UDC 004.272.2 (075.8)

PRIHOZHY A. A.

OPTIMIZATION OF DATA ALLOCATION IN HIERARCHICAL MEMORY FOR BLOCKED SHORTEST PATHS ALGORITHMS

Belarusian National Technical University

This paper is devoted to the reduction of data transfer between the main memory and direct mapped cache for blocked shortest paths algorithms (BSPA), which represent data by a D[M×M] matrix of blocks. For large graphs, the cache size S = δ×M2, δ < 1 is smaller than the matrix size. The cache assigns a group of main memory blocks to a single cache block. BSPA performs multiple recalculations of a block over one or two other blocks and may access up to three blocks simultaneously. If the blocks are assigned to the same cache block, conflicts occur among the blocks, which imply active transfer of data between memory levels. The distribution of blocks on groups and the block conflict count strongly depends on the allocation and ordering of the matrix blocks in main memory. To solve the problem of optimal block allocation, the paper introduces a block conflict weighted graph and recognizes two cases of block mapping: non-conflict and minimum-conflict. In first case, it formulates an equitable color-class-size constrained coloring problem on the conflict graph and solves it by developing deterministic and random algorithms. In second case, the paper formulates a problem of weighted defective color-count constrained coloring of the conflict graph and solves it by developing a random algorithm. Experimental results show that the equitable random algorithm provides an upper bound of the cache size that is very close to the lower bound estimated over the size of a complete subgraph, and show that a non-conflict matrix allocation is possible at δ = 0.5 for M = 4 and at δ = 0.1 for M = 20. For a low cache size, the weighted defective algorithm gives the number of remaining conflicts that is up to 8.8 times less than the original BSPA gives. The proposed model and algorithms are applicable to set-associative cache as well.

Keywords: *shortest paths algorithm, hierarchical memory, direct mapped cache, performance, block conflict graph, data allocation, equitable coloring, defective coloring.*

Introduction

The shortest paths search problem in weighted graphs is formulated in different settings [1–4]. The all-pair shortest paths problem (APSP) has many application domains: from the city traffic optimization to computer games. Although the APSP algorithms (including the Floyd-Warshall one) have polynomial computational complexity and have been studied for a long time, their realization on modern multi-processor computing systems is still an attractive research area since actual graphs can reach very large sizes.

The parallel APSP algorithm execution time mostly depends on how it distributes the work among the processor cores and what is the throughput and load of each core. The hierarchical memory is also a key contributor in the execution time [5, 6]. Caches are intermediate level between the CPU and main memory, which accelerate the data access. If a program accesses data and the data is not in cache, a miss has occurred. The key step in improving the cache performance is reducing the miss rate [7–9].

The hierarchical memory employs three strategies of mapping main memory blocks to cache blocks: direct mapping, set-associative mapping and full-associative mapping. Usually the cache stores a small number of blocks against the main memory. That is why the main memory blocks are grouped when mapping to a cache block. When executing an algorithm, blocks of the same group compete for the cache block. Conflicts may occur among the blocks simultaneously requested. Optimizing the distribution of the set of blocks on the set of groups may greatly reduce the conflict count and the data miss rate.

The temporal and spatial localities [11] associated with data accesses the executed algorithm generates allow a reduction of data misses in the cache. The locality can also help in the efficient allocation of data in the main memory. The paper considers a complement for the locality approach, which allocates data [12–14] of a blocked algorithm in such a way that maps the conflicting blocks of the slow main memory to different

block locations of the fast cache. The placement order of the main memory blocks determines a group associated with each cache block.

The paper formulates the data allocation problem for blocked shortest paths algorithms, proposes a block conflict weighted graph model, and develops efficient extensions of equitable and defective coloring algorithms targeting the minimization of cache size, decreasing the number of remaining conflicts among blocks, and reduction of the algorithm execution time.

Blocked all pairs shortest paths algorithms

Let $G = (V, E)$ be a directed weighted graph, where $V=\{0,\ldots, N\text{-}1\}$ and $E \subseteq \{(i, j) \mid i, j \in V\}$ are the vertex and edge sets respectively. A weight function assigns a weight w_{ij} to an edge $(i, j) \in E$. Matrix W represents the function, in which $W(i, j) = 0$ if $i = j$, $W(i, j) = w_{ij}$ if $(i, j) \in E$, and $W(i, j) = \infty$ if $(i, j) \notin E$.

The all-pair shortest paths problem is formulated as to find the paths of the shortest length between all pairs of vertices, $i, j \in V$. The Floyd–Warshall (*FW*) algorithm [1, 2] uses a matrix D that describes the all-pair shortest path lengths. The algorithm computational complexity is $O(N^3)$. For large matrices, the execution time of *FW* is high, and a significant part of the time is due to the hierarchical memory operation.

Let the matrix $D[N\times N]$ be blocked resulting in a $M\times M$ matrix of smaller matrices B_{ij}, $0 \le i, j < B$, where $B = N / M$. Algorithm 1 known as the blocked Floyd–Warshall (*BFW*) [3], iteratively calls a function *BCA* (B^1, B^2, B^3) realized by Algorithm 2 of calculating block B^1 over blocks B^2 and B^3. Figure 1 illustrates the behavior of *BFW* on matrix $D[4\times4]$. In an Iteration, *BFW* calculates the diagonal $D0$ block, blocks $C1$ and $C2$ of cross, and peripheral blocks $P3$, and moves the cross from the left-top corner to the right-bottom one. Work [4] extended *BFW* to the heterogeneous four-type-block algorithm *HBFW*. *BSPA* denotes both *BFW* and *HBFW*. The computational complexity of *BSPA* and *FW* is the same. *BSPA's* advantage is the ability to localize data and computations within blocks, which is important for efficient cache operation, and for the organization of parallel computation of blocks [7–9]. *BSPA* does not worry about allocating data in hierarchical memory.

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

Input: A number N of graph vertices
Input: A matrix W of graph edge weights
Input: A size B of block
Output: A matrix D of lengths of all-pair shortest paths

```
M ← N / B    D[M×M] ← W[N×N]
for m ← 0 to M -1 do
    BCA (B_m,m, B_m,m, B_m,m)                    // D0
    for i ← 0 to M -1 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 ← 0 to M -1 do
        if i ≠ m then
            for j ← 0 to M -1 do
                if j ≠ m then
                    BCA (B_i,j, B_i,m, B_m,j)    // P3
return D
```

Algorithm 2: Block calculation algorithm (*BCA*)

Input: B – size of block
Input: B^1 – first input block
Input: B^2 – second input block
Input: B^3 – third input block
Output: B^1 – recalculated block

```
for k ← 0 to B -1 do
    for i ← 0 to B -1 do
        for j ← 0 to B -1 do
            sum ← B²_ik + B³_kj
            if B¹_ij > sum then B¹_ij ← sum;
return B¹
```

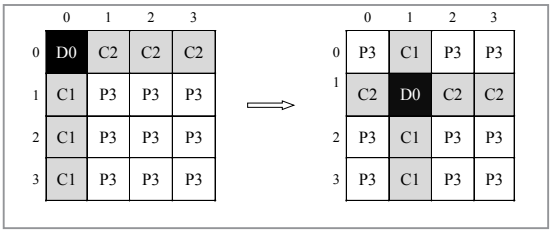


Fig. 1. Illustration of BFW operation

Formulation of data allocation problem

In blocked algorithms that processes big data the overall size of blocks is larger than the available cache size, therefore several blocks are mapped to the same slots of the direct mapped cache (Fig.2). Thus, the main memory blocks 0, 4, … are assigned to the slot group 0 of cache. A problem arises when the executed program accesses simultaneously blocks 0 and 4. In this case, the blocks are in conflict, the cache flaking takes place, and the program execution slows down significantly. An appropriate allocation of blocks in the main memory can solve the problem. The conflicting blocks have to be assigned to different cache slots. This leads to reordering of blocks in the main memory. The exhaustive analysis of the executed algorithm is a way to

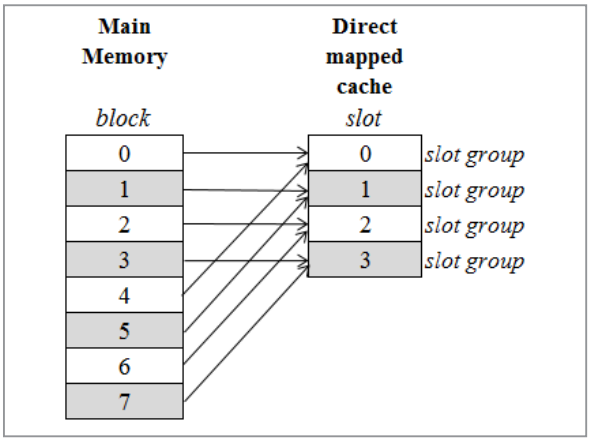


Fig. 2. Mapping memory blocks to slot groups of direct mapped cache

the construction of a non-conflict or minimum-conflict block allocation. The paper proposes a model of weighted block-conflict graph, which allows for *BSPA* to find a block placement with a minimum number of conflicts.

Weighted block-conflict graph

Figure 3 shows an enumeration and initial row-major memory layout of 16 blocks of matrix $D[4\times4]$ in the main memory. Fig. 4 depicts a matrix of block conflict ternary relation. In the matrix, every filled cell indicates a tuple (i, j, w) of the relation where w is a conflict count between the blocks i and j. For *BSPA*, $w \in \{1, 2\}$. For instance, the cell (0, 5) indicates the absence of conflicts between blocks 0 and 5 and does not describes a tuple. The cell (0, 12) describes a tuple

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Fig. 3. Initial placement of blocks of matrix $D[4\times4]$ in main memory

(0, 12, 2) that indicates the presence of 2 conflicts between blocks 0 and 12.

In Fig. 4, two right columns *edge* and *weight* describe for each block the number of other conflict blocks and the overall conflict count respectively. For instance, block 0 has six other conflict blocks with the overall conflict count of 12.

A weighted undirected graph $G_T = (T, C)$, where T is a set of blocks and C is a set of weighted edges (Fig. 5), is an alternative representation of the conflict relation. An edge (i, j) $\in C$ has a weight (conflict count) w(i, j). In Figure 5, the edges represented by solid lines have the weight of 2, and the dash-line edges have the weight of 1.

Assertion 1. Graph G_T has a complete subgraph whose chromatic number is $2\times M-1$.

A proof of the assertion is based on the consideration of a subgraph constructed of the vertices, which correspond to the $2\times M-1$ blocks of a cross. It shows that all the vertices are adjacent in the graph.

The number $2\times M-1$ is a lower bound of the conflict graph chromatic number $\chi(G_T)$. Thus, the

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	edge	weight
0	0	2	2	2	2				2				2				6	12
1		0	2	2	2	2	1	1	1	2			1	2			11	18
2			0	2	1		2		2	1	2	1	1		2		11	18
3				0	1			2	1			2	2	1	1	2	11	18
4					0	2	2	2	2	1			2	1			11	18
5						0	2	2		2				2			6	12
6							0	2	1	2	2	1		1	2		11	18
7								0		1		2	1	2	1	2	11	18
8									0	2	2	2	2		1		11	18
9										0	2	2		2	1		11	18
10											0	2			2		6	12
11												0	1	1	2	2	11	18
12													0	2	2	2	11	18
13														0	2	2	11	18
14															0	2	11	18
15																0	6	12

Fig. 4. Block conflict relation for D[4×4]

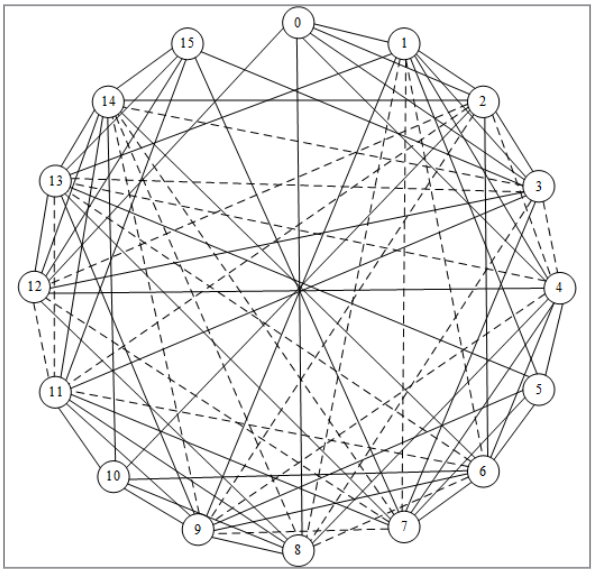


Fig. 5. Block conflict graph G_T for D[4×4]:
edges of weight 2 are solid and edges of weight 1 are dash

graph for matrix D[4×4] has a chromatic number lower bound of 7.

Non-conflict allocation of matrix blocks

In work [15], the authors proposed a graph coloring technique for minimizing the storage consumed by an algorithm. The technique models and evaluates the lifetime of each variable and assigns two variables to the same memory location if their lifetimes are not intersected.

A proper coloring of the graph G_T is a mapping μ: $T \to R_\mu$ of a set T of vertices to a set R_μ of colors so that for two adjacent vertices t_i, $t_j \in T$ the inequality $\mu(t_i) \neq \mu(t_j)$ holds. A color class $T_\mu(r) \subseteq T$ is a set of vertices labeled by a single color $r \in R_\mu$. In a properly colored graph, each color class is an independent vertex set. Let the color classes $T_\mu(1) \cup \ldots \cup T_\mu(\chi) = T$ represent the coloring μ where $\chi = |R_\mu|$. Let Ω be a set of all proper colorings of graph G_T. Then the chromatic number of G_T is

$$\chi(G_T) = \min_{\mu \in \Omega} |R_\mu| \qquad (1)$$

The chromatic number $\chi(G_T)$ determines the size of direct mapped cache that is sufficient for non-conflict allocation of matrix $D[M \times M]$. Let $o(G_T)$ be a maximum color class size in the μ coloring. Then (2) determines the number $\rho(G_T)$ of blocks needed for proper allocation of the matrix in the main memory.

$$\rho(G_T) = \chi(G_T) \times o(G_T) \qquad (2)$$

The inequality $\rho(G_T) \geq M^2$ must hold, and $\eta = \rho(G_T) - M^2$ is the number of garbage blocks that are added to matrix D.

Fig. 6 shows a result of applying the coloring technique to the block conflict graph G_T depicted in Fig. 5. The graph chromatic number $\chi(G_T)$ equals 7. The maximum color class size $o(G_T)$ equals 4. The number of blocks equals 16. As many as 28 main memory blocks are needed for the non-conflict allocation of D[4×4]. Fig. 6*a* depicts the mapping of 16 block-vertices to 7 colors. Fig. 6*b* depicts the assignment of blocks to the cache slot groups and the placement of the blocks in main memory. A filled cell represents a garbage block denoted by 'x'. Since the color classes have different size, the placement 0, 1, 2, 3, 4, 8, 9, 5, 11, 7, 6, 14, 13, 12, 10, x, x, x, x, x, x, 15, x, x, x, x, x, x provides a big fragmentation of main memory.

Optimization of non-conflict block allocation

The section targets two goals: first to minimize the size of cache that supports a non-conflict block allocation, and second to reduce the main memory fragmentation. Fig. 6*b* shows that the known coloring algorithm has introduced too many garbage blocks. This is because the algorithm minimizes the number of colors by generating a color class of possibly maximal size for each color, which leads to high value of $o(G_T)$ and to misbalancing of cache slot load. As a result, the cache size and main memory fragmentation are large. The algorithm is not capable of generating a satisfactory block matrix placement.

Work [16] introduces equitable coloring, which aims at balancing the size of color classes. It assign colors to vertices in such a way that no two adjacent vertices have the same color, and

	0	1	2	3
0	0	1	2	3
1	4	0	3	2
2	5	6	0	1
3	6	5	4	0

a

Slot group	Blocks			
0	0	5	10	15
1	1	11	x	x
2	2	7	x	x
3	3	6	x	x
4	4	14	x	x
5	8	13	x	x
6	9	12	x	x

b

Fig. 6. Coloring technique application:
a) colors of blocks in matrix D[4×4];
b) assignment of blocks to slot groups of cache

the numbers of vertices in any two color classes differ by at most one. The Hajnal–Szemerédi theorem [17] proves that any graph with maximum degree Δ has an equitable coloring with Δ + 1 colors. The theorem applied to the graph with Δ = 11 (Fig. 5) gives the color count of 12, which is much larger than the graph chromatic number of 7 (Fig. 6). It means the theorem provides a too pessimistic solution that is not practically acceptable.

We introduce a color-class-size constraint *CSC* and formulate a new csc-coloring problem on graph G_T to find a constrained chromatic number $\gamma(G_T)$:

$$\text{minimize } \gamma(G_T) = \min_{\mu\in\Omega} \left|R_\mu\right| \qquad (3)$$

subject to

$$\left|T_\mu(r)\right| \le CSC\ ,\ \mu\in\Omega \text{ and } r\in R_\mu \qquad (4)$$

The *CSC* constraint describes a requirement for the number of blocks assigned to the same slot group in cache. The formulation aims at both obtaining a low fragmentation of main memory and minimizing the cache size.

Color-class-size constrained coloring algorithms

Since the graph chromatic number problem is NP-hard, we propose two heuristic color-class-size constrained coloring algorithms: Algorithm 3 is a constrained deterministic graph coloring (*CDGC*), and Algorithm 4 is a constrained random graph coloring (*CRGC*).

CDGC traversals all vertices and chooses an earlier introduced proper color if any; otherwise, it adds a new color and assigns it to the current vertex. The color is proper if it does not label an adjacent vertex and its vertex class size does not exceed *CSC*. *CRGC* randomly generates many proper csc-colorings and returns the best of them as output. While generating the next coloring, it randomly selects an uncolored vertex and randomly selects an earlier introduced proper color if any; otherwise, it adds a new color and assigns it to the current vertex.

We have realized the both algorithms and conducted experiments on various matrix configurations. Fig. 7 reports results the *CRGC* algorithm obtained for the *D*[4×4] matrix. Fig. 7*a* depicts the optimal csc-coloring of 16 blocks. Fig. 7*b* depicts the optimal placement of the blocks in the main

Algorithm 3: Constrained deterministic graph csc-coloring (*CDGC*)

Input: A weighted undirected graph $G_T = (T, C)$ block conflicts
Input: A number M^2 of blocks in set T
Input: A conflict relation C
Input: A constraint *CSC* on the color class size
Output: A vector *Coloring* of vertex colors in graph G_T
Output: A chromatic number γ of graph G_T

```
Colors ← ∅
for b ← 0 to M^2 do
    AvailColor ← undefined
    for c ∈ Colors do
        if UseCnt(c) < CSC then
            flag ← true
            for bc ← 0 to b–1 do
                if Coloring(bc) = c and (b, bc) ∈ C then
                    flag ← false
                    break
            if flag then
                AvailColor ← c
                break
    if AvailColor = undefined then
        AvailColor ← NewColor
        Colors ← Colors ∪ {AvailColor}
        UseCnt(AvailColor) ← 1
    else
        UseCnt(AvailColor) ← UseCnt(AvailColor) + 1
    Coloring(b) ← AvailColor
γ ←|Colors|
return γ, Coloring
```

Algorithm 4: Constrained random graph csc-coloring (*CRGC*)

Input: A weighted undirected graph $G_T = (T, C)$ of block conflicts
Input: A number M^2 of blocks in set T
Input: A conflict relation C
Input: A constraint *CSC* on the color class size
Input: A constraint *RunCount* on the coloring run count
Output: A vector *BestColoring* of vertex colors in graph G_T
Output: A chromatic number γ of graph G_T

```
γ ← ∞
for run ← 1 to RunCount do
    Tcolored ← ∅    Colors ← ∅
    while T \ Tcolored ≠ ∅ do
        Randomly select b ∈ T \ Tcolored
        ColAvailable ← ∅
        for c ∈ Colors do
            if UseCnt(c) < CSC then
                flag ← true
                for bc ∈ Tcolored do
                    if Coloring(bc) = c and (b, bc) ∈ C then
                        flag ← false
                        break
                if flag then
                    ColAvailable ← ColAvailable ∪ {c}
        if |ColAvailable| > 0 then
            Randomly select c ∈ ColAvailable
            Coloring(b) ← c
            UseCnt(c) ← UseCnt(c) + 1
        else
            Colors ← Colors ∪ {NewColor}
            Coloring(b) ← NewColor
            UseCnt(NewColor) ← 1
        Tcolored ← Tcolored ∪ {b}
    if γ > |Colors| then
        γ ← |Colors|
        BestColoring ← Coloring
return γ, BestColoring
```

memory and cache. Table 1 provides a comparison of *CRGC* against *CDGC* on matrix *D*[12×12] depending on the *CSC* constraint.

The comparison concerns three parameters: the cache size, the overall block count in main memory, and the garbage blocks count in overall count. *CRGC* has reduced the cache size by up to 17.1% against *CDGC*. It also introduced much less garbage blocks.

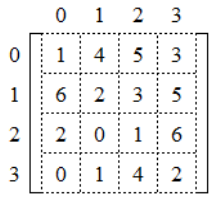
Table 2 reports conflict graph parameters such as the vertex count, edge count, maximum, minimum and average vertex degree, and chromatic number upper bound depending on *M*.

Table 3 reports the lower bound that is evaluated by Assertion 1 and the upper bound that is evaluated by *CRGC* with respect to the cache size, memory size and garbage block count that are sufficient for non-conflict allocation of matrix *D* depending on *M* and *CSC*. If *M* equals 4 and 6, the lower and upper bounds are the same, it means *CRGC* has given a minimum of cache size. If *M* equals 8, 10 and 12, the upper bound of cache size is 1, 2 and 2 blocks respectively that is larger than the lower bound, but the load of a cache block is one memory block lower, and the garbage block count are reduced from 11, 14 and 17 to 0, 5 and 6 respectively. The matrix *D* allocations given by *CRGC* are much better over those given by the lower bound. If *M* equals 5, 7, 9 and 11, the upper bound loses 1, 1, 1 and 2 blocks of the cache size respectively, and has a larger main memory fragmentation against the lower bound. The overall conclusion is in most cases *CRGC* has given optimal results and in other cases has given high quality solutions that are close to optimal ones.

Fig. 8 shows a reduction of the cache size against the main memory size in non-conflict allocation of matrix *D* depending on *M*. It can be observed that the increase in the number of matrix blocks leads to the relative reduction of the cache size from 50% at $M = 4$ down to about 10% at $M = 20$.

Defective weighted coloring algorithm

Defective coloring may color adjacent vertices by the same color [18]. A (k, d)-coloring of a graph is a coloring of its vertices with k colors such that each vertex has at most d neighbors with the same color. The minimum number of colors k required for which the graph is (k, d)-colorable is



a

Slot group	Blocks		
0	9	12	x
1	0	10	13
2	5	8	15
3	3	6	x
4	1	14	x
5	2	7	x
6	4	11	x

b

Fig. 7. Constrained csc-coloring algorithm:
a) block-vertex colors in graph GT;
b) assignment of memory blocks to slot groups in cache and placement of blocks in main memory

Table 1. **Comparison of deterministic and random coloring algorithms regarding the cache size and the overall and garbage block count in main memory for D[12×12]**

Algorithm	Parameter	CSC				
		2	3	4	5	6
CDGC	Cache blocks	75	53	42	35	28
	Memory blocks	150	159	168	175	168
	Garbage blocks	6	15	24	31	24
CRGC	Cache blocks	72	48	36	29	25
	Memory blocks	144	144	144	145	150
	Garbage blocks	0	0	0	1	6
Ran/Det	Cache gain (%)	4.0	9.4	14.3	17.1	10.7

Table 2. **Conflict graph G_T parameters vs. M**

M	6	7	8	9	10	11	12
Vertices	36	49	64	81	100	121	144
Edges	315	525	812	1188	1665	2255	5940
Edges (%)	50.0	44.6	40.3	36.7	33.6	31.1	28.9
Vertex degree max	19	23	27	31	35	39	43
Vertex degree min	10	12	14	16	18	20	22
Vertex degree aver	17.5	21.4	25.4	29.3	33.3	37.3	41.3
Chromatic number	11	14	16	18	20	23	25

Table 3. **Lower and upper bounds of cache size γ, main memory size ρ and garbage block count η sufficient for non-conflict allocation of matrix D vs. M and CSC**

M	Lower bound				Upper bound			
	CSC	γ	ρ	η	CSC	γ	ρ	η
4	3	7	21	5	3	7	21	5
5	3	9	27	2	3	10	30	5
6	4	11	44	8	4	11	44	8
7	4	13	52	3	4	14	56	7
8	5	15	75	11	4	16	64	0
9	5	17	85	4	5	18	90	9
10	6	19	114	14	5	21	105	5
11	6	21	126	5	6	23	138	17
12	7	23	161	17	6	25	150	6

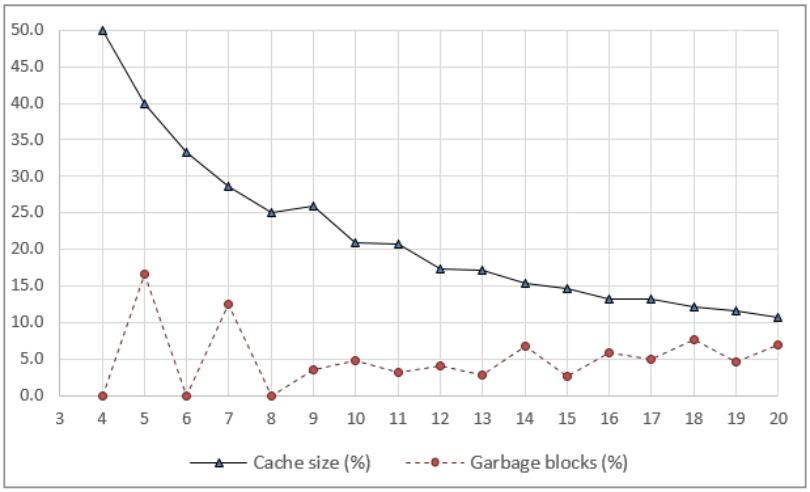


Fig. 8. Cache size (%) over matrix *D* size, and garbage block count (%) over *D* block count in non-conflict allocation vs. *M*

called the *d*-defective chromatic number. The impropriety of a vertex is the number of neighbors that have the same color. The impropriety of the coloring is the maximum of the improprieties of all vertices of the graph.

In the paper, we have extended the concept of defective coloring to the concept of weighted defective coloring μ of graph G_T. In the coloring, at least one color class $T_\mu(r) \subseteq T$, $r \in R_\mu$ is a dependent vertex set. Since the class contains at least one weighted edge, we define a weighted defect with Equation (5).

$$\varphi(T_\mu(r)) = \sum_{i,j \in T_\mu(r)} w(i,j) \qquad (5)$$

A weighted defect of the coloring μ is

$$\Phi(\mu) = \max_{r \in R_\mu} \varphi(T_\mu(r)) \qquad (5)$$

We formulate the defective weighted constrained coloring problem as follows:

$$\text{minimize } \omega(G_T) = \min_{\mu \in \Omega} \Phi(\mu) \qquad (6)$$

subject to

$$\left|R_\mu\right| \le CCC\ ,\ \mu \in \Omega, \qquad (7)$$

$$\left|T_\mu(r)\right| \le CSC\ ,\ \mu \in \Omega \text{ and } r \in R\mu, \qquad (8)$$

$$CCC \times CSC \ge M^2\ , \qquad (9)$$

where *CCC* is a color-count-constraint. In case of $CCC \times CSC = M^2$ a solution of problem (6) – (9) gives a block-matrix allocation without garbage blocks in the main memory and with a minimum of conflicts among blocks assigned to the same cache block. A permutation of *D* matrix blocks represents the allocation.

Fig. 9 depicts a solution for *D*[4×4], *CCC* = 4 and *CSC* = 4. The obtained weighted defect $\omega(G_T)$ is 3 conflicts. In the figure, each column represents a color class corresponding to a single cache block. The allocation of blocks in main memory is: 0, 2, 1, 3, 6, 4, 7, 5, 11, 9, 10, 8, 13, 15, 12, 14.

We have developed Algorithm 5 of defective weighted constrained random coloring (*DWCRGC*) of the conflict graph. The algorithm iteratively generates

	0	1	2	3
0	0	2	1	3
1	6	4	7	5
2	11	9	10	8
3	13	15	12	14

Fig. 9. Defective weighted constrained coloring of *D*[4×4]

RunCount vertex random permutations (*order*) and selects a coloring that has a minimum ω of weighted defect. Each iteration produces a graph vertex coloring that meets the given constraints. After selecting a vertex $u \in T \setminus L$ where $L \subseteq T$ is a subset of already colored vertices, the algorithm chooses a color *c* using seven parameters:

- an overall weighted defect *D*(*c*) on *L*;
- a weighted additional defect *d*(*c*) after including *u* in *c*;
- a maximum defect $D_{max} = max\ D(c)$ over all *c*;
- a maximum defect $d_{max} = max\ d(c)$ over all *c*;
- a weight function *W*(*c*) on *L*, whose maximum value indicate a selected color of vertex *u*;

- a maximum value $W_{\max} = max\ W(c)$ of the weight function over all c;
- a color class *BestC* with $W_{\max}$.

For each run of coloring and each color class c, Algorithm 5 first initializes three variables: a number $vCnt(c)$ of vertices in c, an overall defect $D(c)$ and an additional defect $d(c)$. Then in a loop, it traverses all vertices. For each vertex *block*, it traverses all color classes as candidates for color assignment. For each class c whose cardinality is less than *CSC*, the algorithm calculates the additional defect $d(c)$ using the weights of conflict graph edges. It also calculates $d_{\max}$. Then the algorithm calculates the weight function $W(c)$ of each c using (10), and selects a class *BestC* with the maximum value of $W_{\max}$.

$$W(c) = \alpha \times (D_{\max} - D(c)) / D_{\max} + \\ + \beta \times (d_{\max} - d(c)) / d_{\max}. \quad (10)$$

$W(c)$ depends on two parameters: weighted defect $D(c)$ of c over all colored vertices and additional defect $d(c)$ due to coloring vertex u. In (10), we assume the first term be zero if $D_{\max} = 0$, and the second term be zero if $d_{\max} = 0$. Algorithm 5 adds vertex *block* to class *BestC* and recalculates $D(BestC)$ and $D_{\max}$. After coloring all vertices, the algorithm updates *BestColoring* and its defect ω if the obtained *Coloring* is better than the *BestColoring*.

We have implemented Algorithm 5 in C/C++ and have performed several experiments. Table 4 reports results for $D[6\times6]$ with respect to the weighted defect of the *CSC* constraint and factors α and β. When $\alpha = 1$ the algorithm yields a maximum defect. It gives a lower defect when α is closer to zero (in our experiment at $\alpha = 0.3$). We can explain it as balancing the load among cache blocks ($D(c)$ and $D_{\max}$ are responsible for the balancing) is less important than avoiding conflicts when mapping the main memory blocks to cache blocks ($d(c)$ and $d_{\max}$ are responsible for the avoiding). *CSC* has taken values 3, 4, 6, 9 and 12, which guaranty the absence of garbage blocks at the *D* size of 36. The weighted defect has reduced as 42, 22, 6, 2 and 0 respectively with increasing *CSC*. At *CSC* = 12 the algorithm has generated a non-conflict block allocation.

Table 5 compares the matrix row-major memory defective allocation of *BSPA* (Fig. 3) against the optimized cache allocation (Fig. 9) produced by the defective weighted coloring algorithm *DWCRGC* for matrix $D[M\times M]$ at $M = 4, \ldots, 12$, $CSC = CCC = M$. In both cases, the allocation is defective since the conflict graph chromatic number is larger than M.

Algorithm 5: Defective weighted constrained random conflict graph coloring (*DWCRGC*)

Input: A weighted undirected graph $G_T = (T, C)$ of block conflicts
Input: A number M^2 of blocks in set T
Input: A conflict relation C
Input: A factor α in the objective function
Input: A constraint *CSC* on the color class size
Input: A constraint *CCC* on the color count
Input: A constraint *RunCount* on the coloring run count
Output: A vector *BestColoring* of vertex colors in graph G_T
Output: A minimal weighted defect ω of best graph coloring

```
ω ← ∞    β ← 1 – α
for run ← 1 to RunCount do
    Order ← RandomBlockOrdering(M²)
    for c ← 0 to CCC – 1 do
        vCnt(c) ← 0    D(c) ← 0    d(c) ← 0
    D_max ← 0
    for i ← 0 to M² – 1 do
        block ← Order(i)    d_max ← –1
        for c ← 0 to CCC – 1 do
            d(c) ← 0
            if vCnt(c) < CSC then
                for j ← 0 to i – 1 do
                    b ← Order(j)    d ← w(b, block)
                    if d > 0 and Coloring(b) = c then
                        d(c) ← d(c) + d
                d_max ← Max(d_max, d(c))
        W_max ← –1    BestC ← –1
        for c ← 0 to CCC – 1 do
            W(c) ← 0    W1 ← W2 ← –1
            if vCnt(c) < CSC then
                if D_max ≠ 0 then
                    W1 ← α × (D_max – D(c)) / D_max
                if d_max ≠ 0 then
                    W2 ← β × (d_max – d(c)) / d_max
                if W1 ≠–1 or W2 ≠–1 then
                    if W1 ≠–1 then W(c) ← W1
                    if W2 ≠–1 then W(c) ← W(c) + W2
                    if W_max < W(c) then
                        W_max ← W(c)    BestC ← c
                else
                    if BestC = –1 then BestC = c
        Coloring(block) ← BestC
        D(BestC) ← D(BestC) + d(BestC)
        D_max ← Max(D_max, D(BestC))
        d(BestC) ← 0    vCnt(BestC) ← vCnt(BestC) + 1
    if ω > D_max then
        ω ← D_max    BestColoring ← Coloring
return ω, BestColoring
```

Table 4. **Maximum-minimum weighted defect of a single color class in defective coloring for *M*=6 vs. α, β and *CSC***

α	β	CSC				
		3	4	6	9	12
0.0	1.0	43–56	22–31	6–14	2–7	0–6
0.3	0.7	42–57	22–30	6–14	2–8	0–6
1.0	0.0	47–74	25–50	11–27	5–12	2–6

With the increase of *M* from 6 to 12 the minimized weighted defect ω per cache block given by *DWCRGC* has grown from 6 to 15 conflicts. The results given by the row-major allocation of *BSPA* are much worse: from 30 to 132 conflicts respectively. The gain of *DWCRGC* has increased from 5.0 to 8.8 times.

Table 5. **The number of conflicts given by *DWCRGC* against *BSPA* (row-major block matrix layout) vs. *M***

M	6	7	8	9	10	11	12
DWCRGC, conflict	6	8	9	11	12	14	15
BSPA, conflict	30	42	56	72	90	110	132
Gain, times	5.0	5.3	6.2	6.6	7.5	7.9	8.8

Conclusion

The paper has formulated the problem of optimizing the data allocation in main and cache memory to reduce the data miss count during execution of blocked all-pair shortest paths algorithms. We have introduced the model of block conflict weighted graph for solving the problem. The known coloring techniques does not solve the problem efficiently since they generate color classes of different size and give big fragmentation of the main memory. The paper has introduced two types of block allocation: non-conflict and weighted defective. We have pro-posed the color-class-size constrained coloring algorithms for the non-conflict allocation. Experimental results have shown the gain our random coloring algorithm provides against the deterministic one. To minimize the conflict count at the restricted cache size, we have extended the known concept of defective coloring to the concept of weighted defective coloring of the block conflict graph. Our random weighted constrained defective coloring algorithm minimizes the number of conflicts and balances the load on the cache slots for the given cache size. The model and algorithms target first the direct mapped cache although they are also applicable being modified to the set associative cache.

REFERENCES

1. **R.W. Floyd** "Algorithm 97: Shortest path", Communications of the ACM, 1962, 5(6), p.345.
2. **Hofner, P.** Dijkstra, Floyd and Warshall Meet Kleene / P. Hofner and B. Moller // Formal Aspect of Computing, Vol. 24, No. 4, 2012, № 2, pp. 459–476.
3. **G. Venkataraman, S. Sahni, S. Mukhopadhyaya** "A Blocked All-Pairs Shortest Paths Algorithm", Journal of Experimental Algorithmics (JEA), Vol. 8, 2003, pp. 857–874
4. **Prihozhy A.A., Karasik O.N.** "Heterogeneous blocked all-pairs shortest paths algorithm". «System analysis and applied information science». 2017; (3): 68–75. (In Russ.) https://doi.org/10.21122/2309–4923–2017–3–68–75.
5. **C. Kozyrakis.** "Computer Systems Architecture. Advanced Caching Techniques", Stanford University, pp. 1–35, 2012.
6. **Smith, A.J.,** "Cache Memories", Computing Surveys. 1982, 14 (3): 473–530.
7. **J.S. Park, M. Penner, and V.K. Prasanna.** "Optimizing graph algorithms for improved cache performance" / J.S. Park, // IEEE Trans. on Parallel and Distributed Systems, 2004, 15(9), pp. 769–782.
8. **Prihozhy A.A.** Simulation of direct mapped, k-way and fully associative cache on all pairs shortest paths algorithms. «System analysis and applied information science». 2019; (4):10–18.
9. **Solomonik, E.** Minimizing Communication in All Pairs Shortest Paths / E. Solomonik, A. Buluc, and J. Demmel // IEEE 27th International Symposium on Parallel & Distributed Processing, 2013, pp. 548–559.
10. **Tang, P.** Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-Core Computers / P. Tang // IEEE SOUTHEASTCON 2014, pp. 1–7.
11. **Prihozhy, A.A.** Adaptive memory management. Automation and computer technology, 1988, № 3, c. 58–65.
12. **Prihozhy, A.A.** Asynchronous scheduling and allocation / A.A. Prihozhy / Proceedings Design, Automation and Test in Europe. Paris, France.– IEEE, 1998, pp. 963–964.
13. **Prihozhy A.A., Karasik O.N.** Investigation of methods for implementing multithreaded applications on multicore systems. Informatization of education, 2014, № 1, c. 43–62.
14. **Prihozhy A.A., Karasik O.N.** Cooperative model for optimization of execution of threads on multi-core system. «System analysis and applied information science». 2014;(4):13–20. (In Russ.)
15. **Chaitin, G. J.** "Register allocation & spilling via graph colouring", Proc. 1982 SIGPLAN Symposium on Computer Construction, 1982, pp. 98–105.
16. **Bodlaender, H.L., Fomin, F.V.** "Equitable colorings of bounded treewidth graphs", Theoretical Computer Science, 2005, 349 (1): 22–30.
17. **Hajnal, A., Szemeredi E.** "Proof of a conjecture of P. Erdős", Combinatorial theory and its applications, II (Proc. Colloq., Balatonfüred, 1969), North-Holland, 1970, pp. 601–623
18. **Cowen, L.J., Cowen, R. H., Woodall, D. R.** "Defective colorings of graphs in surfaces: Partitions into subgraphs of bounded valency". Journal of Graph Theory, 2006, 10 (2): 187–195.

ЛИТЕРАТУРА

1. **R.W. Floyd** "Algorithm 97: Shortest path", Communications of the ACM, 1962, 5(6), p. 345.
2. **Hofner, P.** Dijkstra, Floyd and Warshall Meet Kleene / P. Hofner and B. Moller // Formal Aspect of Computing, Vol. 24, No. 4, 2012, № 2, pp. 459–476.
3. **G. Venkataraman, S. Sahni, S. Mukhopadhyaya** "A Blocked All-Pairs Shortest Paths Algorithm", Journal of Experimental Algorithmics (JEA), Vol 8, 2003, pp. 857–874
4. **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.– С. 68–75.
5. **C. Kozyrakis** "Computer Systems Architecture. Advanced Caching Techniques", Stanford University, pp. 1–35, 2012.
6. **Smith, A.J.** "Cache Memories", Computing Surveys. 1982, 14 (3): 473–530.
7. **J.S. Park, M. Penner, and V.K. Prasanna** "Optimizing graph algorithms for improved cache performance" / J.S. Park, // IEEE Trans. on Parallel and Distributed Systems, 2004, 15(9), pp.769–782.
8. **Prihozhy A.A.** Simulation of direct mapped, k-way and fully associative cache on all pairs shortest paths algorithms. «System analysis and applied information science». 2019; (4):10–18.
9. **Solomonik, E.** Minimizing Communication in All Pairs Shortest Paths / E. Solomonik, A. Buluc, and J. Demmel // IEEE 27th International Symposium on Parallel & Distributed Processing, 2013, pp. 548–559.
10. **Tang, P.** Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-Core Computers / P. Tang // IEEE SOUTHEASTCON 2014, pp. 1–7.
11. **Прихожий, А.А.** Адаптивное управление памятью / А.А. Прихожий // Автоматика и вычислительная техника, 1988, № 3, с. 58–65
12. **Prihozhy, A.A.** Asynchronous scheduling and allocation / A.A. Prihozhy / Proceedings Design, Automation and Test in Europe. Paris, France.– IEEE, 1998, pp. 963–964.
13. **Прихожий, А.А.** Исследование методов реализации многопоточных приложений на многоядерных системах / А.А. Прихожий, О.Н. Карасик // Информатизация образования, 2014, № 1, с. 43–62.
14. **Прихожий, А.А.** Кооперативная модель оптимизации выполнения потоков на многоядерной системе / А.А. Прихожий, О.Н. Карасик // Системный анализ и прикладная информатика, 2014, № 4, с. 13–20.
15. **Chaitin, G. J.** "Register allocation & spilling via graph colouring", Proc. 1982 SIGPLAN Symposium on Computer Construction, 1982, pp. 98–105.
16. **Bodlaender, H.L., Fomin, F.V.** "Equitable colorings of bounded treewidth graphs", Theoretical Computer Science, 2005, 349 (1): 22–30.
17. **Hajnal, A., Szemeredi E.** "Proof of a conjecture of P. Erdős", Combinatorial theory and its applications, II (Proc. Colloq., Balatonfüred, 1969), North-Holland, 1970, pp. 601–623
18. **Cowen, L.J., Cowen, R.H., Woodall, D.R.** "Defective colorings of graphs in surfaces: Partitions into subgraphs of bounded valency". Journal of Graph Theory, 2006, 10 (2): 187–195.

Поступила	*После доработки*	*Принята к печати*
11.08.2021	*01.09.2021*	*01.09.2021*

ПРИХОЖИЙ А. А.

ОПТИМИЗАЦИЯ РАЗМЕЩЕНИЯ ДАННЫХ В ИЕРАРХИЧЕСКОЙ ПАМЯТИ ДЛЯ БЛОЧНЫХ АЛГОРИТМОВ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ

Статья посвящена сокращению обмена данными между основной памятью и кэш прямого сопоставления при выполнении блочных алгоритмов поиска кратчайших путей, представляющих данные матрицей блоков D[M×M]. Для больших графов размер кэш $S = \delta \times M^2$, $\delta < 1$ меньше размера матрицы. Кэш назначает группу блоков основной памяти на один блок кэш. Алгоритмы пересчитывают блок матрицы через один или два других блока и могут обращаться сразу к трем блокам. Если эти блоки назначены на один блок кэш, между ними возникает конфликт, приводящий к активному обмену данными между уровнями памяти. Распределение блоков по группам и число конфликтов сильно зависят от размещения и упорядочения блоков матрицы в основной памяти. В статье предлагается решать проблему оптимального размещения на взвешенном графе конфликтов блоков и различать два случая назначения блоков на кэш: безконфликтного и минимально-конфликтного. В первом случае формулируется проблема равномерной раскраски графа конфликтов, предлагаются детерминированный и случайный алгоритмы ее решения. Во втором случае формулируется проблема взвешенной дефектной раскраски графа при ограничении на число цветов, предлагается случайный алгоритм ее решения. Экспериментальные результаты показывают, что случайный алгоритм равномерной раскраски дает верхнюю границу размера кэш очень близкую к нижней границе, оцениваемой через полный подграф, и показывает, что бесконфликтное размещение матрицы возможно при $\delta = 0.5$ для $M = 4$ и при $\delta = 0.1$ для $M = 20$. Для малого размера кэш взвешенный дефектный алгоритм дает число оставшихся конфликтов до 8.8 раз меньшее чем начальное размещение. Предложенные модель и алгоритмы применимы также к k-канальному ассоциативному кэш.

Ключевые слова: *алгоритм поиска кратчайших путей, иерархическая память, кэш прямого отображения, производительность, размещение данных, граф конфликтов блоков, равномерная раскраска, дефектная раскраска.*

Anatoly Prihozhy is a full professor at the Computer and system software department of Belarus national technical university, doctor of science (1999) and full professor (2001). His research interests include programming and hardware description languages, parallelizing compilers, and computer aided design techniques and tools for software and hardware at logic, high and system levels, and for incompletely specified logical systems. He has over 300 publications in Eastern and Western Europe, USA and Canada. Such worldwide publishers as IEEE, Springer, Kluwer Academic Publishers, World Scientific and others have published his works.