UDC 004

DATAFLOW NETWORK OF CAL-ACTORS FOR ALL-PAIR SHORTEST PATHS SEARCH

Prihozhy A. A.
Belarusian National Technical University, Minsk, Belarus
prihozhy@yahoo.com

CAL is a high-level actor-based dataflow language [1–6]. A CAL-program is defined as a network of actors that interact and communicate by sending and receiving data (tokens) along data lossless and order preserving communication channels. An actor is a computational entity that consists of input and output ports, state variables, actions, and a scheduler. Actors are executed in parallel. When an actor is fired, it consumes tokens from input ports, changes the internal state and produces tokens on output ports. An action is a piece of computation that an actor performs in the firing process. An actor may contain any number of actions. When an actor is being firing, it selects one of them based on the availability of input tokens and optionally based on conditions relating to the values of tokens and state variables. An action guard enables conditional action firing. A finite state machine allows actions to be scheduled according to the current state of the actor and action priorities.

The shortest path problem in a weighted directed cyclic graph [7–13] has many application domains. Although a variety of shortest path algorithms in different settings exist, the scaling and parallelization of the problem on multi-processor systems are still open. Analysis and simulation of the Floyd-Warshall (*FW*) and Blocked Floyd-Warshall (*BFW*) all-pair shortest paths algorithms [10–11] have shown that the *BFW* is more suitable for parallelization and speeding up the computations. Moreover, it supports spatial data locality within block and leads to the reduction of data transfer in hierarchical memory and to decreasing the overall execution time. The authors of work [14] proposed an advanced heterogeneous blocked all-pair shortest paths algorithm. A drawback of the algorithms is the realization of fork-join parallelism, which makes them slower against network algorithms. Such computer architectures as multi-core systems include a set of cores and a hierarchical memory consisting of local and shared cache levels, which differ on memory capacity and data transfer time delays. The cores read and write data through fast local caches, and therefore efficiently execute algorithms which support spatial and temporal locality in big data processing.

The advantage of the blocked Floyd–Warshall (*BFW*) Algorithm 2 proposed in [10, 11, 14] is the introduction of spatial data locality due to decomposing matrix $D[N \times N]$ of graph edge weights into blocks of size $S \times S$ each and forming a blocked matrix $B[M \times M]$, where $M = N / S$ is the number of blocks per row. The algorithm provides sequential data locality within each block. Its main loop has M iterations, S times less compared to *FW*. Every iteration of the loop recalculates each block once and tries to update each element S times. Totally, every element of matrix D has N attempts to update. The block recalculation is performed locally by using from one to two other blocks. Algorithm 2, *BCA* implements the *FW* algorithm, recalculates block B^1 consuming two additional blocks B^2 and B^3. It is possible to choose the block size in such a way as the processed blocks could be deployed in fast caches simultaneously, which reduces the data traffic between memory levels. The *BFW* algorithm operation time crucially depends on which level of memory the matrix D fits completely in. If in level L1 and/or L2 do, the algorithm runs quickly. If level L3 does, the data transfer time delay is higher, and the algorithm runs slower. The slowest case takes place when the size of matrix D is larger than the size of cache L3; elements of D are read from and written to main memory many times, which produces big cache pressure and consumes much time.

To speed up the shortest paths algorithm search due to asynchronous behavior [15–17], the paper introduces a dataflow network of actors, which is realized in the CAL language [1]. It also presents a CAL-engine implemented in the C/C++ language as a multi-threaded application on multi-core systems. The CAL-engine efficiently accounts for features of the dataflow network.

Algorithm 1: Blocked Floyd–Warshall (*BFW*)

———————————————————————

Input: A number N of graph vertices
Input: A matrix W of graph edge weights
Input: A size S of block
Output: A blocked matrix B of path distances

```
M ← N / S     B[M×M] ← W[N×N]
for m ← 1 to M do
    BCA (B_m,m, B_m,m, B_m,m)                  // D0
    for i ← 1 to M do
        if i ≠ m then
            BCA (B_i,m, B_i,m, B_m,m)          // C1
            BCA (B_m,i, B_m,m, B_m,i)          // C2
    for i ← 1 to M do
        if i ≠ m then
            for j ← 1 to M do
                if j ≠ m then
                    BCA (B_i,j, B_i,m, B_m,j)  // P3
return B
```

———————————————————————

Algorithm 2: Block calculation (*BCA*)

———————————————————————

Input: A size S of block
Input: Blocks B^1, B^2 and B^3
Output: B^1 – recalculated block

```
for k ← 1 to S do
    for i ← 1 to S do
        for j ← 1 to S do
            sum ← b²_i,k + b³_k,j
            if b¹_i,j > sum then b¹_i,j ← sum
return B¹
```

Let assume the D matrix be mapped to a B[3×3] blocked matrix. In this case, we can model each block B_{rc} by a CAL actor for which M and B are global variables. Figure 1 shows the block-actor interface of four input ports, L_{r1}, L_{r2}, L_{1c} and L_{2c}, and two output ports, L_r and L_c. The input ports receive tokens from output ports of other actors, which describe the calculation level of associated blocks. The output ports L_r and L_c describe the B_{rc} block calculation level that is used by other blocks located in row r and column c. We recognize two types of block-actors: diagonal and non-diagonal. Algorithm 3 depicts the behavior of CAL-actor *Block*_0_0 that models the calculation of diagonal block B_{00}. Input ports *L*_0_1 and *L*_0_2 describe the calculation level of the B_{01} and B_{02} blocks in row 0. Input ports *L*_1_0 and *L*_2_0 describe the calculation level of the B_{10} and B_{20} blocks in column 0. Output ports *Lrow* and *Lcol* describe the calculation level of block B_{00}. State variables *Lev*, *Row* and *Col* describe the calculation level, row and column respectively of block B_{00}. The *Block*_0_0 actor contains three actions: one diagonal and two peripherals. The input and output tokens, and the guard condition of the diagonal action distinct from those of the peripheral one. The diagonal action has no input token and has two output tokens associated with the *Lrow* and *Lcol* ports and getting the value of state variable *Lev*. The guard condition requires *Lev* to be equal to *Row*. The action body increments the block calculation level and calls the *BCA* function to recalculate the diagonal block. The action is fired when the block calculation level is equal to its row and column. Each of the two peripheral actions has three input and no output tokens. In the first action, two input tokens *L*01 and *L*10 arrive from ports *L*_0_1 and *L*_1_0, and the third token is a constant *k*. The guard condition requires *Lev* be lower than the calculation levels *L*01 and *L*10. The action body increments the block calculation level and calls the *BCA* function to recalculate the B_{rc} block over the B_{rk} and B_{kc} blocks. The peripheral action is fired when the input tokens have arrived and the block B_{rc} calculation level is lower than those of blocks B_{rk} and B_{kc}.

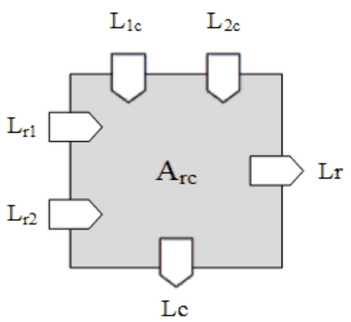

Figure 1 – Interface of the A_{rc} CAL-actor that models the calculation of block B_{rc} in matrix *D*[3×3]

Algorithm 4 depicts the CAL-actor *Block*_0_1 that models the calculation of non-diagonal block B_{01}. Input ports *L*_0_0 and *L*_0_2 describe the calculation level of B_{00} and B_{02} blocks in row 0. Input ports *L*_1_1 and *L*_2_1 describe the calculation level of B_{11} and B_{21} blocks in column 1. Output ports *Lrow* and *Lcol* describe the calculation level of blocks B_{01}. State variables *Lev, Row* and *Col* describe the calculation level, row and column of block B_{01}.

Algorithm 3: CAL-actor *Block*_0_0 calculating diagonal block B_{00} of matrix *B*[3×3]

```
actor Block_D ()   int L_0_1, int L_0_2, int L_1_0, int L_2_0 ==>  int Lrow, int Lcol:
   int Lev:= 0;
   int Row:= 0;
   int Col:= 0;
   Diagonal: action    ==>  Lrow: [Lev], Lcol: [Lev]
   guard   Lev = Row
   do
      Lev:= Lev + 1;
      BCA (B[Row * M + Col], B[Row * M + Col], B[Row * M + Col]);
   end
   Peripheral_1: action  L_0_1: [L01], L_1_0: [L10], const#1:[k] ==>
   guard   L01 >= Lev + 1 and L10 >= Lev + 1
   do
```

```
    Lev:= Lev + 1;
    BCA (B[Row * M + Col], B[Row * M + k], B[k * M + Col]);
  end
  Peripheral_2: action L_0_2: [L02], L_2_0: [L20], const#2:[k] ==>
  guard  L02 >= Lev + 1 and L20 >= Lev + 1
  do
    Lev:= Lev + 1;
    BCA (B[Row * M + Col], B[Row * M + k], B[k * M + Col]);
  end
end
```

Algorithm 4: CAL- actor *Block*_0_1 calculating non-diagonal block B_{01} of matrix *B*[3×3]

```
actor Block_N ()  int L_0_0, int L_0_2, int L_1_1, int L_2_1 ==> int Lrow, int Lcol:
  int Lev:= 0;
  int Row:= 0;
  int Col:= 1;
  Cross1: action L_1_1: [L11] ==> Lrow: [Lev]
  guard  Col = L11 - 1 and Lev = L11 – 1
  do
    Lev := Lev + 1;
    BCA (B[Row * M + Col], B[Row * M + Col], B[Col * M + Col]);
  end
  Cross2: action L_0_0: [L00] ==> Lcol: [Lev]
  guard  Row = L00 - 1 and Lev = L00 – 1
  do
    Lev:= Lev + 1;
    BCA (B[Row * M + Col], B[Row * M + Row], B[Row * M + Col]);
  end
  Peripheral_3: action L_0_2: [L02], L_2_1: [L21], const#2:[k] ==>
  guard  L02 >= Lev + 1 and L21 >= Lev + 1
  do
    Lev:= Lev + 1;
    BCA (B[Row * M + Col], B[Row * M + k], B[k * M + Col]);
  end
end
```

Actor *Block*_0_1 contains three actions: *Cross1*, *Cross2* and *Peripheral*. The *Cross1* action has an input token *L*11 associated with port *L*_1_1, and has an output token associated with port *Lrow* and getting the value from state variable *Lev*. The guard condition requires *Col* and *Lev* be equal to *L*11–1. The action body increments the block calculation level and calls the *BCA* function to recalculate block B_{01} over block B_{11}. The *Cross1* action is fired when a token arrives at its input port and its guard condition evaluates to true. The behavior of *Cross2* action is similar to those of *Cross1* action. The behavior of the Peripheral action in a non-diagonal actor is the same as those in the diagonal one.

Composing the actors into a network together with setting connections among input and output ports and locating buffers at the input ports establish a dataflow network. The network is a coordination model of the concurrent actor computation. Fig. 2 shows a network of nine actors for the $B[3 \times 3]$ matrix. Three actors are diagonal, and six actors are non-diagonal. Right output ports connect actors along rows. Bottom output ports connect actors along columns. All actors can be fired simultaneously.

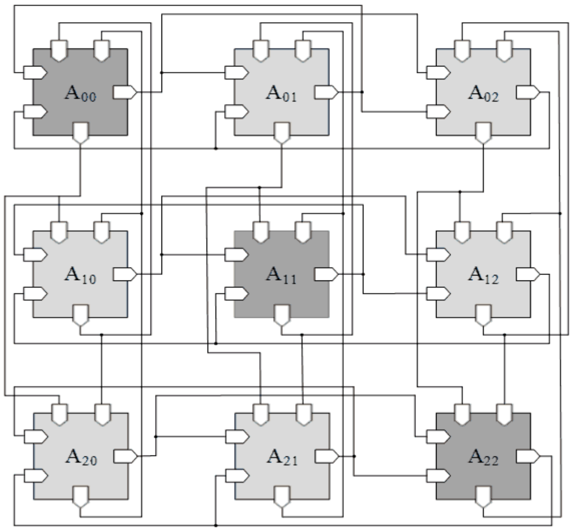

Figure 2 – CAL network of two types block actors for $B[3\times3]$ matrix

To implement the behavior of actors, actions and dataflow network on a multi-core system, we have developed a C/C++ based CAL language engine. Every component needed for the CAL runtime system is implemented by means of an appropriate C/C++ class of objects. As a result, a CAL network and each of its actors are instantiated via complex data structures and a set of methods in C/C++. All actors operate concurrently. One actor sends a flow of tokens to other actors through buffers and ports. Since several actors update common variables in buffers, the engine synchronizes the actor’s communications and the data processing.

We have generated dataflow actor networks for various block-matrix size, M and have done experiments on randomly generated weighted complete graphs of 1200, 2400 and 3600 vertices. Experimental results shown in fig. 3 are obtained on the i7-9750h processor: 6 cores, 12 logical processors, 2.60 GHz of frequency, and 16 GB of main memory. Fig. 3 compares the speedup the multi-threaded dataflow CAL-networks implementing BFW have given against the single-thread FW implementation. On the block-matrix of 1200 graph vertices, the highest speedup of about 5 has been obtained. On 2400 graph vertices providing higher load of cores, the CAL-network has given the speedup larger than the number of cores (more than 6) on matrices $B[5\times5]$, $B[6\times6]$ and $B[7\times7]$. On 3600 graph vertices the speedup is even larger (about 7) on matrix $B[6\times6]$. On 3600 graph vertices the speedup is even larger (about 7) on matrix $B[6\times6]$.

Conclusion. The paper has considered the solving of all-pair shortest paths problem on large graphs by means of the dataflow computation model of the CAL actor language. The proposed CAL language engine-based actor, action and dataflow network models constitute a high-performance scalable parallel implementation of blocked shortest paths algorithms on multicore systems.

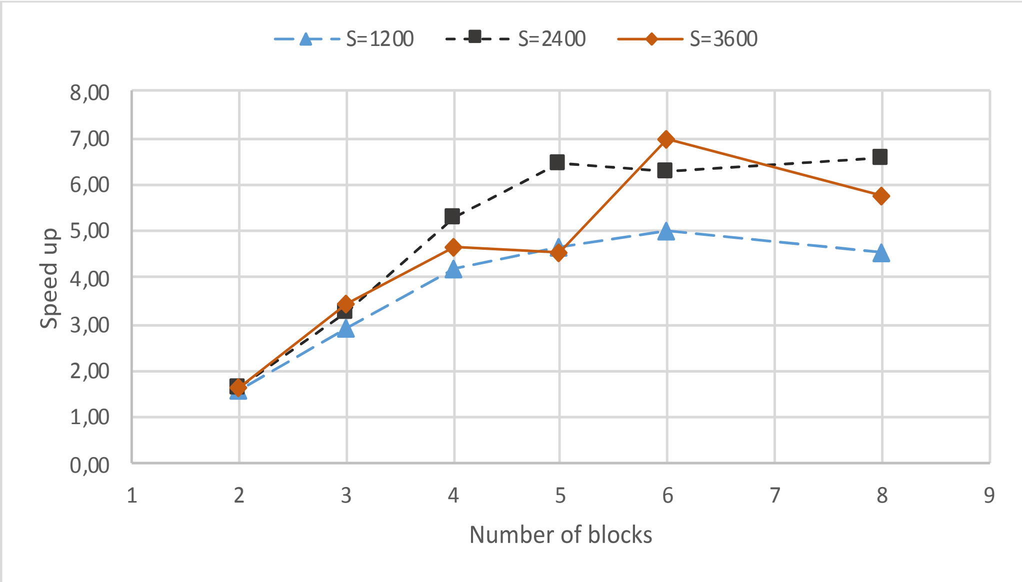

Figure 3 – Speedup (vertical axis) of multi-threaded dataflow network *BFW* against single-thread *FW* vs. *M* (horizontal axis) on graph size of 1200, 2400 and 3600 vertices on i7-9750h processor

References

1. Eker J., Janneck J. CAL language report: Specification of the CAL actor language. Dept. UCB/ERL, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. M03/48, Dec. 2003.

2. Mattavelli M., Amer I., Raulet M. The Reconfigurable Video Coding Standard. [Standards in a Nutshell], Signal Processing Magazine, IEEE 27 (3) (2010) 159 –167 p.

3. Prihozhy A. A., Casale-Brunet S., Bezati E., Mattavelli M. Pipeline Synthesis and Optimization from Branched Feedback Dataflow Programs. Journal of Signal Processing Systems, Springer Nature, 2020, Vol. 92. – 1091–1099 p.

4. Prihozhy A. A., Casale-Brunet S., Bezati E., Mattavelli M. Efficient Dynamic Optimization Heuristics for Dataflow Pipelines. IEEE International Workshop on Signal Processing Systems, IEEE. – 337–342 p.

5. Rahman A. H. Ab., Prihozhy A. A., Mattavelli M. Pipeline synthesis and optimization of FPGA-based video processing applications with CAL. EURASIP Journal on Image and Video Processing, vol. 2011:19. – 1–28 p.

6. Prihozhy A. A., Mattavelli M., Mlynek D. Evaluation of Parallelization Potential for Efficient Multimedia Implementations: Dynamic Evaluation of Algorithm Critical Path. IEEE Trans. on Circuits and Systems for Video Technology, Vol. 15, No. 5. – 593–608 p.

7. Floyd R. W. Algorithm 97: Shortest path. Communications of the ACM, 1962, 5(6). – 345 p.

8. Madkour A, Aref W. G., Rehman F. U., Rahman M. A., Basalamah S. A. Survey of Shortest-Path Algorithms. – 26 p.

9. Prihozhy A. A., Mattavelli M., Mlynek D. Data dependences critical path evaluation at C/C++ system level description. International Workshop PATMOS' 2003, Springer, Berlin, Heidelberg. – 2003. – 569–579 p.

10. Venkataraman G., Sahni S., Mukhopadhyaya S. A. Blocked All-Pairs Shortest Paths Algorithm. Journal of Experimental Algorithmics (JEA), Vol 8, 2003. – 857–874 p.

11. Park J. S., Penner M., Prasanna V. K. Optimizing graph algorithms for improved cache performance. IEEE Trans. on Parallel and Distributed Systems, 2004, 15(9). – 769–782 p.

12. Карасик О. Н., Прихожий А. А. Потоковый блочно-параллельный алгоритм поиска кратчайших путей на графе. Доклады БГУИР. – 2018. – № 2. –77–84 с.

13. Albalawi E., Thulasiraman P., Thulasiram R. Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0. 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013), 2013, Los Angeles, CA, July 1–2, 2013. – 109–112 p.

14. Прихожий, А. А. Разнородный блочный алгоритм поиска кратчайших путей между всеми парами вершин графа / А. А. Прихожий, О. Н. Карасик // Системный анализ и прикладная информатика. – № 3. – 2017. – 68–75 с.

15. Прихожий А. А., Ждановский А. М., Карасик О. Н., Маттавелли М. Эвристический генетический алгоритм оптимизации вычислительных конвейеров. Доклады БГУИР, 2017, № 1/ – 34–41 с.

16. Прихожий, А. А. Распределенная и параллельная обработка данных. – Минск: БНТУ, 2016. – 90 с.

17. Prihozhy A. A., Mlynek D., Solomennik M., Mattavelli M. Techniques for Optimization of Net Algorithms. PARELEC 2002 – Parallel Computing in Electrical Engineering, IEEE CS Press, 2002. – 211–216 p.