + b_{i+j} = 2^r - 1) \right) \right\} = (m-l)pq^l.$$

An r-bit operating system supports numbers with $m = 2^r$ of r-bit digits. Therefore, let $m = 1/q$ be an under-estimation of probabilities P_l, $l = 0, 1, \ldots, m$, and we have $P_l \leq q^{l-1}$.

It is easy to see that the value of the probability satisfies $P_2 \leq q$. Therefore, the mean time of the algorithm execution is equal to $2s$ asymptotically. Also, the average speed of the examined algorithm is n times greater than that of the sequential algorithm, and this figure does not depend on the length of the summands. We can decrease the time of the addition execution for the worst case after improvement of the carry propagation algorithm. One of the possible ways to do that is calculation of the carry propagation chains simultaneously with their propagation. If the carry falls on the calculated chain, then its propagation within this chain is accomplished for one time. The procedure Carry_Propagation described in Algorithm 2 implements such accelerated carry propagation.

The essence of Algorithm 2 can be described as follows. Initially, the result of the procedure Digit_Addition (i.e., the number t and digit-by-series carry c) is represented in the form of n fragments, each digit d_i, $i = 0, 1, \ldots, n-1$ corresponding to the fragment with a separate process i. At the k-th iteration of the **while** cycle, we join the fragments associated with the processes $l2^k$ and $(l+1)2^k$ into one fragment associated with the process $(l+1)2^k$. When joining, the lower process $l2^k$ sends the absent ripple carry flag *NotCarry*, the carry c itself, and possible ripple carry merge V into the higher process $(l+1)2^k$. If the transfer from the lower-order fragment $l2^k$ takes place, the higher joining process $(l+1)2^k$ is produced into all necessary digits (from $l2^k+1$-th to $V((l+1)2^k)$-th). The merge of the propagation of the ripple-through carry in the united fragment also is refined, and unnecessary processes are terminated for all cases.

Consider the time complexity of Algorithm 2. The procedure Carry_Propagation contains a loop that is carried out by any of the processes not more than $\lceil \log_2 n \rceil$ times. Each of the active processes runs operators indicated in the lines 2, 3, 4, 5, and 41 of this loop. After appropriate optimization of the heterogenous computing environment, these operators can be executed in parallel in one tick. Each of the active processes also executes not more than one receiving communication and not more than one sending communication. Their preparation and execution requires two ticks.

Thus, the mean and worst case execution times of the carry propagation by Algorithm 2 are $3s$ and $3s\lceil \log_2 n \rceil$, respectively. Asymptotically, the mean time of executing addition by Algorithm 2 for the carry propagation is $4s$, which is $4/3$ times the mean time for Algorithm 1. However, the efficiency of Algorithm 2 is obvious, provided there are carry circuits with the length of more than two digits.

Algorithm 2 Improved carry propagation.

```
procedure CARRY_PROPAGATION(In: n, i, Out: c, t)
    L ← 1, V ← i                          ▷ length and verge of the joined fragments
    while L ≤ n do                        ▷ there are fragments for joining
        M ← i mod 2L
        if ( M < L ) then                 ▷ i belong to the lower fragment
            if ( M = L − 1 ) then         ▷ i is high digit of the low fragment
                j ← min{i + L, n − 1}     ▷ high digit of joined fragment
                NotCarry ← (t_i ≠ 2^r − 1) ∪ (V ≠ i)     ▷ absent ripple carry
                send {c, NotCarry, V} to process j
                if ((c ≠ 0) ∪ NotCarry) then
                    terminate process
                end if
            end if
        else                              ▷ i belong to the higher fragment
            j ← i + L − M − 1             ▷ j is higher digit of the lower fragment
            flag ← ((M = 2L − 1) ∪ (i = n − 1))
            if flag then                  ▷ i is higher digit of the higher fragment
                receive {Cj, NotCarry, Vj} from process j
                send {Cj, NotCarry} to processes k = j + 1, . . . , V
                if (NotCarry) then
                    V ← Vj
                else if ( i = V ) then
                    (s^r s^{r−1} . . . s^1 s^0)_2 ← (c t_i^{r−1} . . . t_i^1 t_i^0)_2 + Cj
                    t_i ← (s^{r−1} . . . s^1 s^0)_2 , c ← s^r
                end if
            else                          ▷ i is not high digit of the high fragment
                if ( i ≤ V ) then         ▷ i belong to the ripple carry chain
                    receive {Cj, NotCarry} from process j + L
                    if ( Cj ≠ 0 ) then
                        t_i ← 0
                        if ( i = V ) then
                            t_{i+1} ← t_{i+1} + 1
                        end if
                        terminate process
                    else if NotCarry then
                        terminate process
                    end if
                end if
            end if
        end if
        L ← 2L
    end while
    terminate process
end procedure
```

3.2 The binary relations

To check the value of any binary relation $a \, \rho \, b \ : \rho \in \{<, \leq, =, \geq, >, \neq\}$, it is sufficient to check the relations $\rho \in \{=, >\}$. In fact, $(a \leq b) = \neg(a > b)$, $(a \neq b) = \neg(a = b)$, $(a \geq b) = (a > b) \vee (a = b)$, $(a < b) = \neg(a \geq b)$.

Algorithm 3 calculates the Boolean value of the binary relations $(a = b)$ and $(a > b)$ for non-negative integers a and b. The essence of the algorithm can be described as follows. The initial data are presented as n fragments with separated processes $i = 0, 1, \ldots, n-1$. In this case, p_i and q_i are truth of relations $(a_i = b_i)$ and $(a_i > b_i)$ for fragments $i = 0, 1, \ldots, n-1$. With the k-th execution of **for** loop, the confluence of the fragments associated with the processes of $l2^k$ and $(l+1)2^k$ to one fragment associated with the process $l2^{k-1}$ is accomplished. During this confluence, the values of p_i and q_i are recalculated, and the unnecessary processes are terminated.

It is easy to see that the speed of Algorithm 3 is $n/\log_2 n$ times higher in comparison with the sequential one.

3.3 Determination of the number of significant digits

To distribute computational resources rationally for execution of the arithmetic operations, it is necessary to know the number of significant digits of its operands.

For addition, multiplication, and division, the number of significant digits of the operands determines the number of significant digits of the result with the error of one digit. For subtraction, the number of significant digits can be determined only after its execution. Therefore, the rational use of the computational resources requires an algorithm for determining the number of significant digits of the result.

Algorithm 4 calculates the number of significant digits for a non-negative integer represented in the radix (positional) notation with the base $R = 2^r$. It is evident from the description of Algorithm 4 that its execution time does not exceed $4s\lceil \log_2 n \rceil$. It is reasonable to use the number of significant digits as one of the object attributes, and Algorithm 4 should be used only after a subtraction is executed.

3.4 Multiplication of a multi-digit number by a digit

Algorithm 5 calculates the product $(c_n, \ldots, c_0)_R$ of non-negative integers $a = (a_{n-1}, \ldots, a_0)_R$ and $b = (b_0)_R$ represented in the radix notation with base $R = 2^r$. First, the algorithm calculates products of the digit b and digits a_i, $i = 0, \ldots, n-1$ (line 3). In general, any such product is a two-digit number $(x_1 \, x_2)_R \leq (2^r - 1)^2 = 2^r(2^r - 2) + 1$, i.e., the value x_1 carried to the next digit (lines 4,9, and 10) does not exceed $2^r - 2$. Also, one can see that there are no delayed carry chains (lines 11, 12, 13, 14) with the length more than 1, and we have $t_i \leq 2^r - 1$ for all $i = 0, \ldots, n-1, n$.

It is evident from the description of Algorithm 5 that the time expenditure for execution of the procedure M is not greater than $4s$, so the speed of Algorithm 5 is n times higher than that of the sequential algorithm, and this conclusion does not depend on the length of the multiplied numbers.

3.5 Multiplication of multi-digit numbers

Algorithm 6 calculates the product of two non-negative integers represented in the radix notation with the base $R = 2^r$.

Algorithm 3 Checking the truth of the binary relations $(a = b)$ and $(a > b)$.

Requires: $a = (a_{n-1} \dots a_0)_R$, and $b = (b_{n-1} \dots b_0)_R$, $R = 2^r$, $n > 0$, $a_i = \left(a_i^{r-1} \dots a_i^0\right)_2$, $b_i = \left(b_i^{r-1} \dots b_i^0\right)_2$, $i = 0, 1, 2, \dots, n-1$

Produces: p is truth of relation $a = b$, and q represents truth of relation $a > b$.

```
procedure EqG_Process(In: a, b, n, i, Out: p, q)
    p ← (a = b)
    q = (a > b)
    L ← n/2
    while ( L > 1 ) do
        if ( i > 0 ) then
            send {p, q} to process i/2
        end if
        if ( i < L ) then
            if ( there is sending from process 2i ) then
                receive {p_0, q_0}
            else p_0 = true, q_0 = false
            end if
            if ( there is sending from process 2i + 1 ) then
                receive {p_1, q_1}
            else p_1 = true, q_1 = false
            end if
            __syncthreads()
            p ← p_1 ∧ p_0
            q ← q_1 ∨ (p_1 ∧ q_0)
        else
            terminate process
        end if
        L ← L/2
    end while
end procedure

procedure _global_EqG(In: a, b, n, Out: p, q)
    for all i = 0, 1, ..., n − 1 do
        ExecInParallel EqG_Process(a_i, b_i, n, i, p_i, q_i)
    end for
    p ← p_0, q ← q_0
end procedure
```

Algorithm 4 Calculating the number of significant digits of unsigned a

Requires: $a = (a_{n-1} \dots a_0)_R$, $n > 0$, $R = 2^r$.
Produces: S is the number of significant digits of a.

```
procedure NSD_PROCESS(In: a, i, n, Out: s)
    if a > 0 then
        s ← i
    else
        s ← 0
    end if
    L ← n/2
    while ( L > 1 ) do
        if ( i > 0 ) then
            send s to process i/2
        end if
        if ( i < L ) then
            if ( there is sending from process 2i ) then
                (receive value for s_0
            else s_0 ← 0
            end if
            if ( there is sending from process 2i + 1 ) then
                receive value for s_1
            else S_1 ← 0
            end if
            __syncthreads()
            s ← max{s_0, s_1}
        else
            terminate process
        end if
        L ← L/2
    end while
end procedure

procedure _GLOBAL_NSD(In: a, n, Out: s)
    for all i = 0, 1, ..., n − 1  do
        ExecInParallel NSD_PROCESS(a_i, i, n, s_i)
    end for
    s ← s_0
end procedure
```

Algorithm 5 Calculating the product of a and digit b

Requires: $a = (a_{n-1} \, \dots \, a_0)_R$, $n > 0$, $b = (b_0)_R$, $R = 2^r$.
Produces: $(t_n \; t_{n-1} \, \dots \, t_0)_R$ is product of a and b.

procedure M_PROCESS(In: ad, b, i, n, dt)
  if $(i < n)$ **then**
    $(x_1 \; x_0)_R \leftarrow ad \cdot b$
    send x_1 **to process** $(i + 1)$
  else
    $x_0 \leftarrow 0$
  end if
  if $(i > 0)$ **then**
    receive x_1 **from process** $(i - 1)$
    $\left(s^r \; s^{r-1} \, \dots \, s^1 \; s^0\right)_2 \leftarrow x_0 + x_1$
    $c \leftarrow s^r, \quad dt \leftarrow \left(s^{r-1} \, \dots \, s^1 \; s^0\right)$
    send c **to process** $(i + 1)$
    receive c **from process** $(i - 1)$
    $dt \leftarrow dt + c$
  else
    $dt \leftarrow x_0$
  end if
end procedure

procedure _GLOBAL_M(a, b, n, t)
  for all $i = 0, 1, \dots, n$ **do**
    ExecInParallel M_PROCESS(a_i, b, i, n, t_i)
  end for
end procedure

The execution time of procedure _GLOBAL_M (line 2 of Algorithm 6) is $4s$. The body of the **while** loop (lines 4 to 23) is performed not more than $\lceil \log_2 m \rceil$ times. It contains no more than one sending (line 6) communication, two receiving (lines 11 and 15) communications "point-to-point", and one addition of $(n+m-2L)$-digit numbers. The remaining operators can be executed in one tick. Hence, on the average, the time necessary for performing the multiplication does not exceed $(4 + 3\log_2 m)\, s$. In the worst case, the time does not exceed $(4 + 3(n + m)\log_2 m)\, s$.

Procedure M of Algorithm 6 creates m processes, each of which calls procedure M (line 2), which creates n processes. Consequently, the total number of the processes generated is mn. In the worst case, the execution time grows slightly faster than a linear function of the length of the operands, whereas the sequential long multiplication algorithm has quadratic execution time in the length of factors.

3.6 Division

The classical "long division" algorithm, in contrast to the preceding operations, is not scalable. Its execution requires $(n+m-1)$ sequential carry operations of multiplication-

Algorithm 6 Calculating the product $c = a \cdot b$

Requires: $a = (a_{n-1} \dots a_0)_R$, $b = (b_{m-1} \dots b_0)_R$, $n \geq m > 0$, $R = 2^r$
Produces: $(c_{n+m-1}, \dots, c_0)_R$ is product of a and b

```
procedure MM_Process(a, b, i, n, m, c)
    _global_M(a, b_i, n, z)
    L ← m/2, B ← R
    while ( L > 1 ) do
        if ( i > 0 ) then
            send z to process i/2
        end if
        __syncthreads()
        if ( i < L ) then
            if ( there is sending from process 2i ) then
                receive value for s_0
            else s_0 ← 0
            end if
            if ( there is sending from process 2i + 1 ) then
                receive value for s_1
            else s_1 ← 0
            end if
            s_1 ← s_1 · B
            _global_Add(s_0, s_1, n, m, z)
        else
            terminate process
        end if
        L ← L/2, B ← B^2
    end while
    c ← z
end procedure

procedure _global_MM(a, b, n, m c)
    for all i = 0, 1, ..., m − 1 do
        ExecInParallel MM_Process(a, b, i, n, m, z)
    end for
    c ← z
end procedure
```

subtraction with m-digit numbers. References [6] and [13] propose to increase efficiency of the division operation by applying Newton's method. To divide an integer $u = (u[n-1]\ u[n]\ \ldots\ u[1]\ u[0])_R$ by an integer $v = (v[m-1]\ \ldots\ v[0])_R$, we first find a sufficiently accurate approximation to the number $1/v$. Then we multiply it by u, giving an approximation to u/v. The length of the integer answer is not more than $n-m+1$. The number $1/v$ contains not more than m insignificant zeros in the high-order places. To obtain the correct result of division, it is sufficient that the approximate value of $1/v$ additionally contains at least $n-m+1$ significant digits. Thus, adequate accuracy of calculation of $1/v$ is determined by the value R^{-n+1}.

Applying Newton's method to the problem of finding the root of the equation $f(x) = 0$, where $f(x) = v - 1/x$, consists of the sequential calculations

$$x_{k+1} \leftarrow (2 - v \cdot x_k) \cdot x_k, \qquad k = 0, 1, 2, \ldots,$$

where x_0 is a sufficiently accurate initial approximation. The function $f(x) = v - 1/x$ is twice continuously differentiable and strictly convex for $x > 1$. Hence, Newton's method exhibits quadratic convergence, i.e., the number of correct digits doubles with each iteration. The initial approximation of $x_0 = 1/v[m-1]$ for $1/v$ has error

$$\frac{1}{v[m-1] \cdot R^{m-1}} - \frac{1}{v} = \frac{v - v[m-1] \cdot R^{m-1}}{v \cdot v[m-1] \cdot R^{m-1}} \leq \frac{1}{v \cdot v[m-1]} \leq R^{-m+1},$$

i.e., it has m digits correctly calculated. Thus, the required number of Newton iterations does not exceed $4\log_2(n+1) - \log_2 m$.

Algorithm 7 calculates the quotient of two non-negative integers represented in the radix notation with the base $R = 2^r$.

Algorithm 7 Calculating the quotient $c = a/b$ of non-negative integers a and b

Requires: $a = (a_{n-1}\ \ldots\ a_0)_R$, $b = (b_{m-1}\ \ldots\ b_0)_R$, $n \geq m > 0$ are represented in the radix notation with the base $R = 2^r$

Produces: $(c_{n+m-1}, \ldots, c_0)_R$ is the quotient $c = a/b$.

```
1: procedure _GLOBAL_D(a, b, n, m, c)
2:     x ← ⌊(R−1)/b[m−1]⌋_R,   R̃ ← R^m                    ▷ Initial approximation
3:     for i = 0, 1, , ⌈log_2 (n+1)/m⌉ do                  ▷ More precise definition
4:         d ← x,   x ← (2 · R̃ − b · d) · d,   R̃ ← R̃ · R̃
5:     end for
6:     z = a · x                                            ▷ Multiplication
7:     c = z/R̃                                              ▷ Answer forming
8: end procedure
```

At iteration k, $k = 0, 1, 2, \ldots, l$, $l < \log_2(n+1) - \log_2 m$ of the **for** loop, the variable x represents an integer $(2^{k+1}-1)$-digit number. Within the loop body, parallel algorithms perform one multiplication of x by an m-digit number b, one subtraction of (2^k)-digit numbers, and one multiplication of (2^k)-digit numbers. Consequently, the execution time of the loop body does not exceed $11 + 3(\log_2 m + k)] \cdot s$ (average) or

$\left[3 \cdot 2^k k + 2.5 \cdot 2^k + 6k + 3m\log_2 m + 10\right] s$ (worst case). Since

$$\sum_{k=0}^{l} k = \tfrac{l(l+1)}{2}, \quad \sum_{k=0}^{l} 2^k = 2^{l+1} - 1,$$

$$\sum_{k=0}^{l} \left(k \cdot 2^k\right) \leq \sqrt{\sum_{k=0}^{l} k^2 \sum_{k=0}^{l} 4^k} = \sqrt{\tfrac{2l^3+3l^2+l}{6} \cdot \tfrac{4^{l+1}-1}{3}} \leq 2^{l+1}\sqrt{\tfrac{l^3}{3}},$$

the average and worst case execution times of the **for** loop do not exceed $O\left(\log_2 n \cdot \log_2\left(\frac{n+1}{m}\right)\right)$ and $O\left(\frac{n+1}{m}\log_2{}^{3/2}\left(\frac{n+1}{m}\right)\right)$, respectively.

The multiplication of n-digit numbers completes the execution of the procedure D. The execution time of this step does not exceed $O(\log_2 n)$ on average and $O(n \cdot \log_2 n)$ in the worst case. Thus, the final estimates for the execution time of Algorithm 7 on average and in the worst case are equal to $O\left(\log_2 n \cdot \log_2\left(\frac{n+1}{m}\right)\right)$ and $O\left(\frac{n+1}{m}\log_2{}^{3/2}\left(\frac{n+1}{m}\right) + n\log_2 n\right)$, respectively.

4 Use of Signed Radix Notation

The number system we have considered is unsigned, and the digits of the position system with the base R are numbers 0, 1, 2, ..., $R-2$, $R-1$. Its drawback is the quite complex implementation of the addition and subtraction operations, which requires numeric comparison. We can remove that deficiency by applying signed radix notation. The digits of the signed radix notation with base R are integers

$$-\left\lfloor\frac{R}{2}\right\rfloor, -\left\lfloor\frac{R}{2}\right\rfloor + 1, \ldots, -1, 0, 1, 2, \ldots, \left\lceil\frac{R}{2}\right\rceil - 2, \left\lceil\frac{R}{2}\right\rceil - 1.$$

For odd R, the number of positive and negative digits are equal, and for even R the number of positive digits is one less than the number of negative digits.

In the sequel, the representation of a number in the signed position radix notation with the base $R = 2^r$ is designated as $(a_{n-1}, \ldots, a_0)_{\pm R}$, and its digits are $a_i = (a_i^{r-1}\, a_i^{r-2}\, \ldots\, a_i^1\, a_i^0)_{\pm 2}$, $i = 0, 1, \ldots, n-1$. The higher bit of the digit representation determines its sign (0 for positive numbers and 1 for negative ones). Hence, the digits of the signed radix notation are objects of type `integer`. All the basic algorithms for the unsigned numeration systems, except for addition/subtraction, can be transferred to signed systems without any change. The addition/subtraction algorithms are united into one general algorithm, the algebraical addition Algorithm 8.

The procedure SDigit_Addition of Algorithm 8 calculates the sum of unsigned representations of data type `integer` and forms the result of summation and the carry into the next digit. If overflow does not occur, then the carry is absent, and the sign of the result does not change. Otherwise, if overflow occurs, the sign of result changes switches, and the carry of the corresponding sign is formed. The application of signed position systems simplifies the algorithm of algebraic addition, but it does not change the efficiency of computations if there are ripple-through carries.

As in unsigned systems, it is possible to perform accelerated calculation of the chain of ripple-through carry and its propagation, as shown in Algorithms 9 and 10.

Advantages and deficiencies of the accelerated carry propagation are the same as for unsigned systems. The truth of binary relations in signed systems is recognized easily by a subtraction. The sign of the number is determined by the sign of its highest digit. The algorithms for determining the number of significant digits, multiplication, and division are similar to the those for unsigned systems.

Algorithm 8 Algebraic addition

Requires: $a_i = \left(a_i^{r-1} \dots a_i^0\right)_{\pm 2}$, $b_j = \left(b_j^{r-1} \dots b_j^0\right)_{\pm 2}$, $n \geq m$, $R = 2^r$.
Produces: $t = (t_n, \dots, t_0)_{\pm R} = (a_{n-1} \dots a_0)_{\pm R} + (b_{m-1} \dots b_0)_{\pm R}$.

procedure CarryForm(In: s, Out: c, t)
    if $(s^r = s^{r-1})$ **then**
        $c \leftarrow 0$
    else if $(s^r = 1)$ **then**
        $s^{r-1} \leftarrow c \leftarrow 1$
    else if $(s^r = 0)$ **then**
        $s^{r-1} \leftarrow 0$, $c \leftarrow -1$
    end if
    $t_i \leftarrow \left(s_i^{r-1} \dots s_i^1\, s_i^0\right)_{\pm 2}$
end procedure

procedure SCarry_Propagation(In: n, i, c, Out: t)
    while $c \neq 0$ **do** ▷ there is not carry if $c = 0$
        $i \leftarrow i + 1$;
        $\left(s_i^r\, s_i^{r-1} \dots s_i^1\, s_i^0\right)_{\pm 2} \leftarrow t_i + c$
        CarryForm(s_i, c, t)
    end while
    Terminate process
end procedure

procedure SDigit_Addition(In: a, b, i, Out: c, t)
    $\left(s_i^r\, s_i^{r-1} \dots s_i^0\right)_{\pm 2} \leftarrow \left(a_i^{r-1}\, a_i^{r-2} \dots a_i^0\right)_{\pm 2} + \left(b_i^{r-1}\, b_i^{r-2} \dots b_i^0\right)_{\pm 2}$;
    CarryForm(s_i, c, t)
end procedure

procedure AAdd_Process(In: a, b, i, Out: t)
    var c ▷ for carry of this local process
    SDigit_Addition(a, b, i, c, t)
    SCarry_Propagation(n, i, c, t)
end procedure

procedure _global_AAdd(In: a, b, Out: n, m, t) ▷ addition in parallel
    $n \leftarrow$ sizeof (a), $m \leftarrow$ sizeof (b)
    for all $i = 0, 1, \dots, m-1$ **do**
        ExecInParallel AAdd_Process(a, b, i, t)
    end for
end procedure

Algorithm 9 Improved carry propagation. (Part I)

```
procedure SCarry_Propagation(In: n, i, c, InOut: t)
    L ← 1, V ← i                           ▷ length and verge of the joined fragments
    while L ≤ n do                          ▷ there are fragments for joining
        M ← i mod 2L
        if (M < L) then                     ▷ i belongs to the lower fragment
            if (M = L − 1) then             ▷ i is higher digit of the lower fragment
                j ← min{i + L, n − 1}       ▷ higher digit of joined fragment
                nrc ← (−2^{r−1} < t_i < 2^{r−1} − 1) ∪ (t_i ≠ t_{i+1})
                nrc ← nrc ∪ (c ≠ 0) ∪ (V ≠ i)     ▷ no ripple carry through i
                send {c, nrc, V} to process j
                if nrc then
                    terminate process
                end if
            end if
```

5 Use of Redundant Radix Notation

The analysis above shows high average efficiency of parallel execution of all the arithmetic operations. The mean computing time of addition, subtraction, multiplication by a single-digit number, and binary operations is $O(1)$; the mean computing time of multiplication and division of the numbers with word length n does not exceed $O(\log_2^2 n)$. However, in the worst case, the computing time of any operation with n-digit numbers is not smaller than $O(n)$ with the usual carry propagation and not smaller than $O(\log_2 n)$ with accelerated carry propagation. Such deviations from the mean values occur because chains of length more than one appear in carry propagation. To exclude undesired long chains, we can use a redundant radix notation [2].

A positive integer n-digit number N in the radix notation with the base R is represented as a unique ordered set of numbers,

$$N = (a_{n-1}\,\ldots a_1\,a_0)_R = \sum_{l=0}^{n-1} a_l R^l, \quad a_{n-1},\,\ldots,a_1,\,a_0 \in \mathbf{D} = \{0,1,2,\ldots,R-1\}\ .$$

This representation is unique because the set $\mathbf{D}$ contains exactly R elements that represent a segment of the set of positive integers including zero. The extension of the set $\mathbf{D}$ leads to the extension of the family of representations for the number N.

Next, we consider extending the number set $\mathbf{D}$ in a manner which enables adding (and subtracting) in time $O(1)$. Let the computing system use 2^r-bit registers with a base of the number system $R = 2^{r-1}$. Therefore, any digit a_i has non-redundant representation $\left(0\,a_i^{r-2}\,\ldots\,a_i^1\,a_i^0\right)_2$. In a redundant representation, we suppose that a_i may be represented with possible nonzero delayed carry a_i^{r-1} as $\left(a_i^{r-1}\,a_i^{r-2}\,\ldots\,a_i^1\,a_i^0\right)_2$.

Let us consider a possible implementation of addition. Suppose that the i-th digits of the summands have the form

$$a_i = \left(a_i^{r-1}\,a_i^{r-2}\,\ldots\,a_i^1\,a_i^0\right)_2, \quad b_i = \left(b_i^{r-1}\,b_i^{r-2}\,\ldots\,b_i^1\,b_i^0\right)_2\,,$$

i.e., they represent binary r-digit numbers. Digit-by-digit summation for each position yields an $(r+1)$-digit result

$$s_i = a_i + b_i = \left(a_i^{r-1}\,a_i^{r-2}\,\ldots\,a_i^1\,a_i^0\right)_2 + \left(b_i^{r-1}\,b_i^{r-2}\,\ldots\,b_i^1\,b_i^0\right)_2 = \left(s_i^r\,s_i^{r-1}\,\ldots\,s_i^1\,s_i^0\right)_2.$$

Algorithm 10 Improved carry propagation. (Part II)

```
        else                                        ▷ i belongs to the higher fragment
            j ← i + L − M − 1                       ▷ j is higher digit of the lower fragment
            flag ← ((M = 2L − 1) ∪ (i = n − 1))
            if flag then                            ▷ i is higher digit of the higher fragment
                receive {Cj, jnrc, Vj} from process j
                nrc ← jnrc ∪ (i ≠ V) ∪ (c ≠ 0)      ▷ no ripple carry through i
                if (i ≠ V) then
                    send {Cj, jnrc, nrc} to processes j + 1, ..., V − 1, V
                else
                    send {Cj, jnrc, nrc} to processes j + 1, ..., V − 1
                    t_i ← t_i + Cj
                end if
                if (jnrc) then
                    V ← Vj
                end if
            else                                    ▷ i is not higher digit of the higher fragment
                receive {Cj, jnrc, nrc} from process j + L
                if (jnrc) then
                    if (Cj = 1) then
                        if (t_i = 2^(r−1) − 1) then
                            t_i ← −2^(r−1)
                        else if (i = j + 1) then
                            t_i ← t_i + Cj
                        else
                            t_i ← t_i + 1
                        end if
                    else if (Cj = −1) then
                        if (t_i = −2^(r−1)) then
                            t_i ← 2^(r−1) − 1
                        else if (i = j + 1) then
                            t_i ← t_i + Cj
                        else
                            t_i ← t_i + Cj
                        end if
                        terminate process
                    end if
                end if
            end if
        end if
        L ← 2L
    end while
    terminate process
end procedure
```

In each digit, we use two elder bits for the transfer into the next digit, yielding the r-digit result

$$\tilde{s}_i = \left(0\;0\;s_i^{r-2}\;s_i^{r-3}\;\ldots\;s_i^1\;s_i^0\right)_2 + \left(s_{i-1}^r\;s_{i-1}^{r-1}\right)_2 = \left(\tilde{s}_i^{r-1}\;\tilde{s}_i^{r-2}\;\ldots\;\tilde{s}_i^1\;\tilde{s}_i^0\right)_2.$$

Three clock ticks are required for executing the addition: (1) sending one of the terms into a register, (2) summing the terms, and (3) summing the carry.

The above algorithm uses a position numeration system with the redundant digit set. In our example, the use of the redundant bit as a postponed carry makes it possible to perform addition in constant time. Subsequently, we will designate the representation of the number in the redundant radix notation with the base R as $(a_{n-1}, \ldots, a_0)_{*R}$. Since the algorithms for multiplication and division are correct in this numeration system and contain only the addition operation, its use makes the execution time of these operations not worse than the estimates of the mean execution time obtained earlier in this paper.

As a disadvantage, the non-uniqueness of the number representation in the redundant radix notation enables efficient computation of the binary relations, but the number of significant digits can be computed only after the global carry propagation that removes the redundancy of representation. Let us recall that, asymptotically, the probability of the appearance of additional carries approaches zero.

6 Conclusion

Massive parallelism in a heterogeneous computational environment supports increasing efficiency of the software that implements integer arithmetic. Using a redundant radix notation proposed here allows construction of well-scaled algorithms of the basic arithmetic operations. Scalability of the algorithms for integer arithmetic operations in the radix notation can be extended easily to rational-fractional arithmetic.

The results our work relate to local arithmetic operations over numbers whose execution can be organized on computers with random-access memory. If the numbers are so huge that the random-access memory is not sufficient for their storage, then several devices may prove necessary. However, interfaces between the central processor and the device or between the devices have restrictions on capacity and access. The efficiency of the arithmetic operations with huge numbers is the subject of the further research. Perhaps an implementation of Toom-Cook or Karatsuba rapid multiplication algorithms [6] may prove efficient for this case.

References

[1] R. Alt, J.-L. Lamotte, and S. Markov. On the accuracy of the solution of linear problems on the CELL processor. *Reliable Computing*, 15:1–12, 2011.

[2] A.V. Panyukov. Application of redundant positional notations for increasing of arithmetic algorithms scalability. In *15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numeric SCAN'2012, Novosibirsk, Russia, September 23–29, 2012: Book of Abstracts.* Institute of Computational Technologies Publisher, 2012.

[3] A.V. Panyukov, V.A. Golodov.Scalability of algorithms for arithmetic's operations in radix notation. In *GPU Technology Conference 2013 (San Jose, California,*

March 18–21, 2013) Nvidia Corporation, Santa Clara, 2013.
Electronic version avaiable at `http://on-demand.gputechconf.com/gtc/2013/poster/pdf/P0174_Panyukov.pdf`.

[4] O. Beaumont, B. Philippe. Linear interval tolerance problem and linear programming techniques. *Reliable Computing*, 6(4):365–390, 2001.

[5] G.E. Coxson. Computing exact bounds on elements of an inverse interval matrix is NP-hard. *Reliable Computing*, 5(2):137–142, 1999.

[6] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley Longman, 2nd edition, 1981.

[7] C. Keil, C. Jansson. Computational experience with rigorous error bounds for the netlib linear programming library. *Reliable Computing*, 12(4):303–321, 2006.

[8] V.V. Gorbik, A.V. Panyukov. Exact and guaranteed accuracy solutions of linear programming problems by distributed computer systems with MPI. *Tambov University Reports. Series: Natural and Technical Sciences*, 15(4):1392–1404, 2010.

[9] V.V. Gorbik, A.V. Panyukov. Using massively parallel computations for absolutely precise solution of the linear programming problems. *Automation and Remote Control*, 73(2):276–290, 2012.

[10] V.A. Golodov, A.V. Panyukov. Computing the best possible pseudo-solutions to interval linear systems of equations. In *15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numeric (SCAN'2012, Novosibirsk, Russia, September 23-29, 2012): Book of Abstracts*, pages 134–135. Institute of Computational Technologies, 2012.

[11] V.A. Gorbik, A.V. Panyukov, M.I. Germanenko. Library of classes "exact computational". State registration No. 2009612777 on May 29, 2009. In *Programs for Computers, Data bases, Topology of VLSI. Official Bulletin of Russian Agency for Patents and Trademarks*, number 3, page 251. Federal Service for Intellectual Property, 2009.

[12] S.Yu. Lesovoy, A.V. Panyukov. Application of massive-parallel calculations for the realization of the basic operations of the integral arithmetic. In *Performance of Parallel Calculations on the Cluster Systems (HPC - 2010). Materials of the X International Conference (Perm', on November 1 to 3, 2010.). In the 2nd volumes.*, volume 2, pages 77–84. Perm: Publishing house of PermGTU, 2010.

[13] S.Yu. Lesovoy, A.V. Panyukov. Implementing basic operations of integer arithmetic in the heterogeneous systems. In *Parallel Computational Technologies (PaVT' 2012) [electronic resource]. - Proceedings of International Scientific Conference PaVT'2012, Novosibirsk, March 26–30, 2012*, pages 77–84. Chelyabinsk: SUSU Publishing Center, 2012.