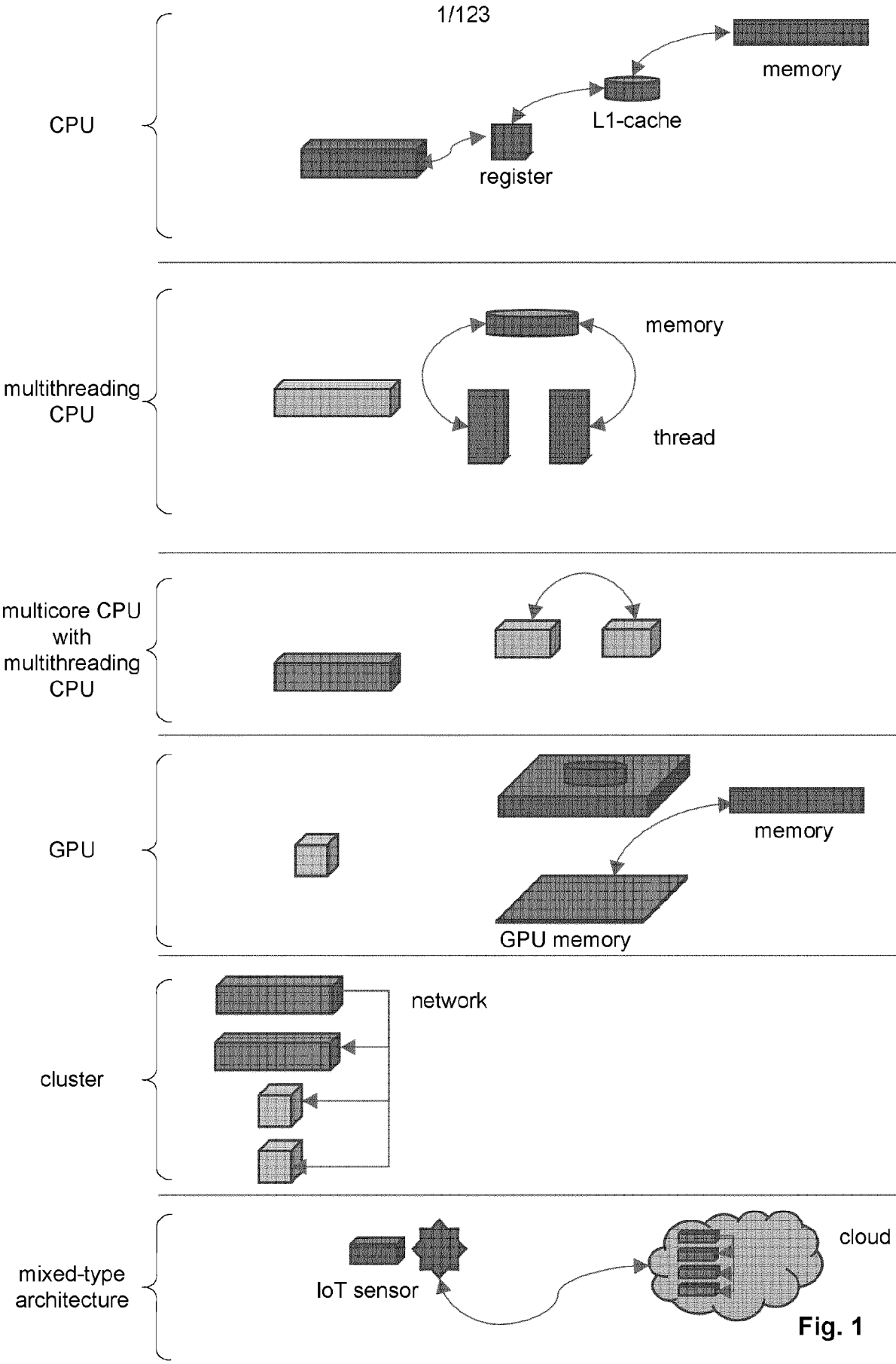

CPU
memory
L1-cache
register
multithreading CPU
memory
thread
multicore CPU with multithreading CPU
GPU
memory
GPU memory
cluster
network
mixed-type architecture
IoT sensor
cloud

Fig. 1

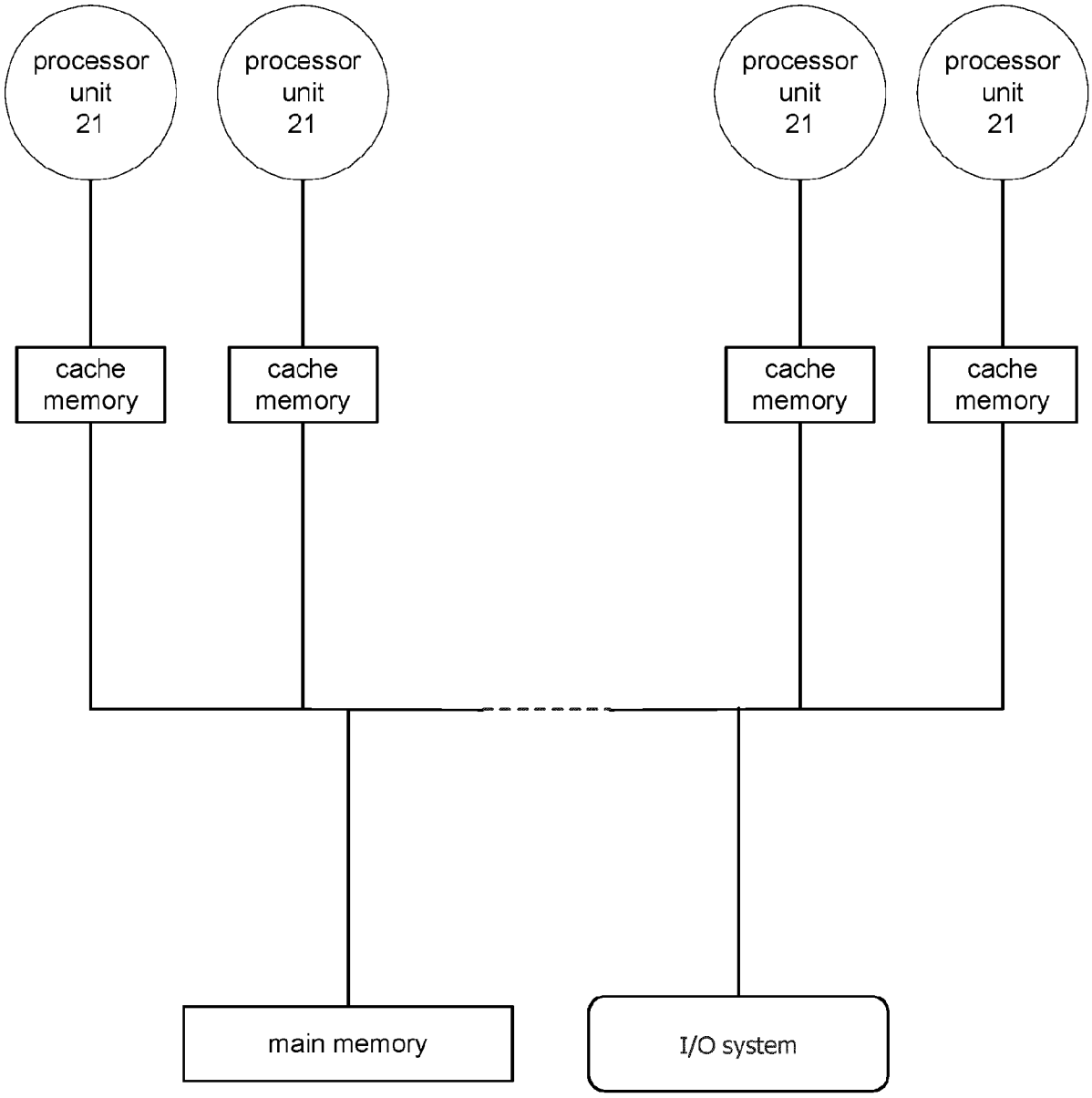

processor unit 21
processor unit 21
processor unit 21
processor unit 21
cache memory
cache memory
cache memory
cache memory
main memory
I/O system

Fig. 2

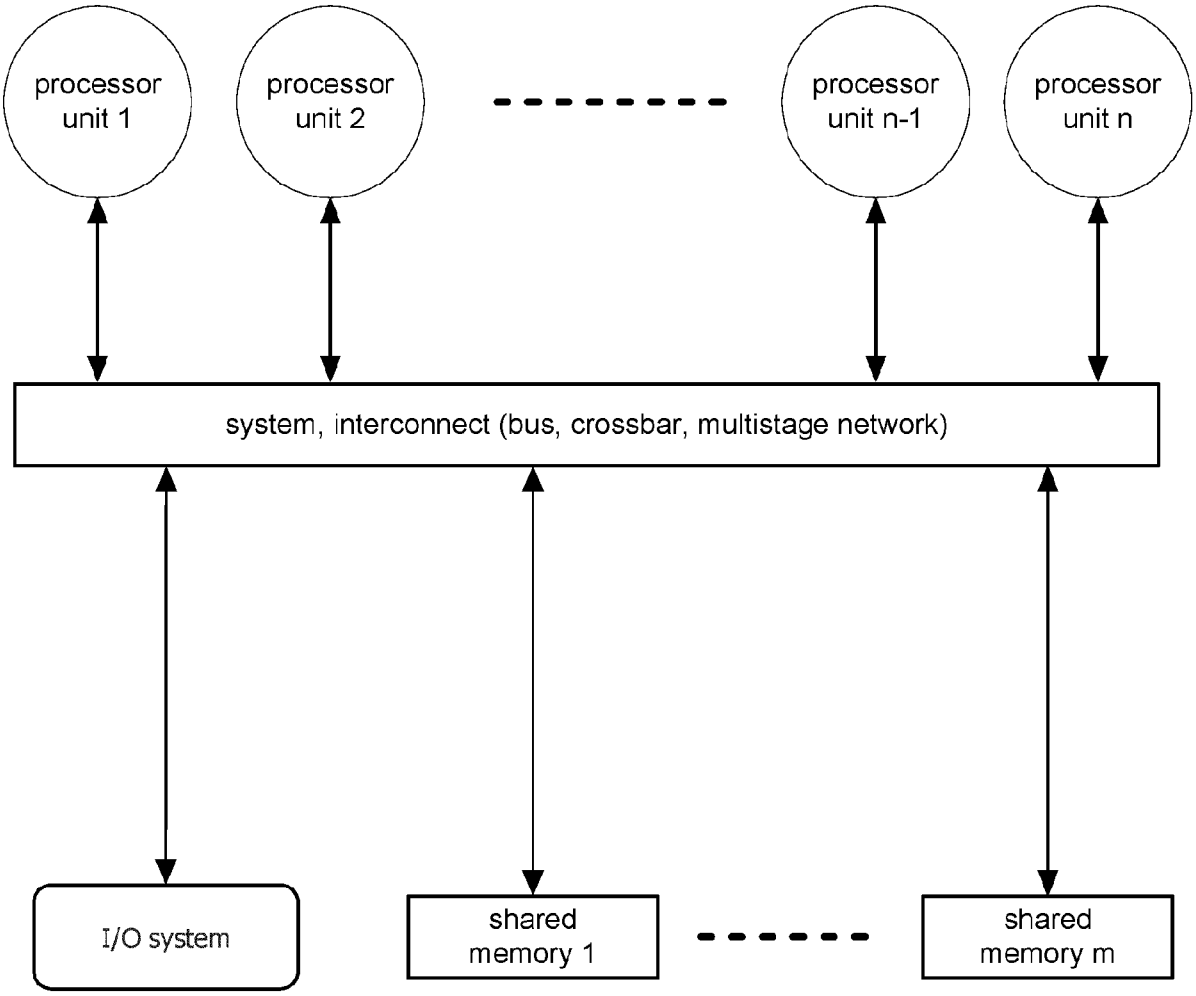

processor unit 1
processor unit 2
processor unit n-1
processor unit n
system, interconnect (bus, crossbar, multistage network)
I/O system
shared memory 1
shared memory m

Fig. 3

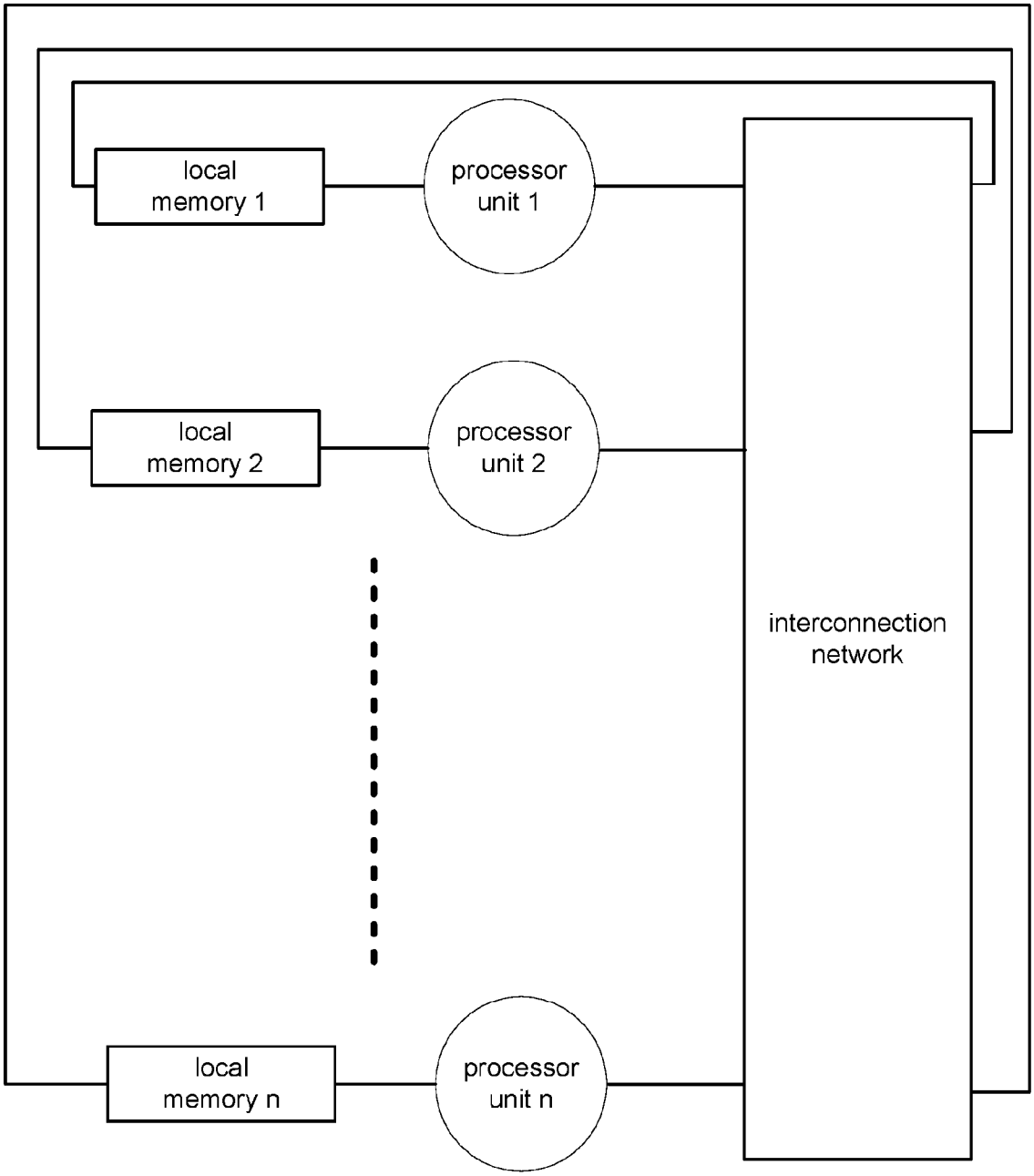

local
memory 1
processor
unit 1
local
memory 2
processor
unit 2
interconnection
network
local
memory n
processor
unit n

Fig. 4

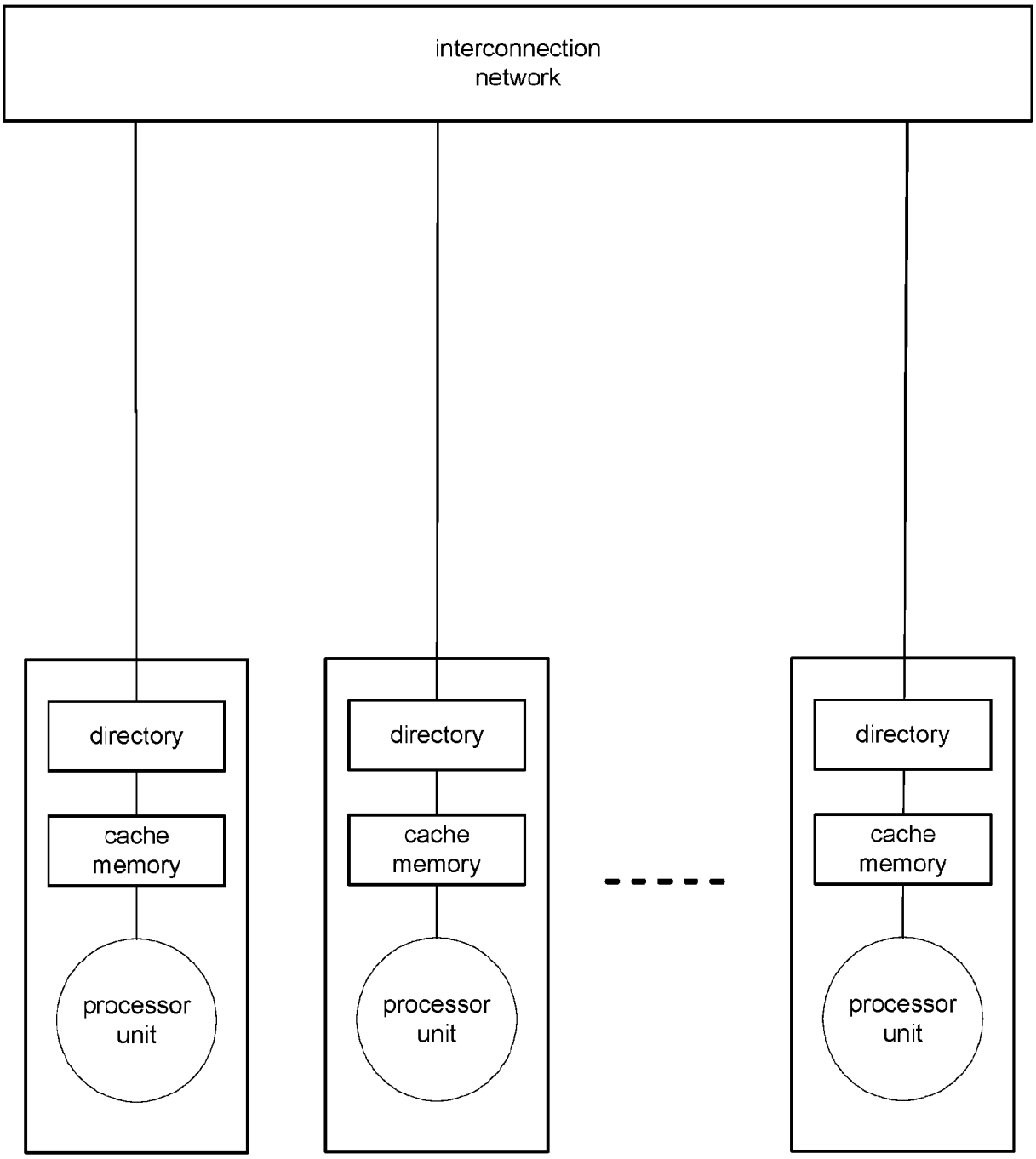

interconnection network
directory
cache memory
processor unit
directory
cache memory
processor unit
directory
cache memory
processor unit

Fig. 5

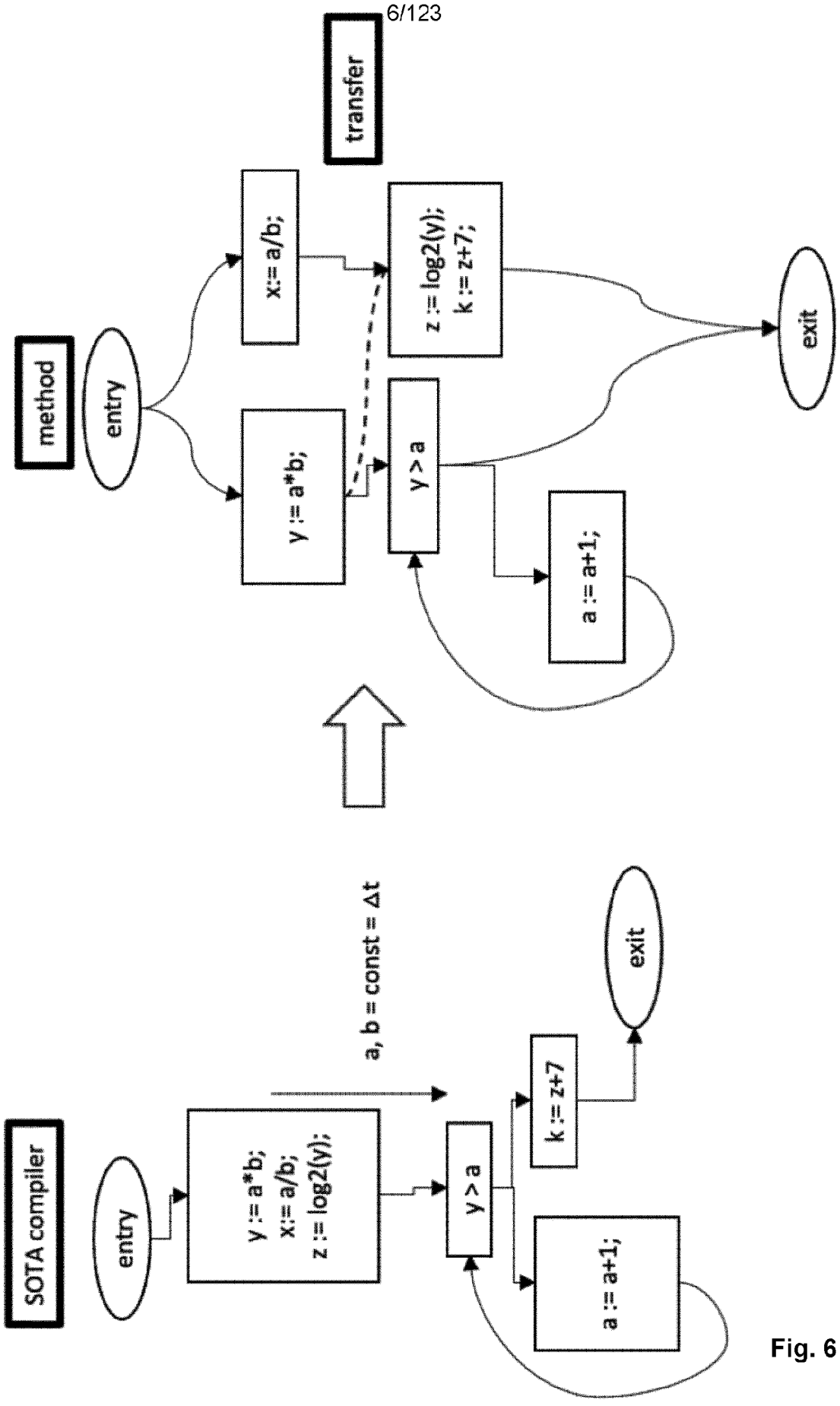
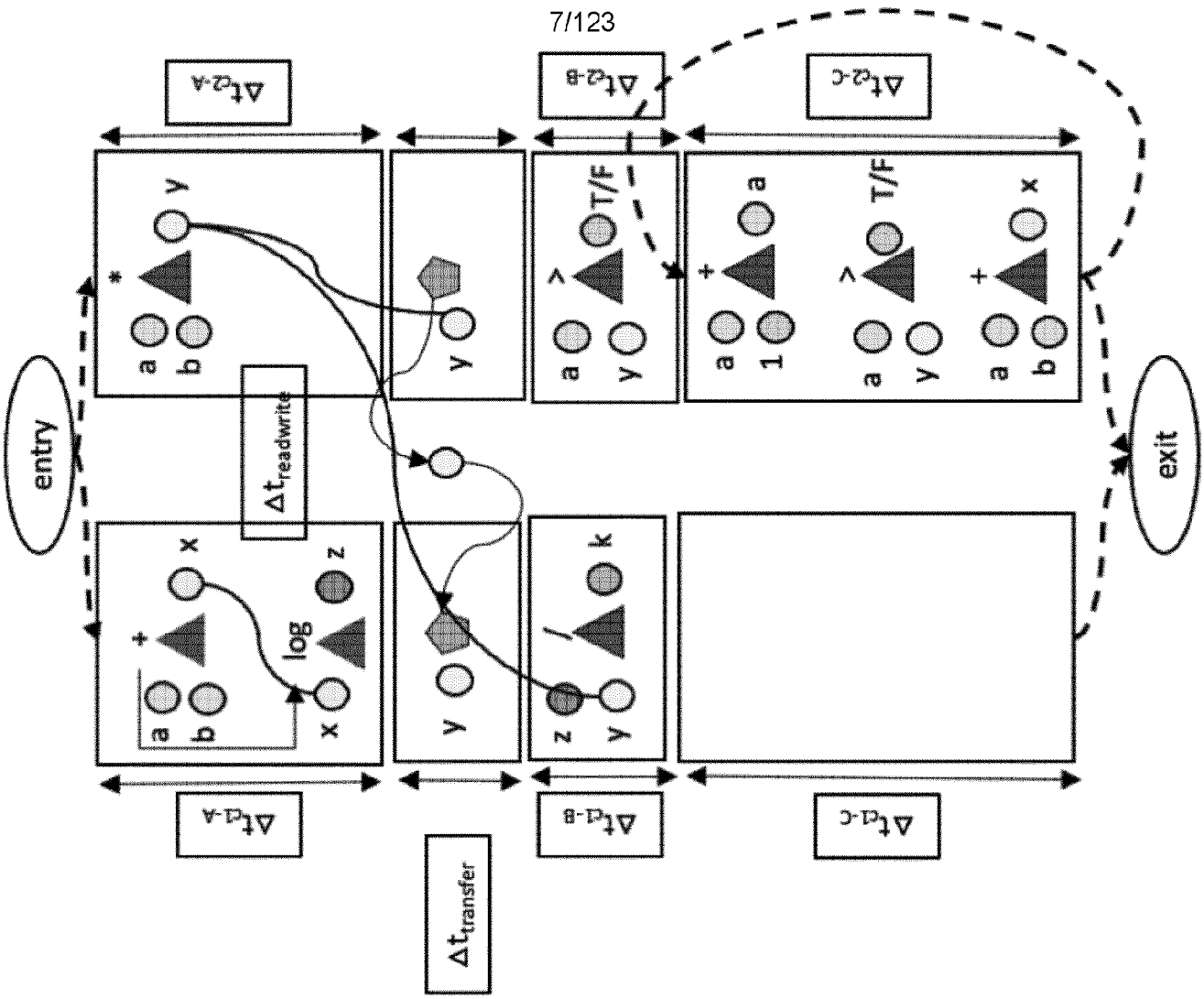

transfer
method
entry
x:= a/b;
y := a*b;
z := log2(y);
k := z+7;
y > a
a := a+1;
exit
a, b = const = Δt
SOTA compiler
entry
y := a*b;
x:= a/b;
z := log2(y);
y > a
k := z+7
a := a+1;
exit

Fig. 6


entry
exit
$\Delta t_{readwrite}$
$\Delta t_{transfer}$
Δt_{a-A}
Δt_{a-B}
Δt_{a-C}
Δt_{d-A}
Δt_{d-B}
Δt_{d-C}
T/F
log

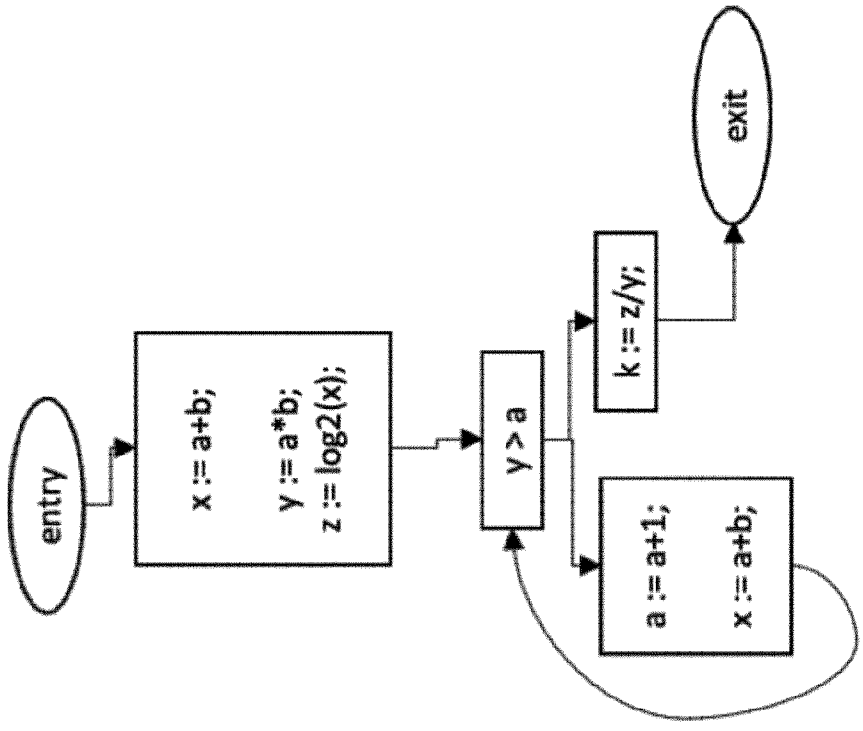

entry
x := a+b;
y := a*b;
z := log2(x);
y > a
k := z/y;
exit
a := a+1;
x := a+b;

Fig. 7

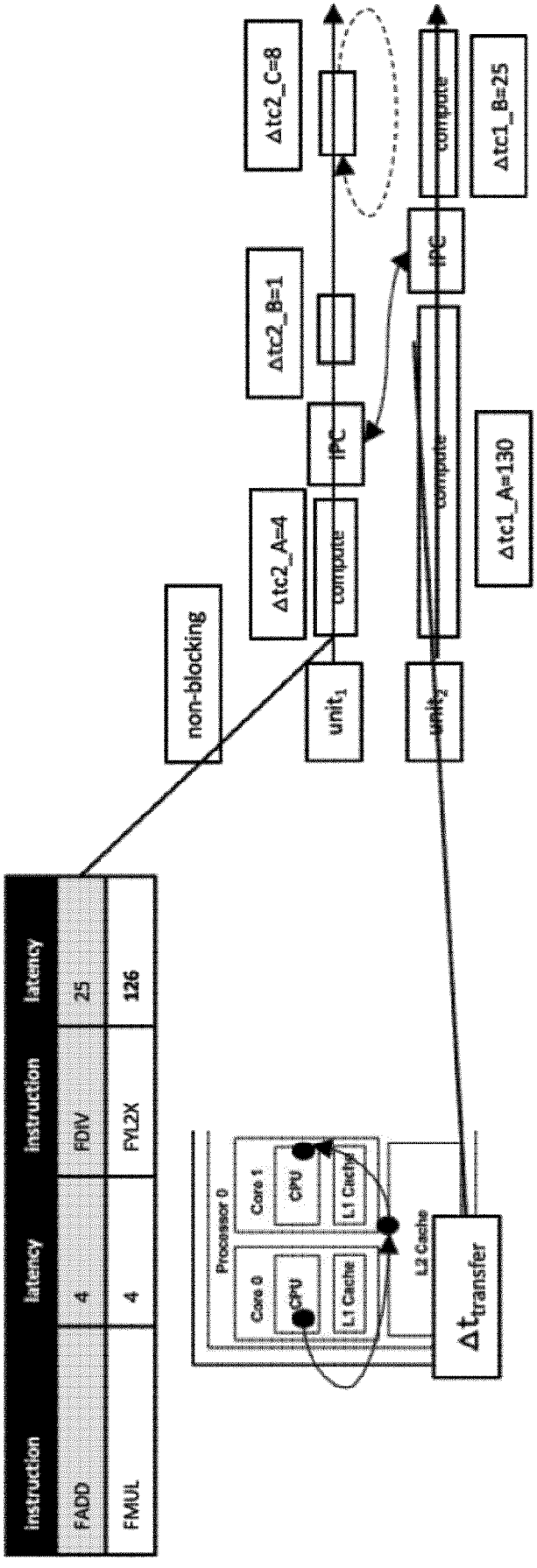

instruction
latency
FADD
4
FMUL
4
instruction
latency
FDIV
25
FYL2X
126
non-blocking
unit1
Δtc2_A=4
compute
IPC
Δtc2_B=1
Δtc2_C=8
unit2
compute
IPC
Δtc1_A=130
Δtc1_B=25
Processor 0
Core 0
CPU
L1 Cache
Core 1
CPU
L1 Cache
L2 Cache
Δttransfer

Fig. 8

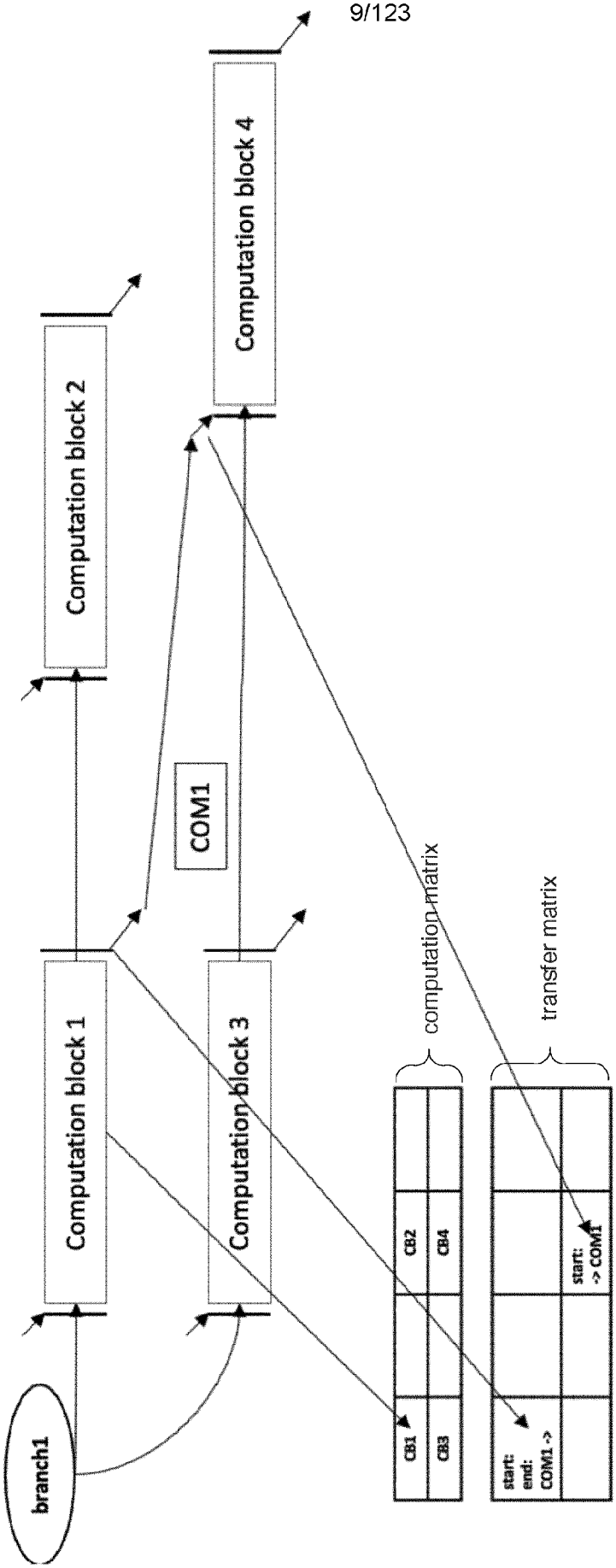

Computation block 4
Computation block 2
COM1
computation matrix
transfer matrix
Computation block 1
Computation block 3
CB2
CB4
start:
-> COM1
CB1
CB3
start:
end:
COM1 ->
branch1

Fig. 9

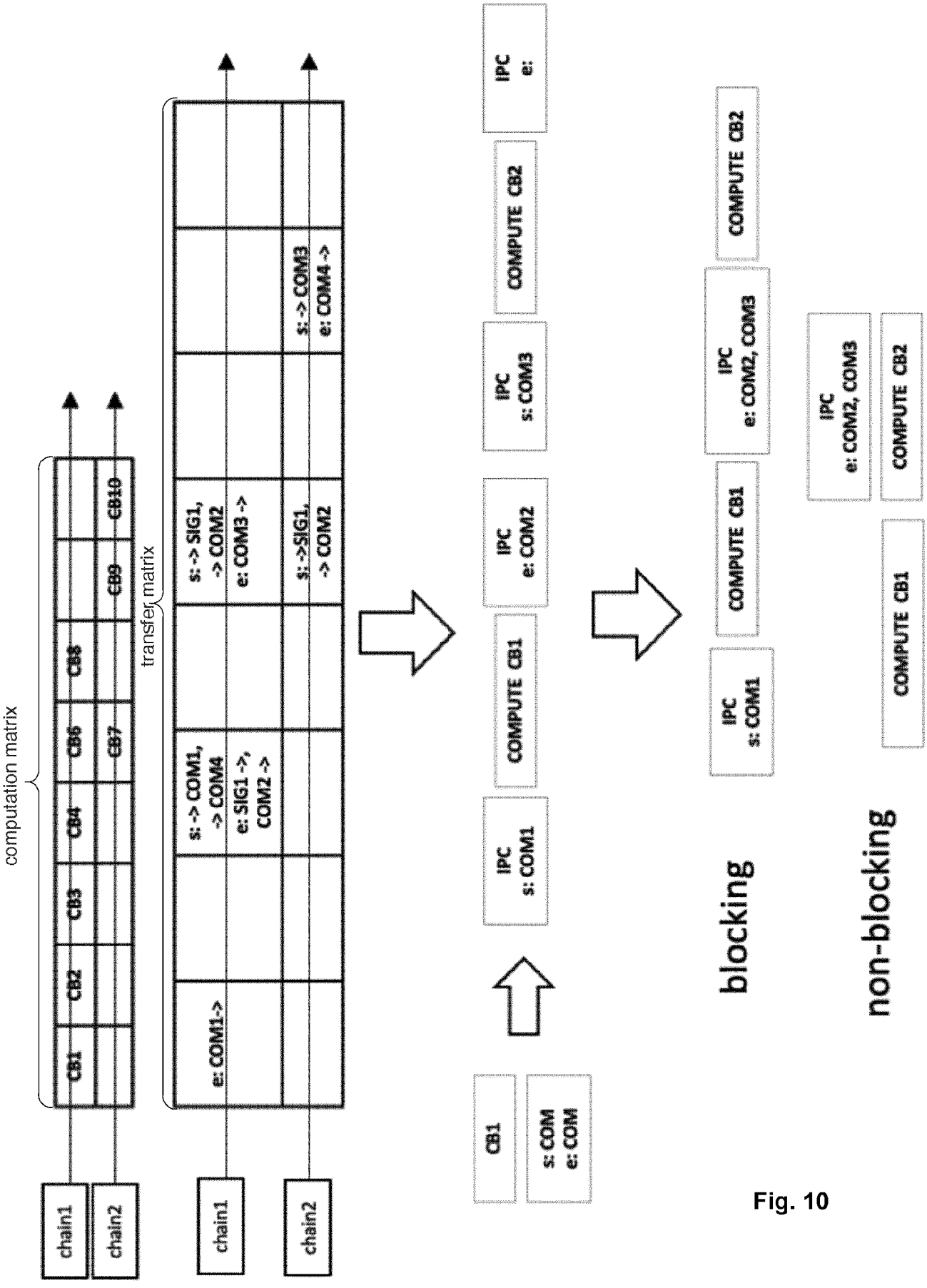
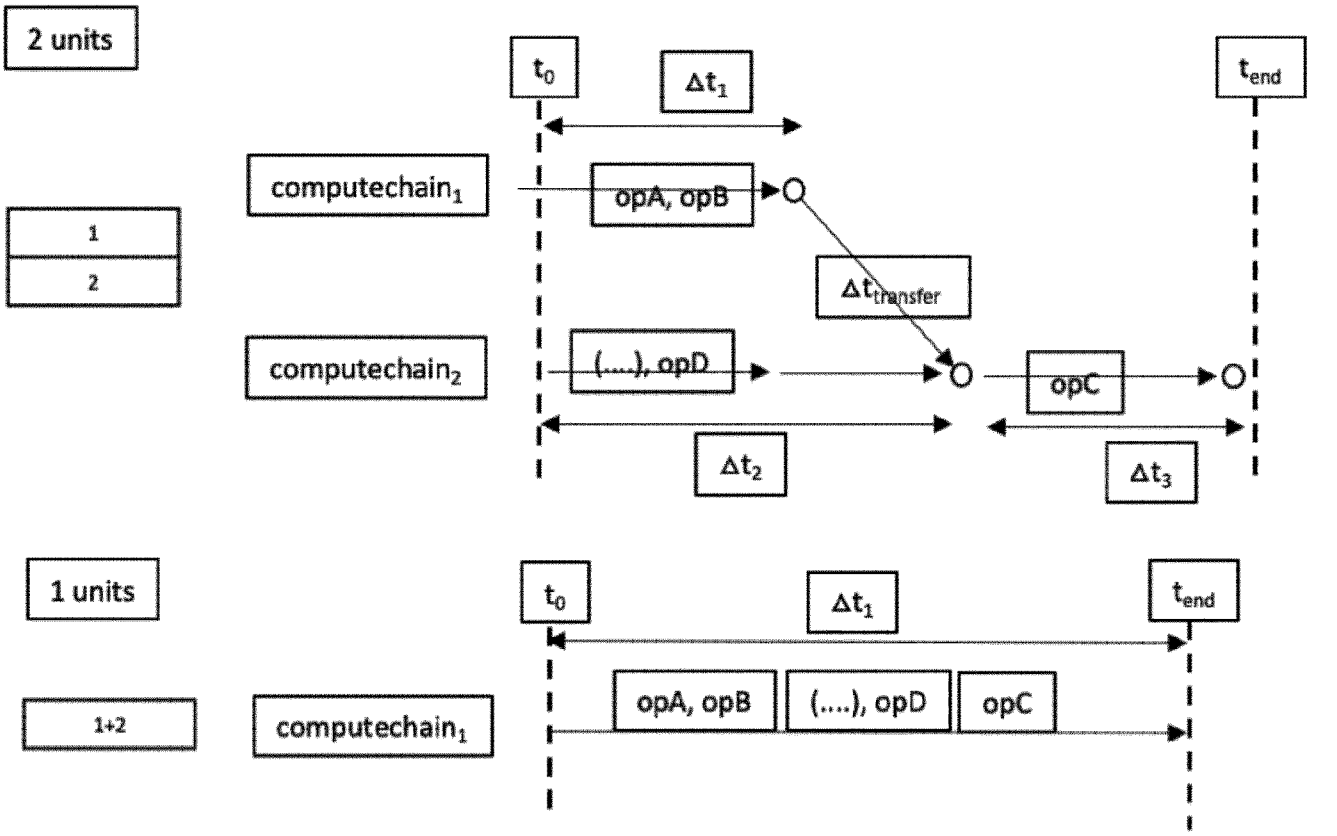

computation matrix
transfer matrix
CB1
CB2
CB3
CB4
CB6
CB8
CB7
CB9
CB10
chain1
chain2
e: COM1->
s: -> COM1, -> COM4
e: SIG1 ->, COM2 ->
s: -> SIG1, -> COM2
e: COM3 ->
s: ->SIG1, -> COM2
s: -> COM3
e: COM4 ->
chain1
chain2
CB1
s: COM
e: COM
IPC
s: COM1
COMPUTE CB1
IPC
e: COM2
IPC
s: COM3
COMPUTE CB2
IPC
e:
blocking
IPC
s: COM1
COMPUTE CB1
IPC
e: COM2, COM3
COMPUTE CB2
non-blocking
IPC
e: COM2, COM3
COMPUTE CB2
COMPUTE CB1

Fig. 10



computation Δt

communication $\Delta t_{transfer}$

$$t_{end1} = \max(\Delta t_1 + \Delta t_{transfer}, \Delta t_2) + \Delta t_3$$

$$t_{end2} = \Delta t_1$$

minimal runtime t: min(t_{end1}, t_{end2})

Fig. 11

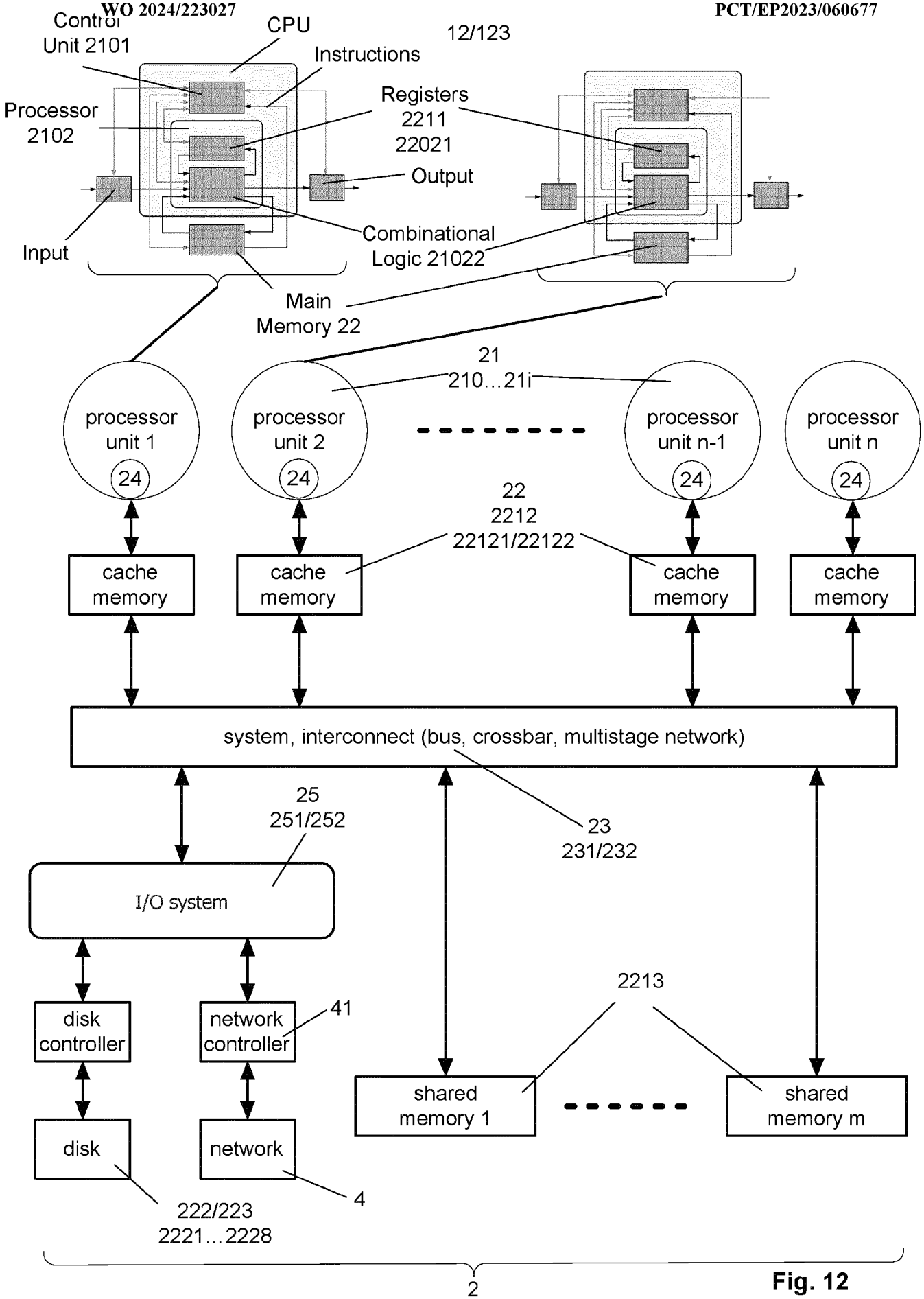

Control Unit 2101
CPU
12/123
Instructions
Processor 2102
Registers 2211 22021
Output
Input
Combinational Logic 21022
Main Memory 22
21
210...21i
processor unit 1
processor unit 2
processor unit n-1
processor unit n
24
22
2212
22121/22122
cache memory
system, interconnect (bus, crossbar, multistage network)
25
251/252
23
231/232
I/O system
2213
disk controller
network controller
41
shared memory 1
shared memory m
disk
network
4
222/223
2221...2228
2

Fig. 12

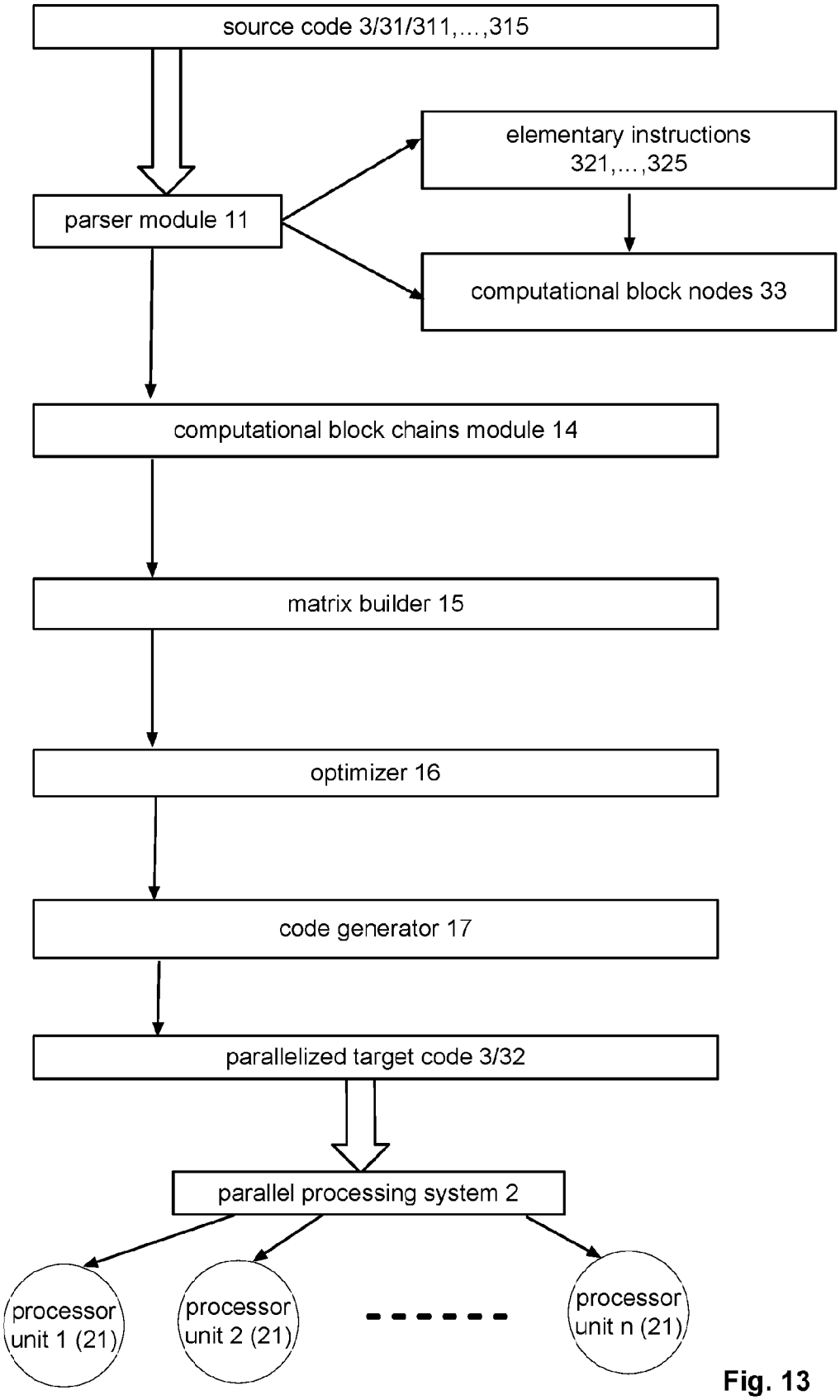

source code 3/31/311,...,315
elementary instructions 321,...,325
parser module 11
computational block nodes 33
computational block chains module 14
matrix builder 15
optimizer 16
code generator 17
parallelized target code 3/32
parallel processing system 2
processor unit 1 (21)
processor unit 2 (21)
processor unit n (21)

Fig. 13

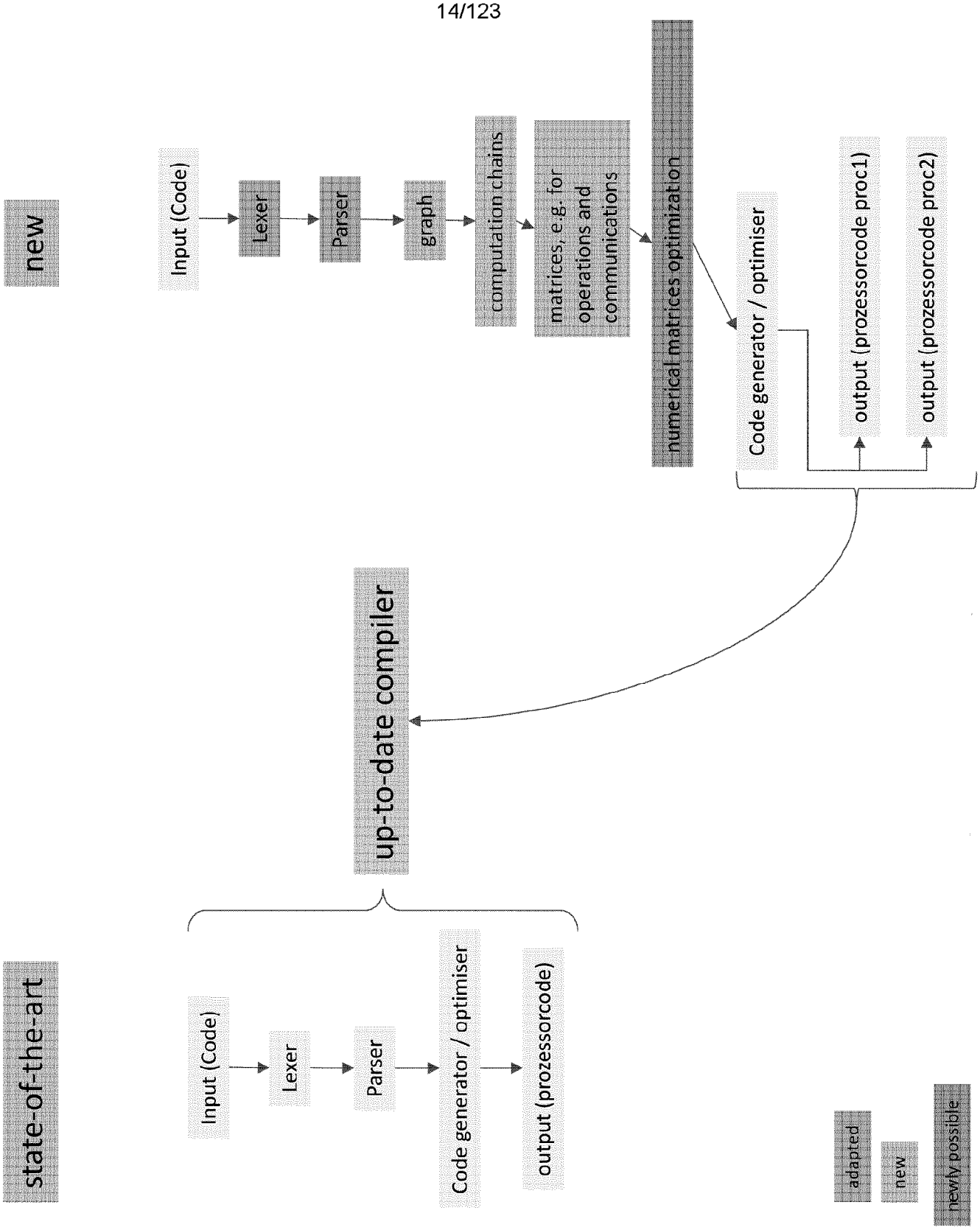

new
Input (Code)
Lexer
Parser
graph
computation chains
matrices, e.g. for operations and communications
numerical matrices optimization
Code generator / optimiser
output (prozessorcode proc1)
output (prozessorcode proc2)
up-to-date compiler
state-of-the-art
Input (Code)
Lexer
Parser
Code generator / optimiser
output (prozessorcode)
adapted
new
newly possible

Fig. 14

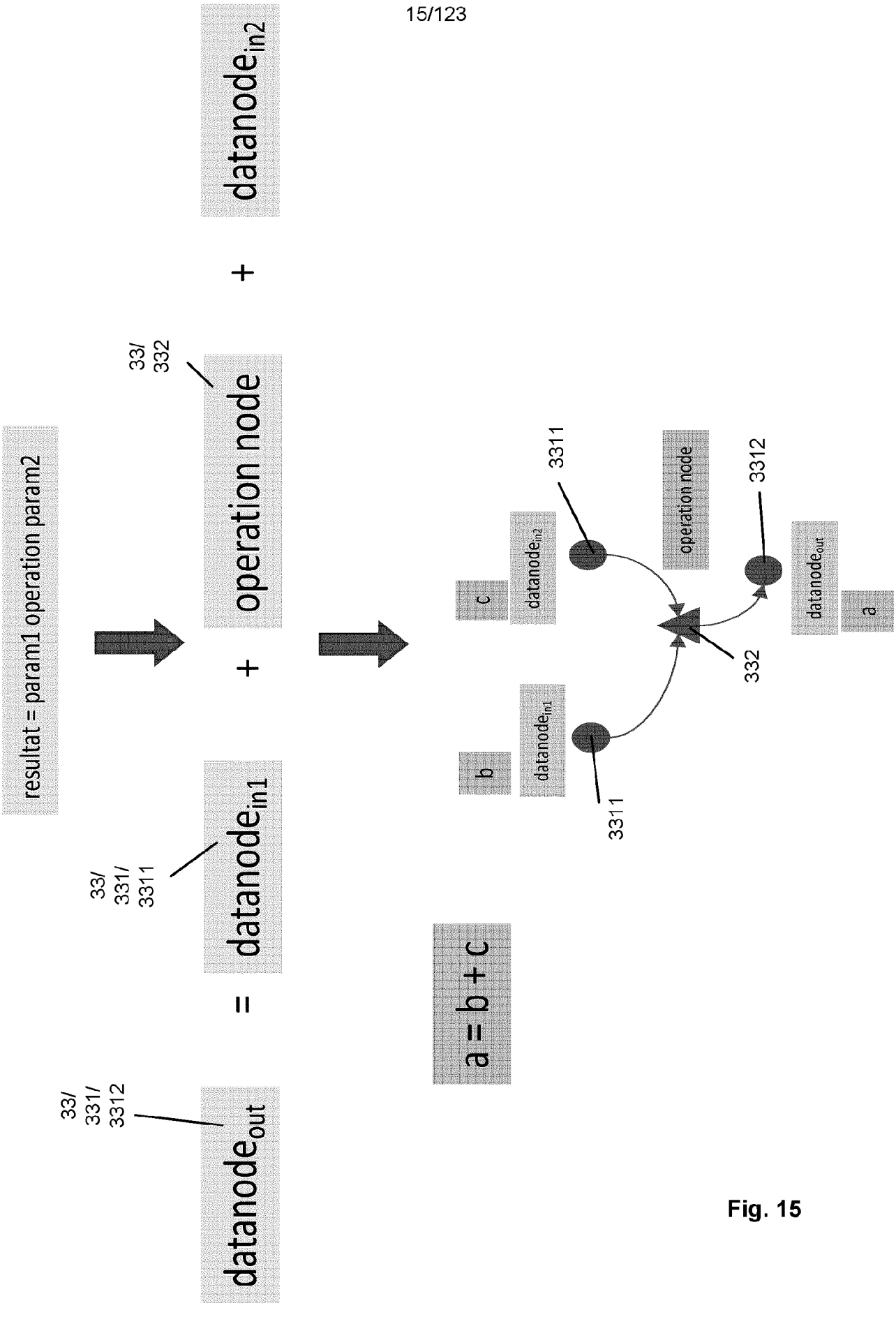
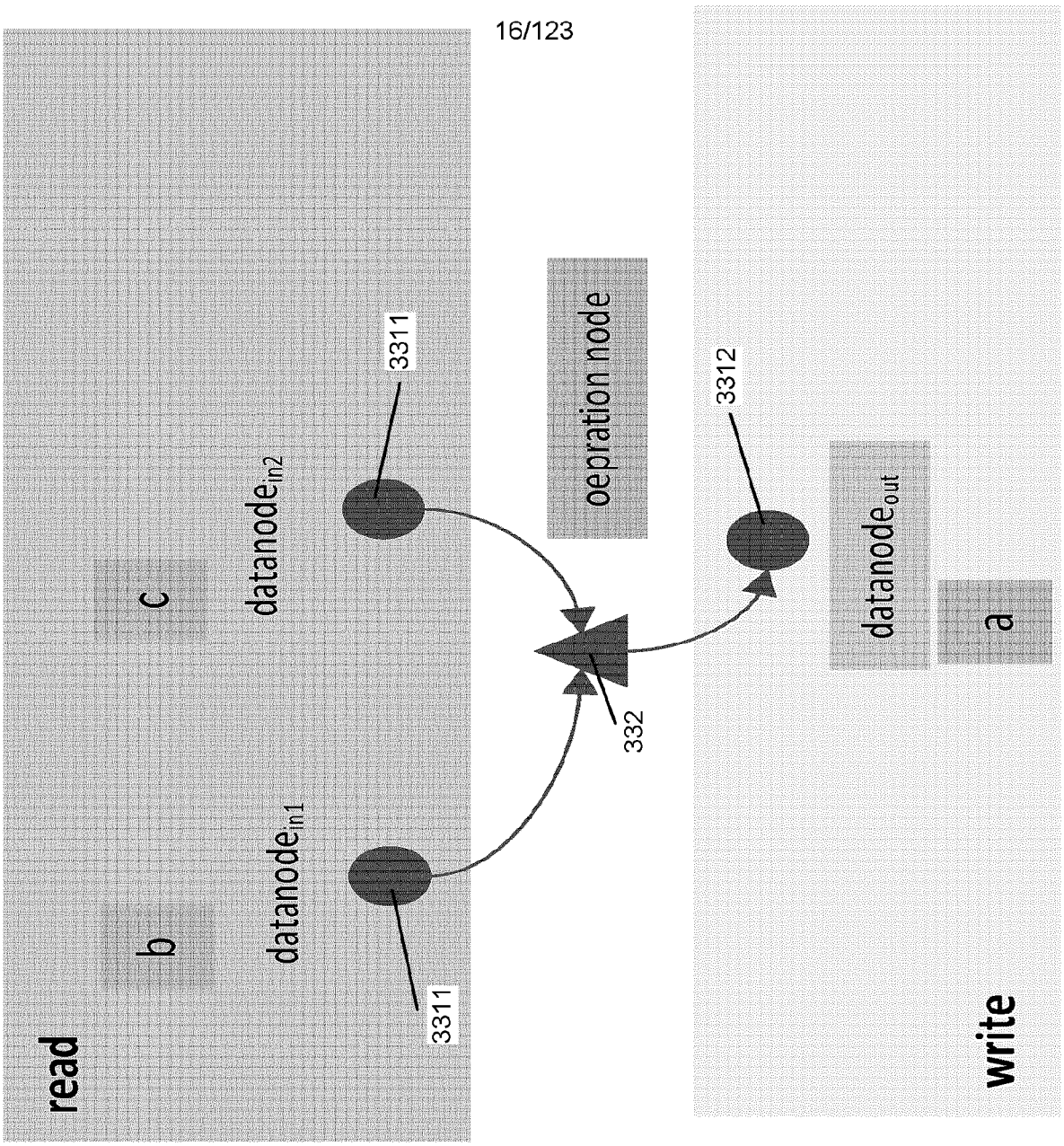
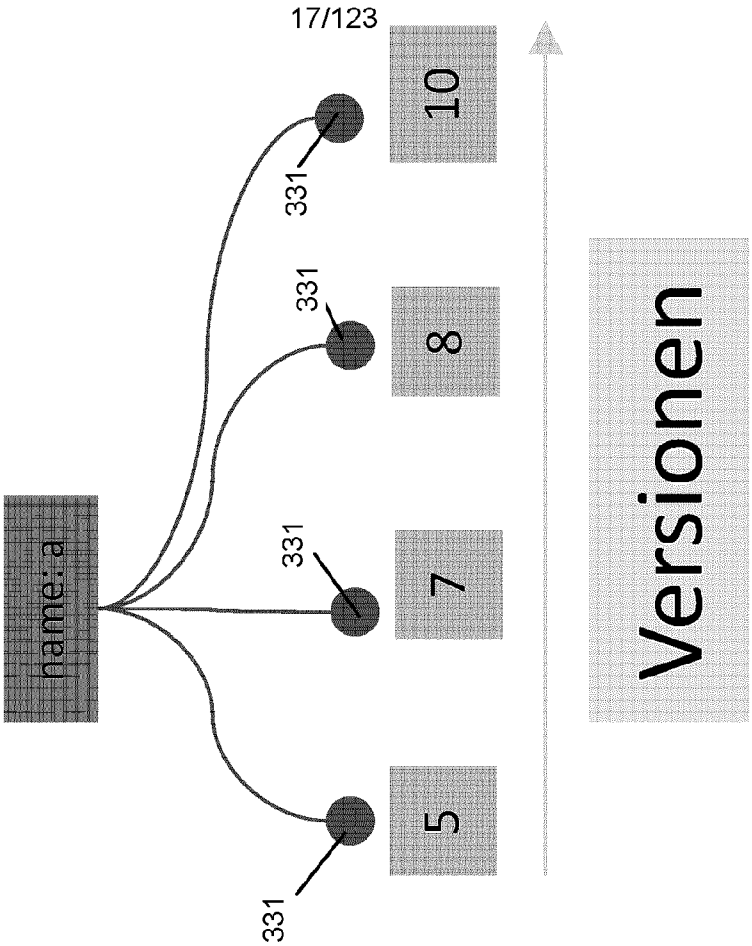

resultat = param1 operation param2
33/ 332
+ operation node
+ datanode$_{in2}$
33/ 331/ 3311
= datanode$_{in1}$
33/ 331/ 3312
datanode$_{out}$
a = b + c
c
datanode$_{in2}$
3311
operation node
3312
b
datanode$_{in1}$
3311
332
datanode$_{out}$
a

Fig. 15


read
c
b
datanode$_{in2}$
datanode$_{in1}$
3311
3311
oepration node
332
3312
datanode$_{out}$
a
write

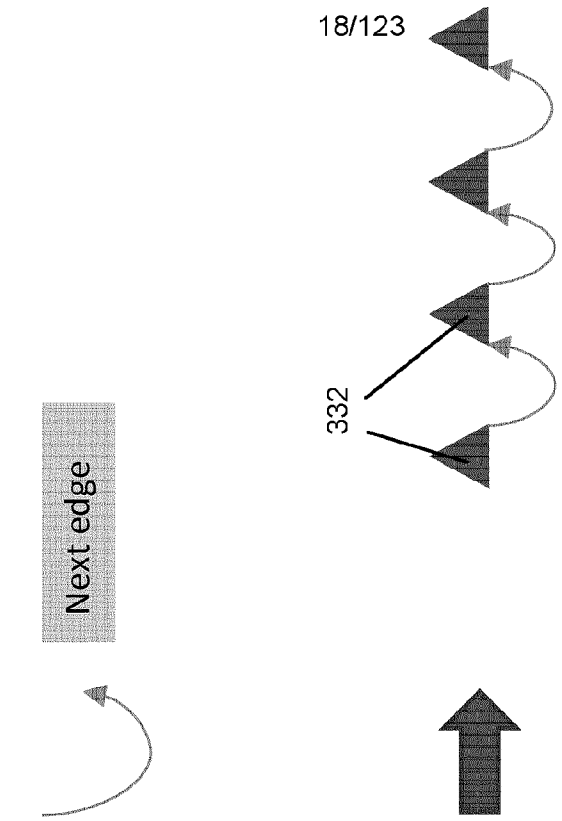
a=b+c

Fig. 16



a = 5
a = 7
a = 8
a = 10

Fig. 17



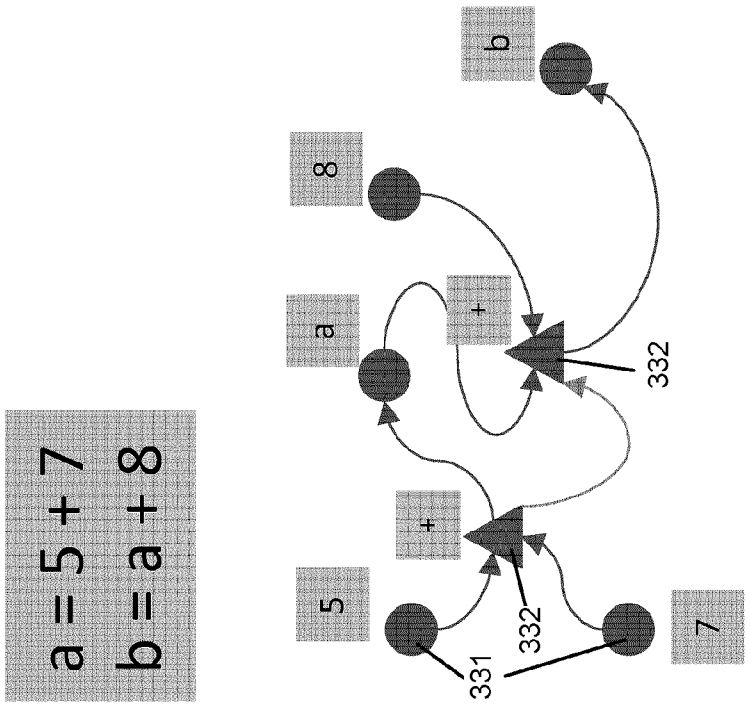


Fig. 18

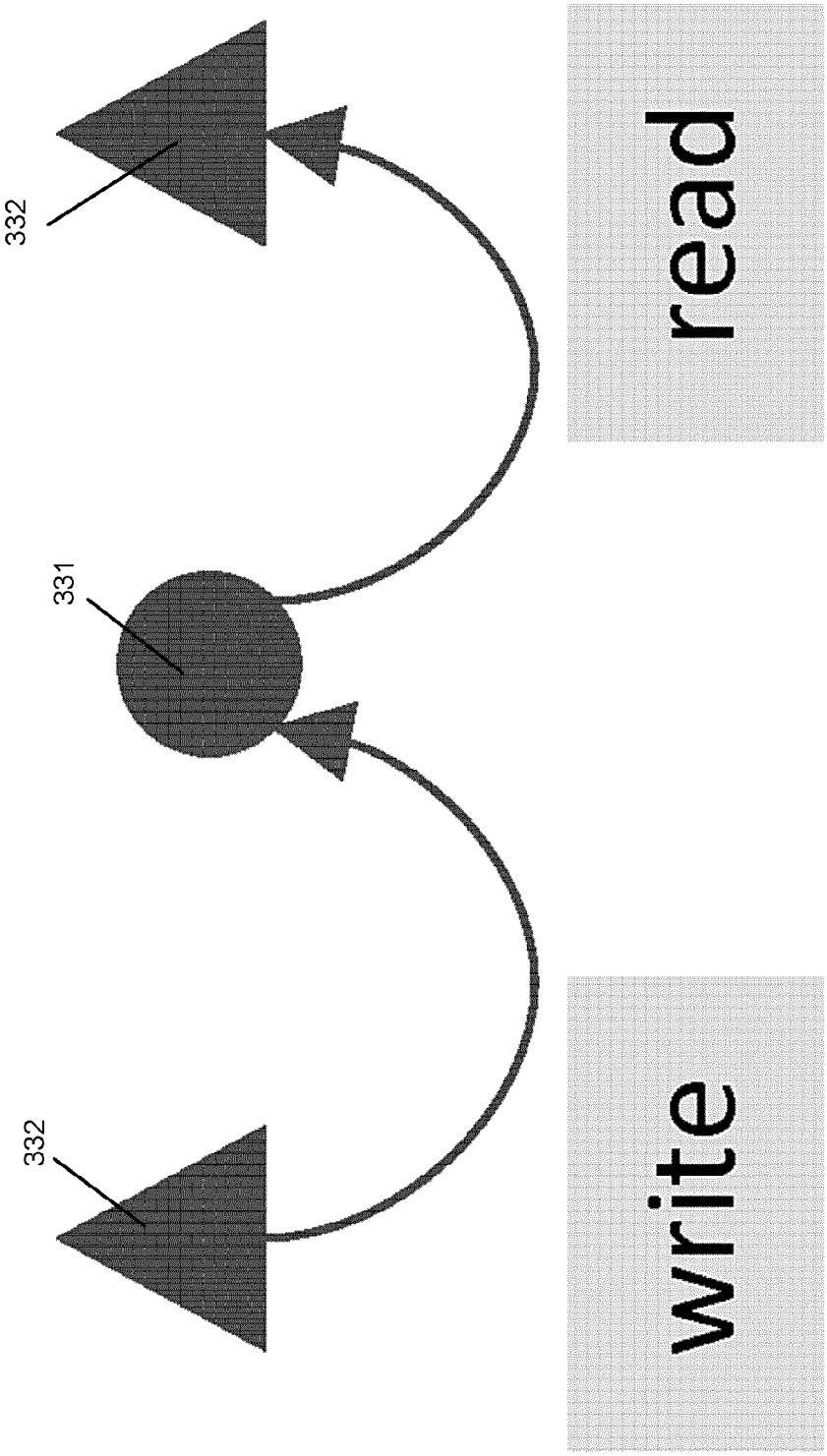


Fig. 19

$\Delta t_{readwrite}$ as a indicator what can be done with this data node

- $\Delta t_{readwrite}$ sehr klein: same cache_line or nearby
- $\Delta t_{readwrite}$ „mittel gross“: on same #cpu unit, use prefetch
- $\Delta t_{readwrite}$ „genug gross“: possible to transfer over network

Fig. 20

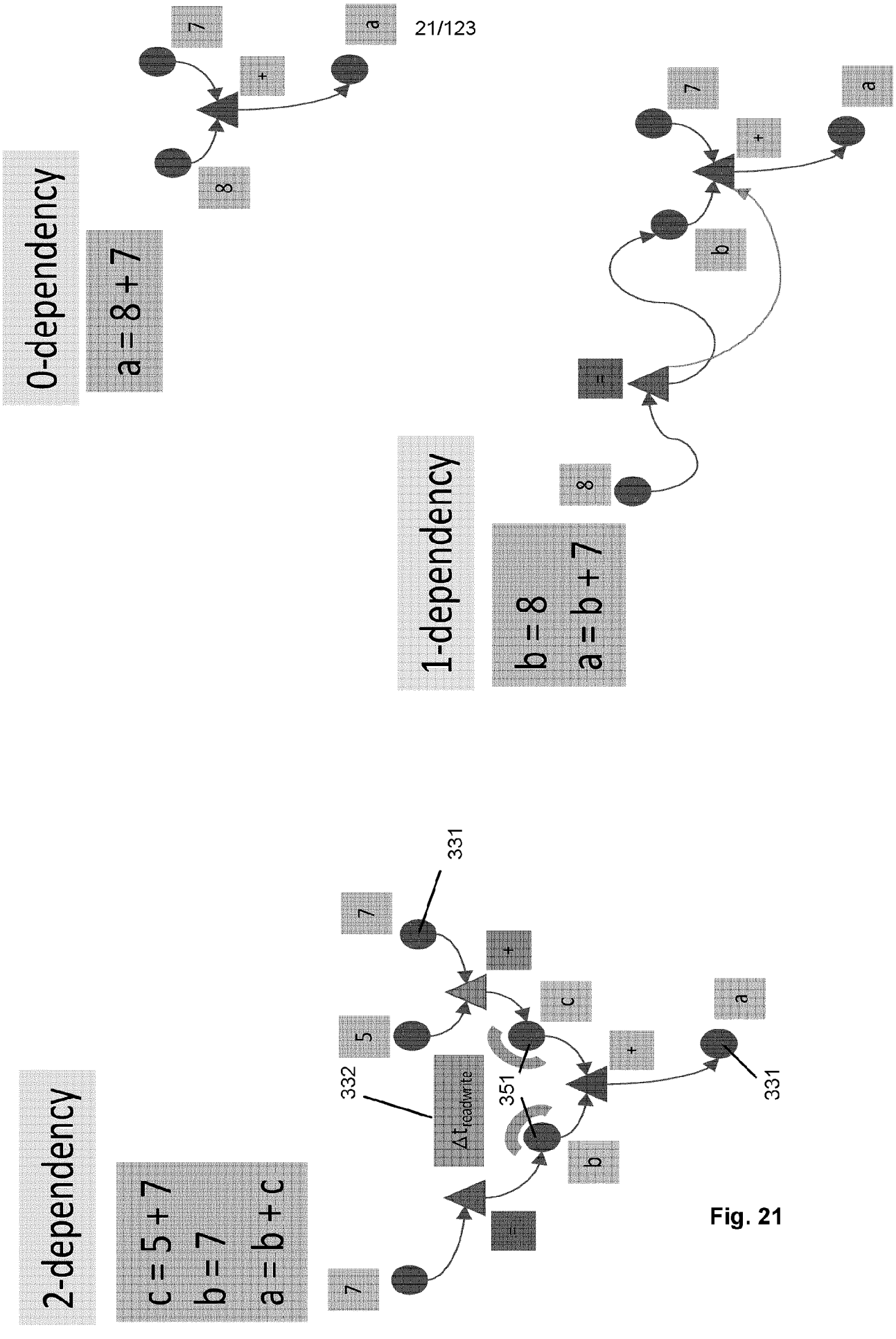

0-dependency
a = 8 + 7
7
8
+
a
1-dependency
b = 8
a = b + 7
7
8
=
b
+
a
2-dependency
c = 5 + 7
b = 7
a = b + c
7
5
331
332
$\Delta t_{readwrite}$
351
=
+
c
b
+
a
331

Fig. 21

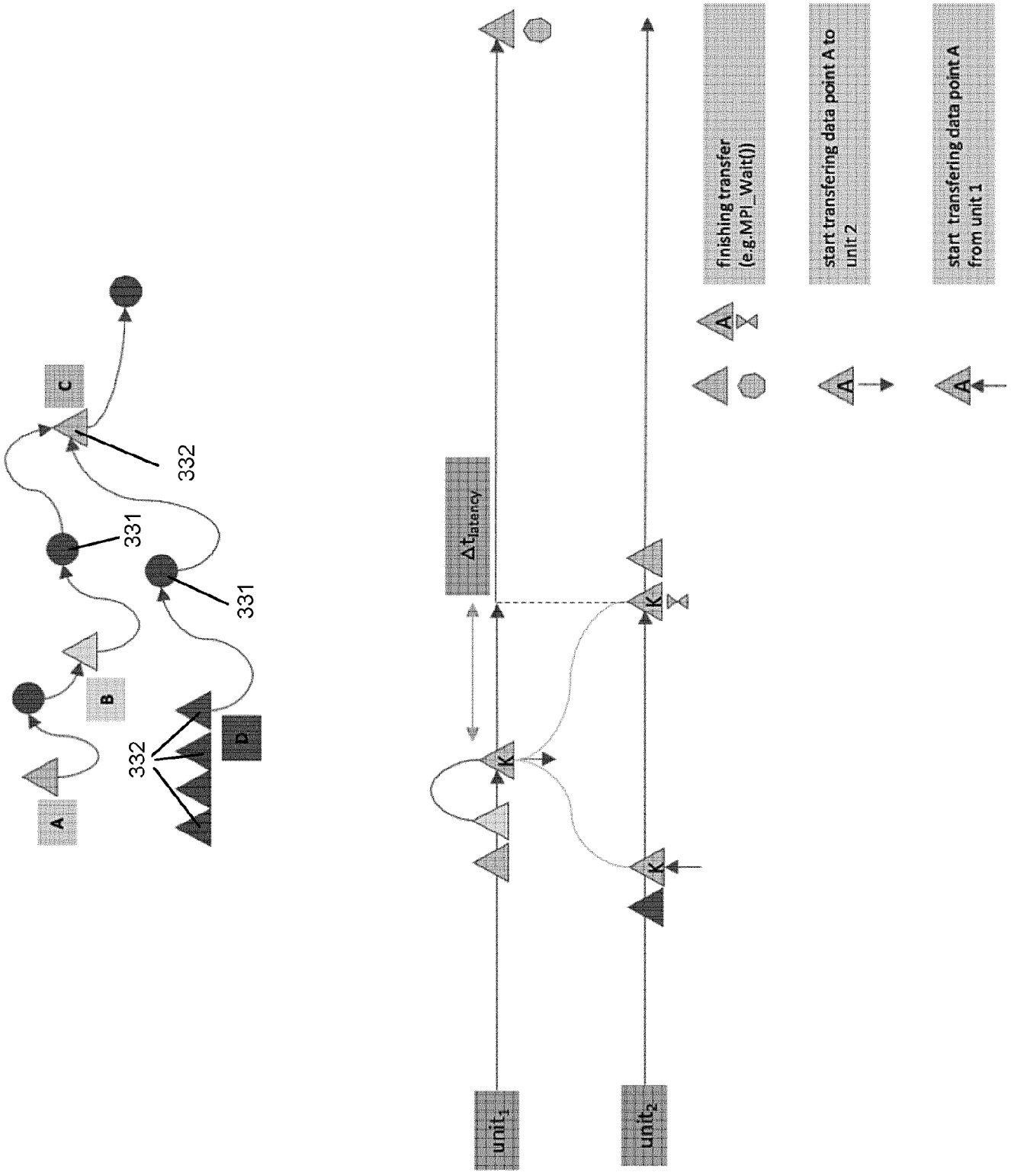
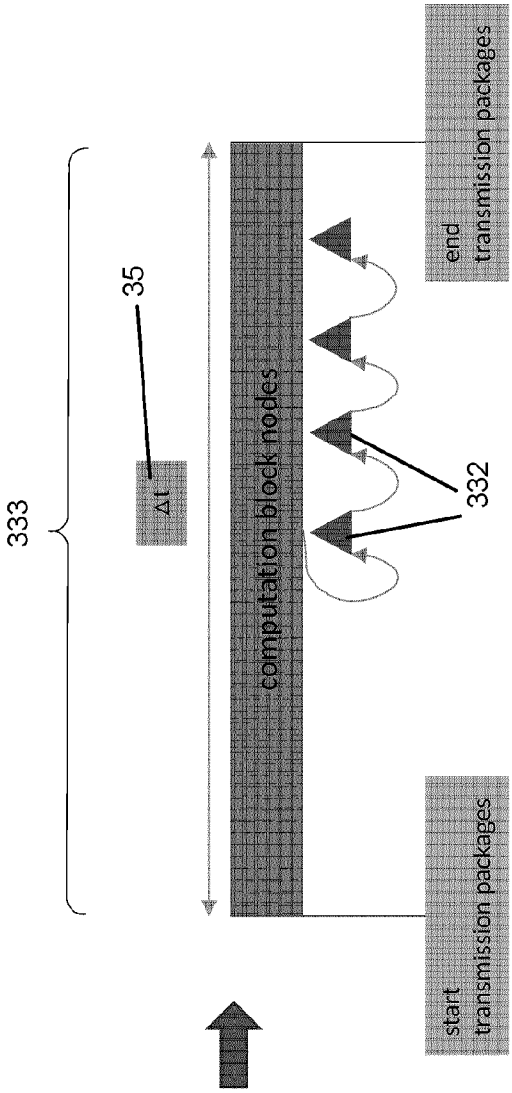

A
B
C
D
331
331
332
332
finishing transfer (e.g.MPI_Wait())
start transfering data point A to unit 2
start transfering data point A from unit 1
$\Delta t_{latency}$
K
K
K
$unit_1$
$unit_2$

Fig. 22


333
35
Δt
computation block nodes
332
end
transmission packages
start
transmission packages

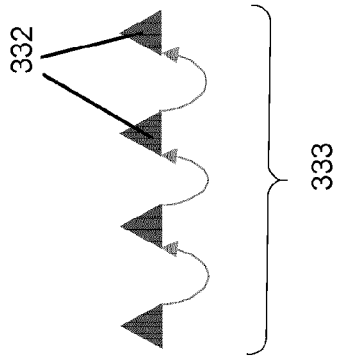

332
333

Fig. 23

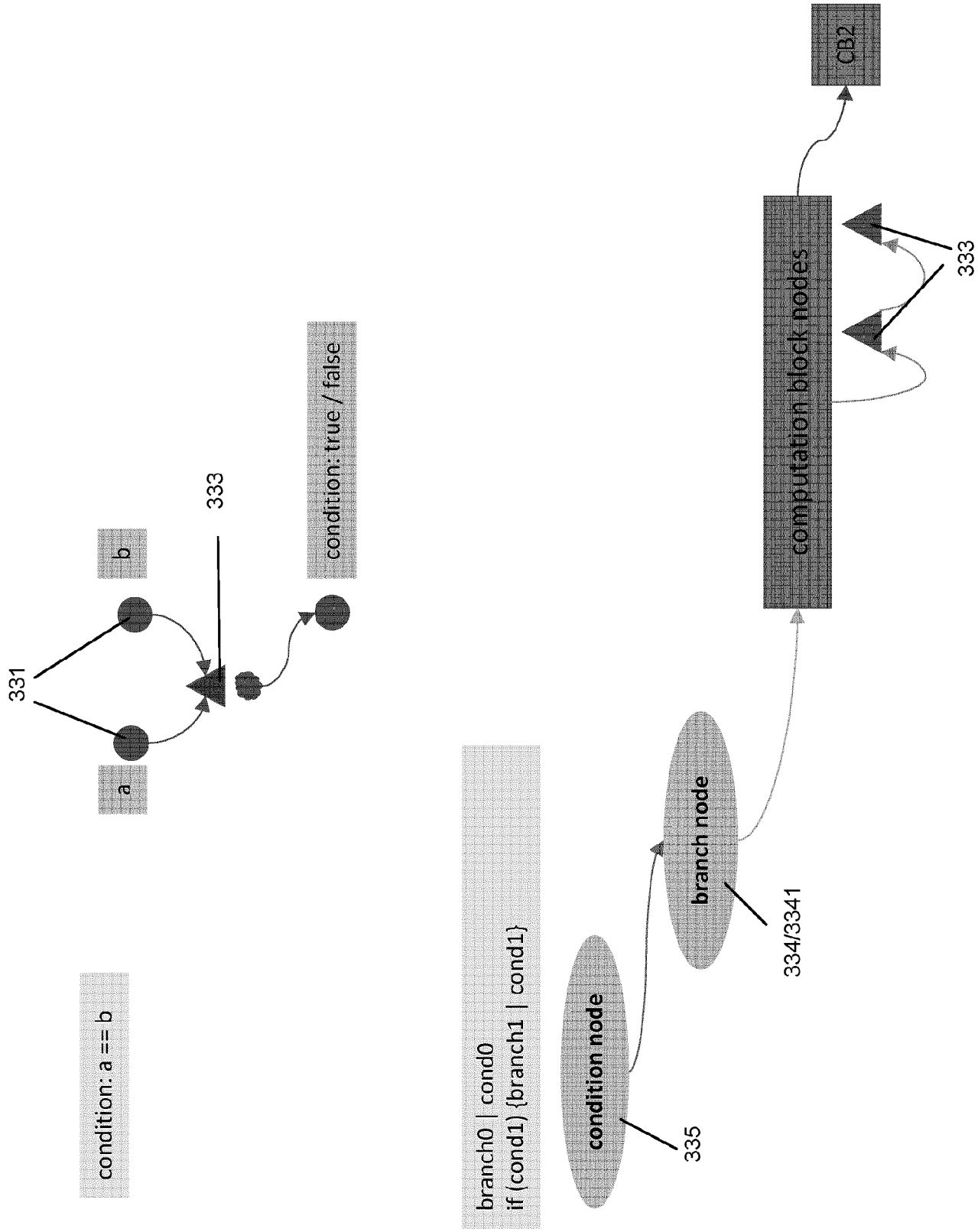

condition: a == b
331
a
b
333
condition: true / false
branch0 | cond0
if (cond1) {branch1 | cond1}
condition node
335
branch node
334/3341
computation block nodes
CB2
333

Fig. 24

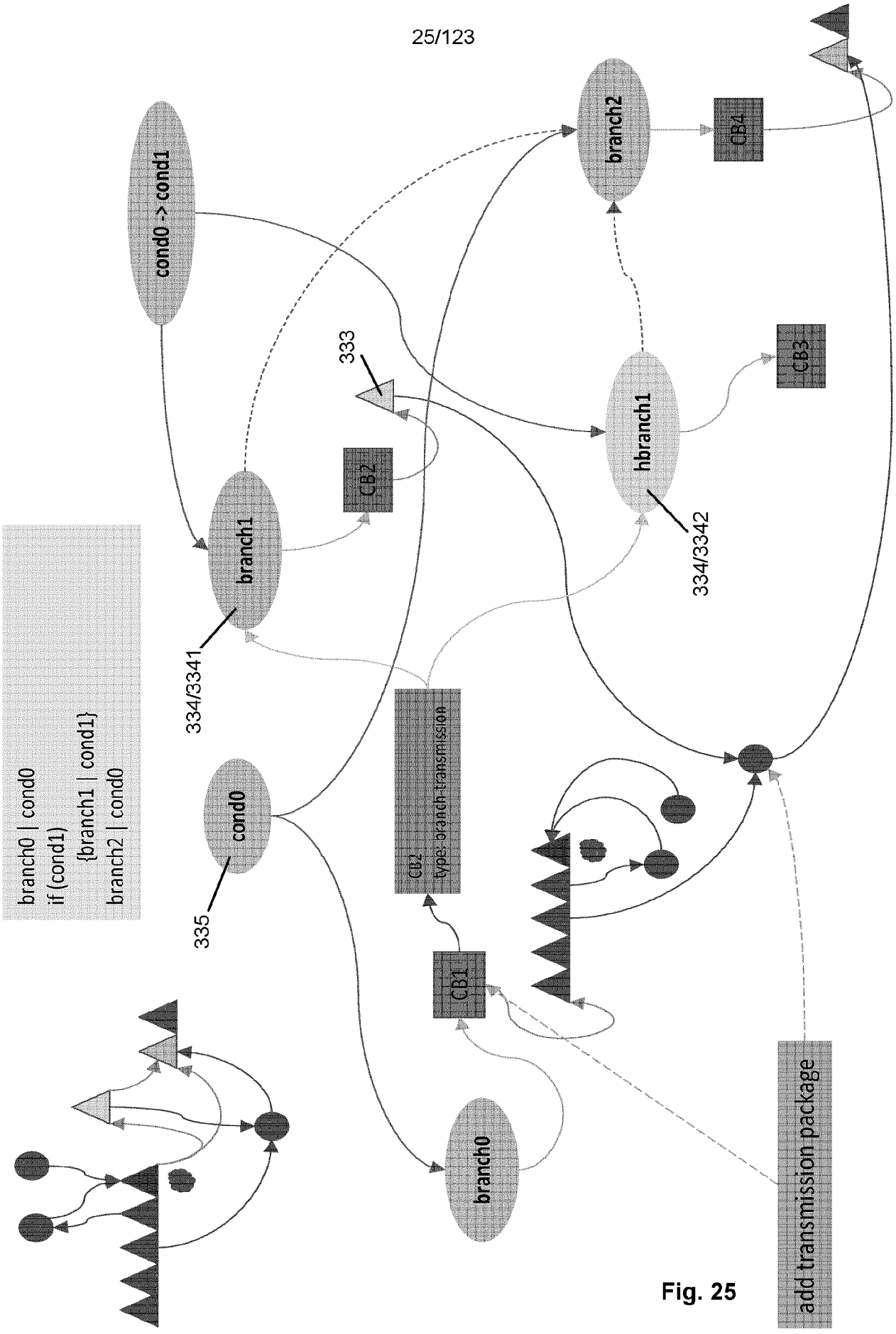

branch0 | cond0
if (cond1)
{branch1 | cond1}
branch2 | cond0
cond0 -> cond1
branch2
CB4
333
CB2
hbranch1
CB3
branch1
334/3342
334/3341
CB2
type: branch-transmission
cond0
335
CB1
branch0
add transmission package

Fig. 25

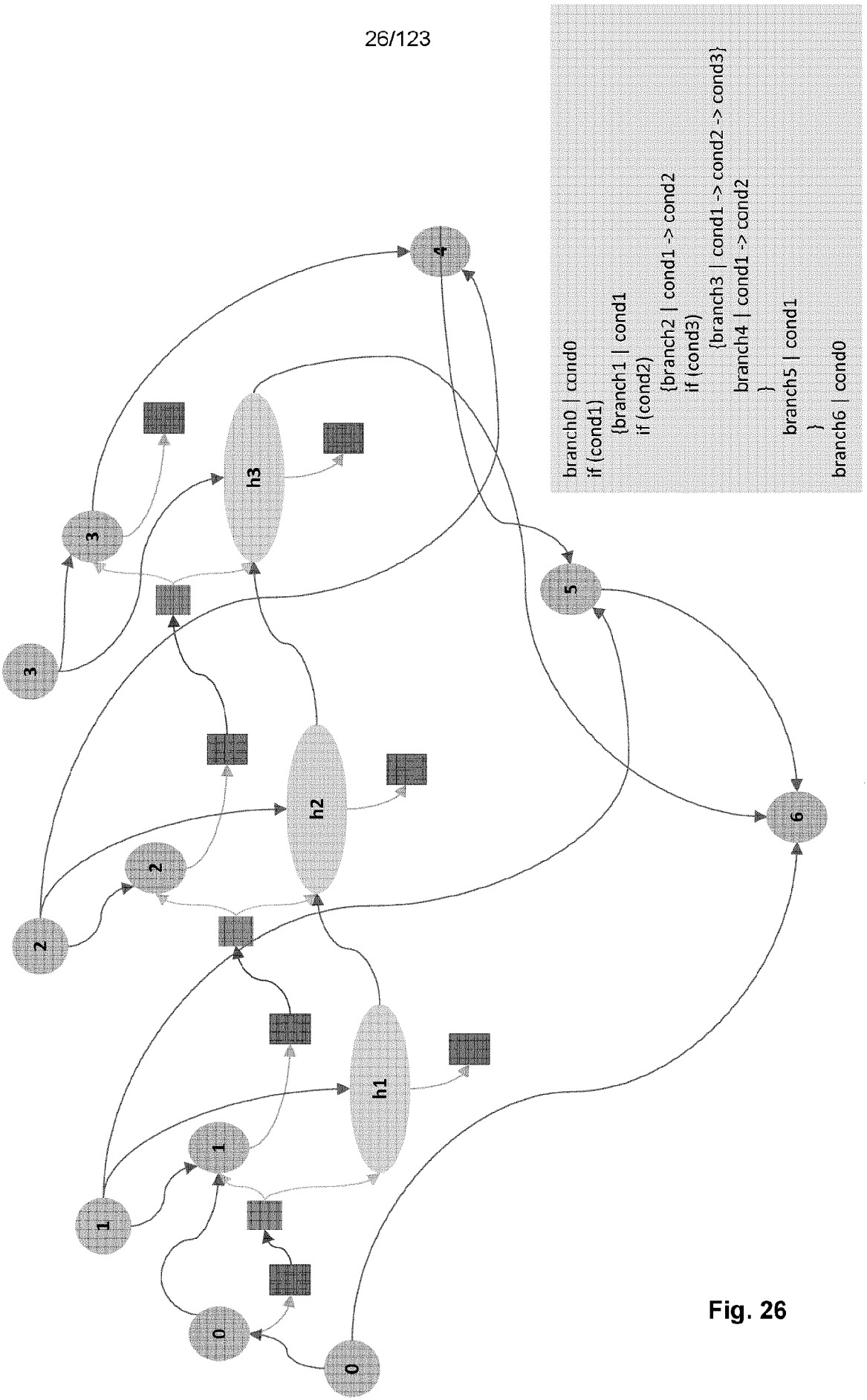

branch0 | cond0
if (cond1)
{branch1 | cond1
if (cond2)
{branch2 | cond1 -> cond2
if (cond3)
{branch3 | cond1 -> cond2 -> cond3}
branch4 | cond1 -> cond2
}
branch5 | cond1
}
branch6 | cond0
h1
h2
h3
0
1
2
3
4
5
6

Fig. 26

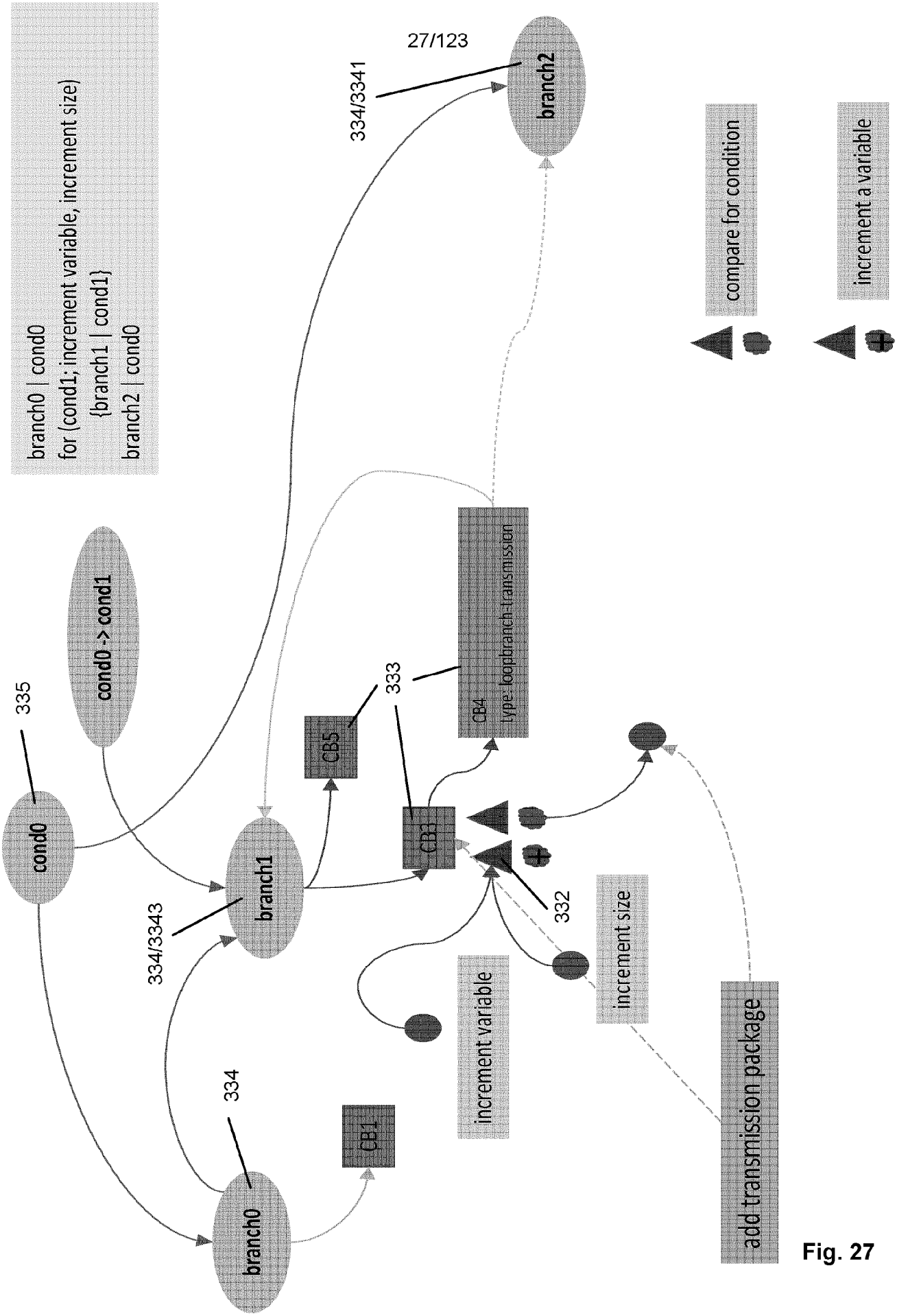

branch0 | cond0
for (cond1; increment variable, increment size)
{branch1 | cond1}
branch2 | cond0
27/123
334/3341
branch2
compare for condition
increment a variable
335
cond0
cond0 -> cond1
CB5
333
CB4
type: loopbranch-transmission
CB3
334/3343
branch1
332
increment size
increment variable
334
CB1
branch0
add transmission package

Fig. 27

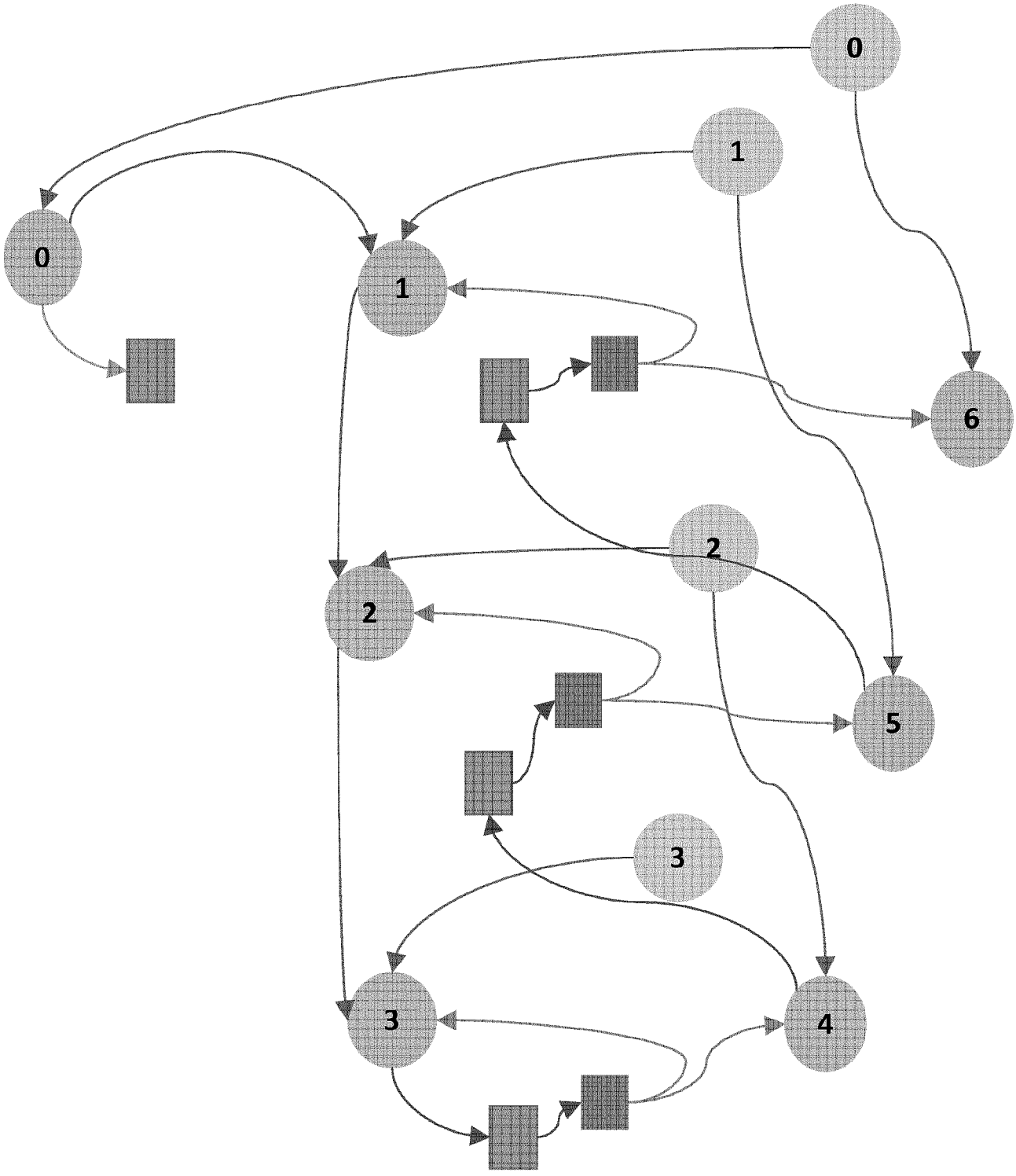


Fig. 28

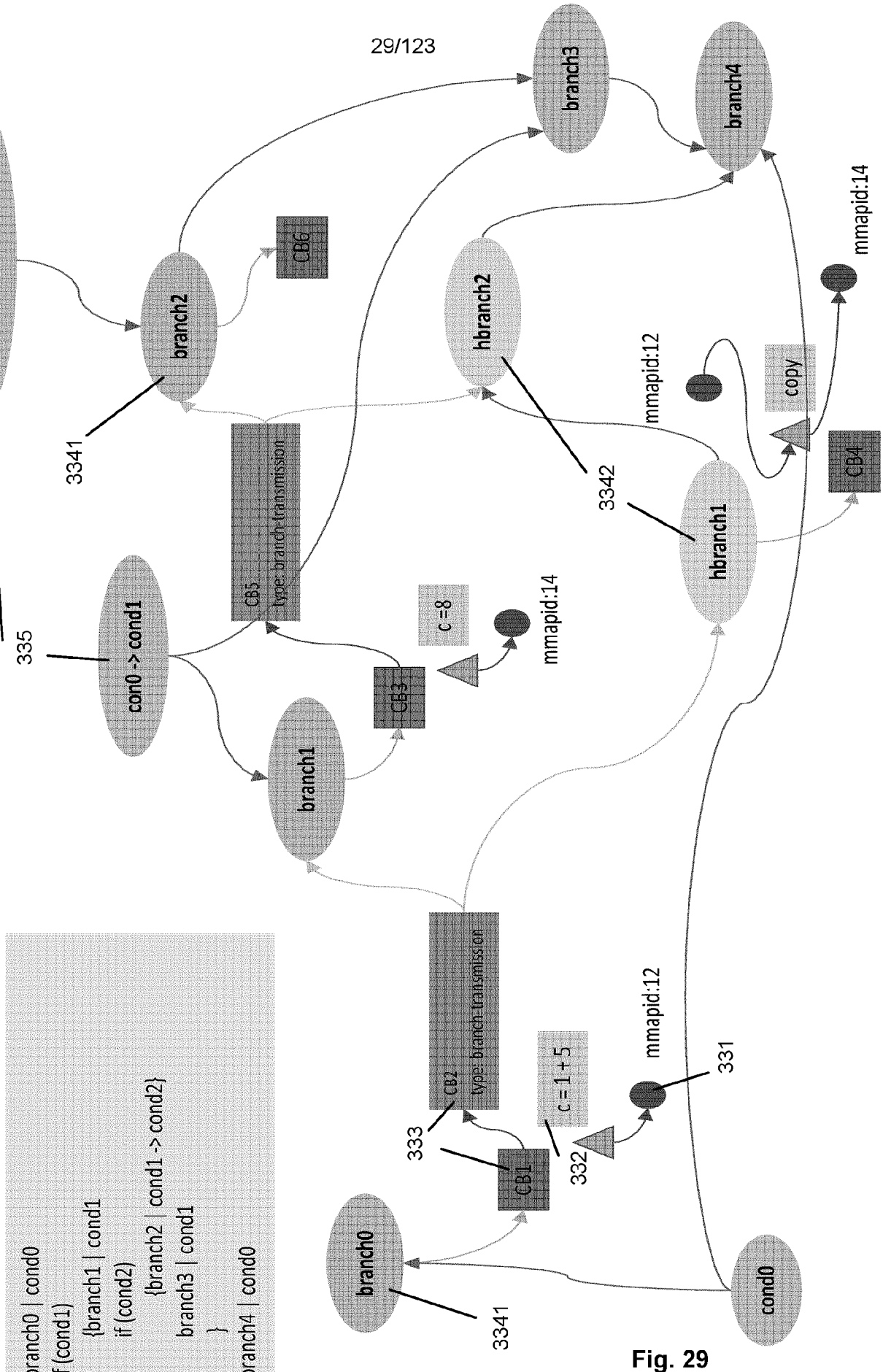

branch0 | cond0
if (cond1)
{branch1 | cond1
if (cond2)
{branch2 | cond1 -> cond2}
branch3 | cond1
}
branch4 | cond0
con0 -> cond1 -> cond2
branch2
CB6
branch3
branch4
3341
335
con0 -> cond1
CB5
type: branch-transmission
hbranch2
mmapid:12
copy
mmapid:14
CB4
c =8
mmapid:14
CB3
3342
hbranch1
branch1
CB2
type: branch-transmission
333
c = 1 + 5
CB1
332
mmapid:12
331
branch0
3341
cond0

Fig. 29

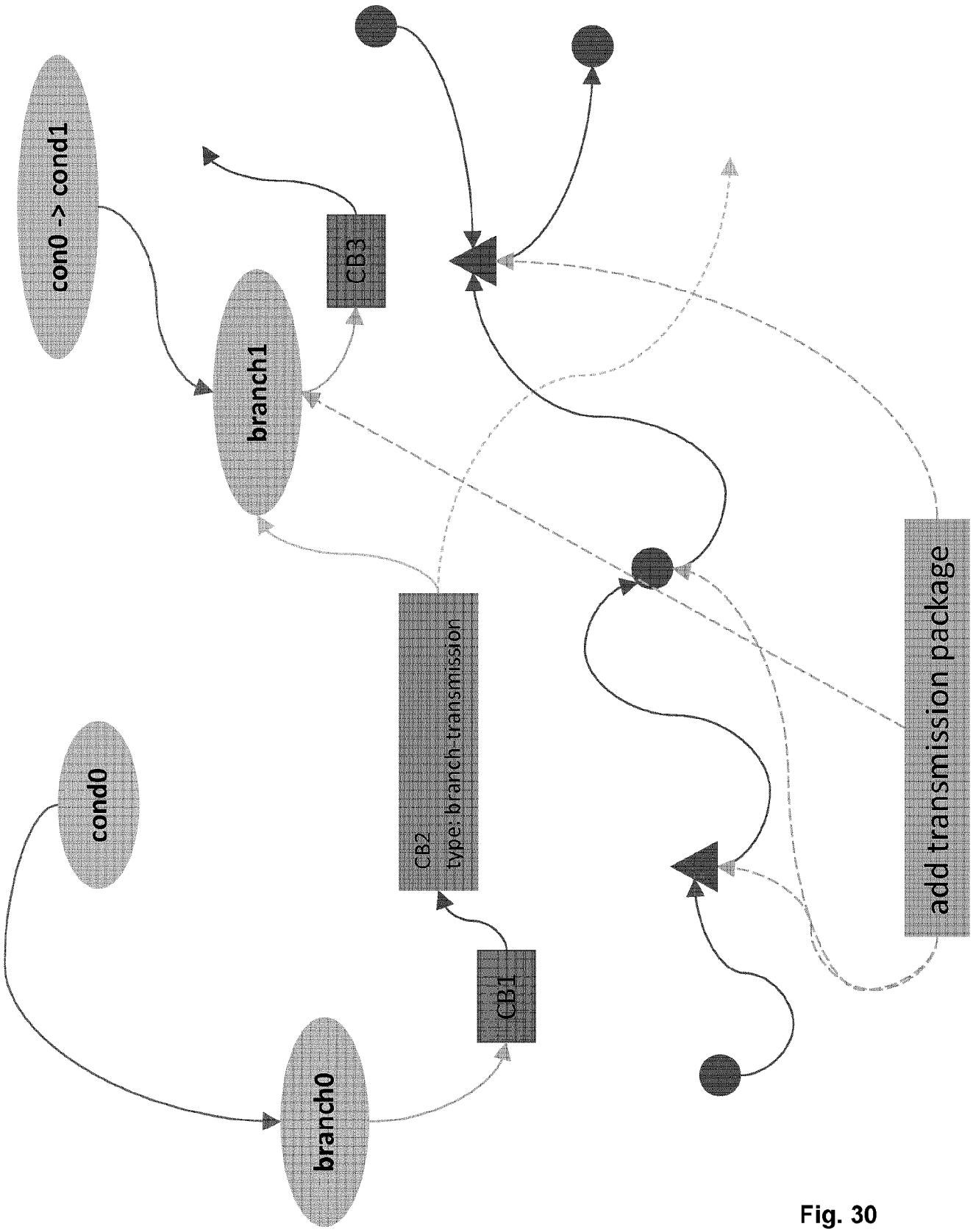

con0 -> cond1
CB3
branch1
CB2
type: branch-transmission
add transmission package
cond0
CB1
branch0

Fig. 30

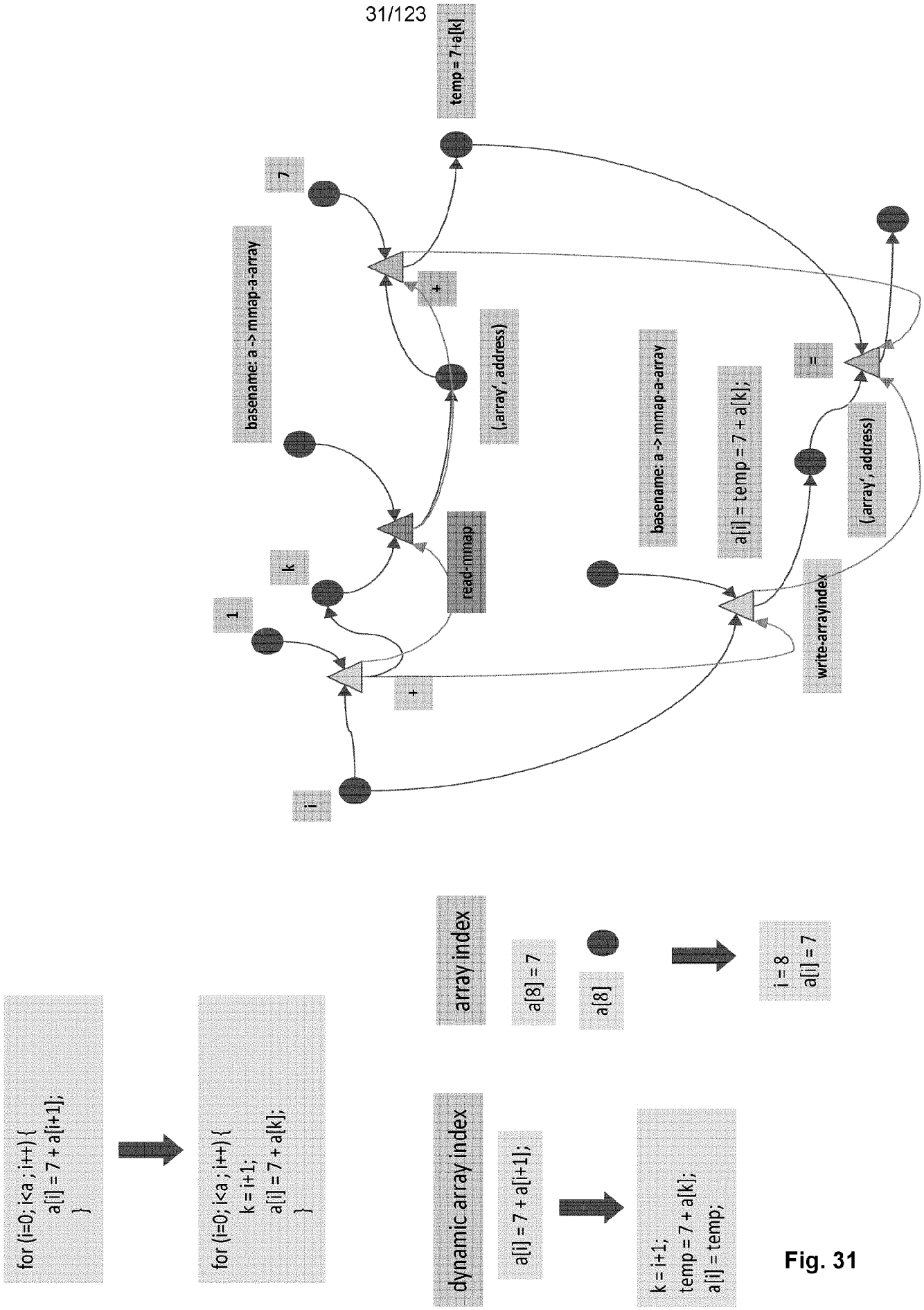

for (i=0; i<a ; i++) {
a[i] = 7 + a[i+1];
}
for (i=0; i<a ; i++) {
k = i+1;
a[i] = 7 + a[k];
}
dynamic array index
a[i] = 7 + a[i+1];
k = i+1;
temp = 7 + a[k];
a[i] = temp;
array index
a[8] = 7
a[8]
i = 8
a[i] = 7
temp = 7+a[k]
7
basename: a -> mmap-a-array
+
(‚array', address)
k
read-mmap
1
+
i
basename: a -> mmap-a-array
a[i] = temp = 7 + a[k];
=
(‚array', address)
write-arrayindex

Fig. 31

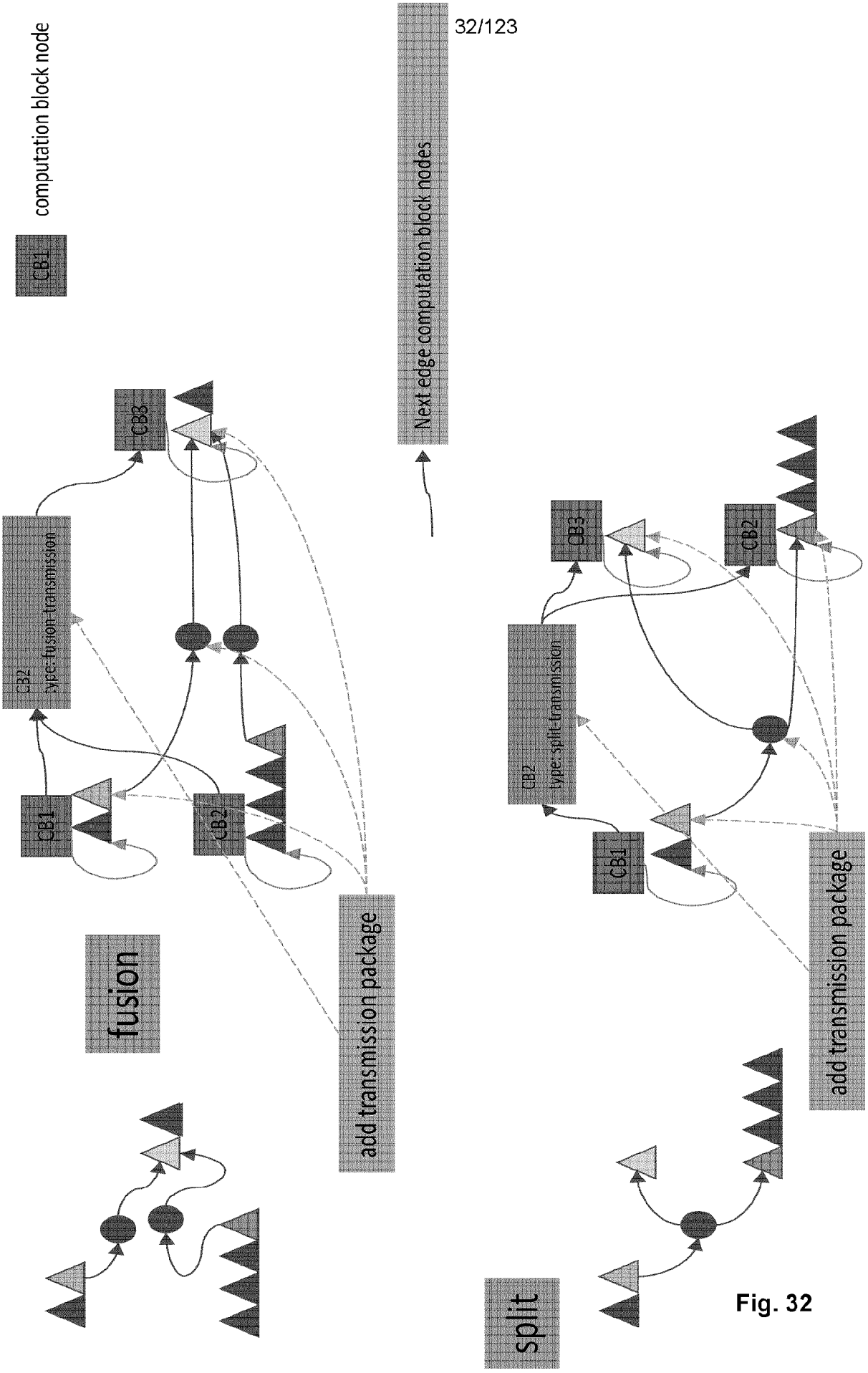
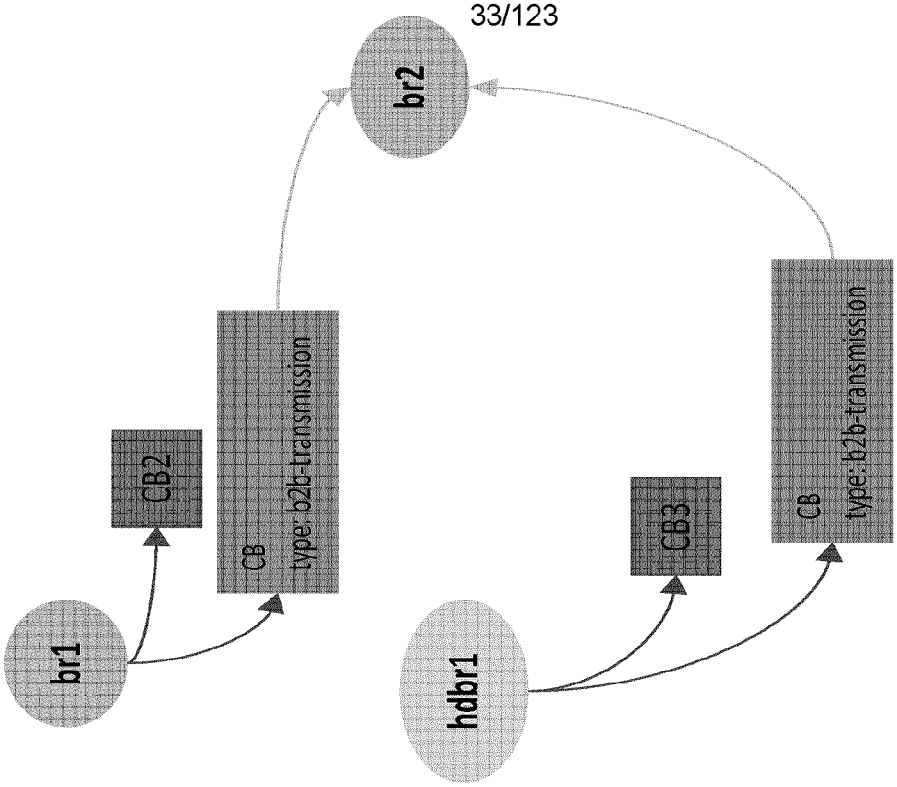

computation block node
CB1
Next edge computation block nodes
CB3
CB2
type: fusion-transmission
CB1
CB2
fusion
add transmission package
CB3
CB2
CB2
type: split-transmission
CB1
add transmission package
split

Fig. 32


br2
CB2
CB type: b2b-transmission
br1
CB3
CB type: b2b-transmission
hdbr1

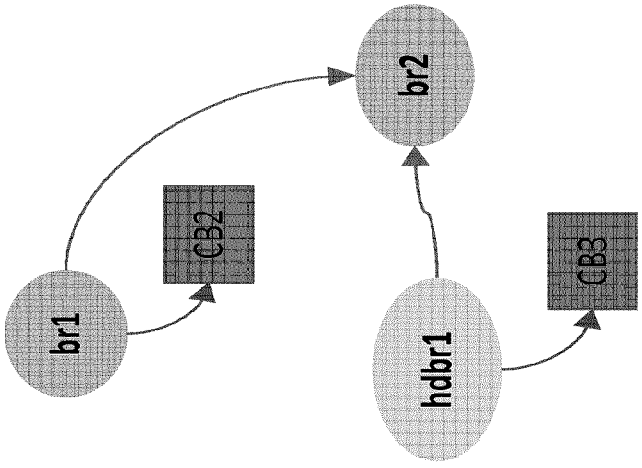

br2
CB2
br1
CB3
hdbr1

Fig. 33

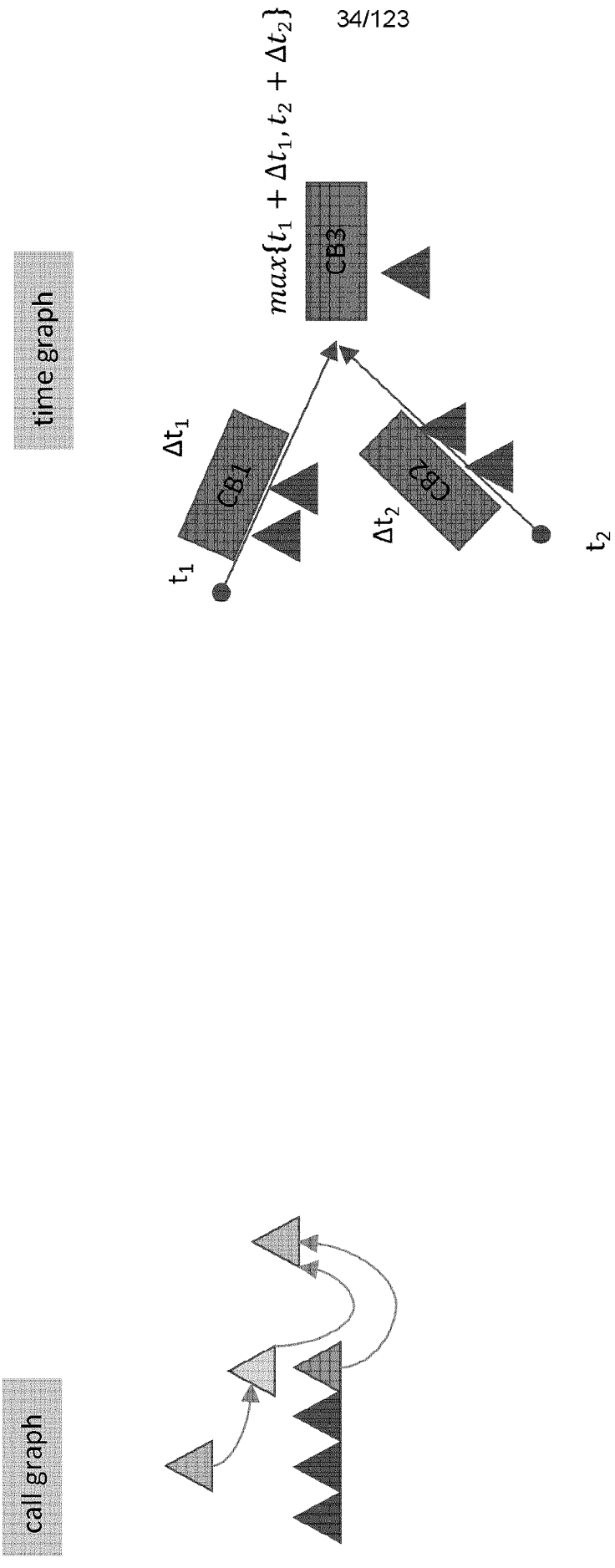

time graph
$max\{t_1 + \Delta t_1, t_2 + \Delta t_2\}$
CB3
Δt_1
CB1
t_1
Δt_2
CB2
t_2
call graph

Fig. 34

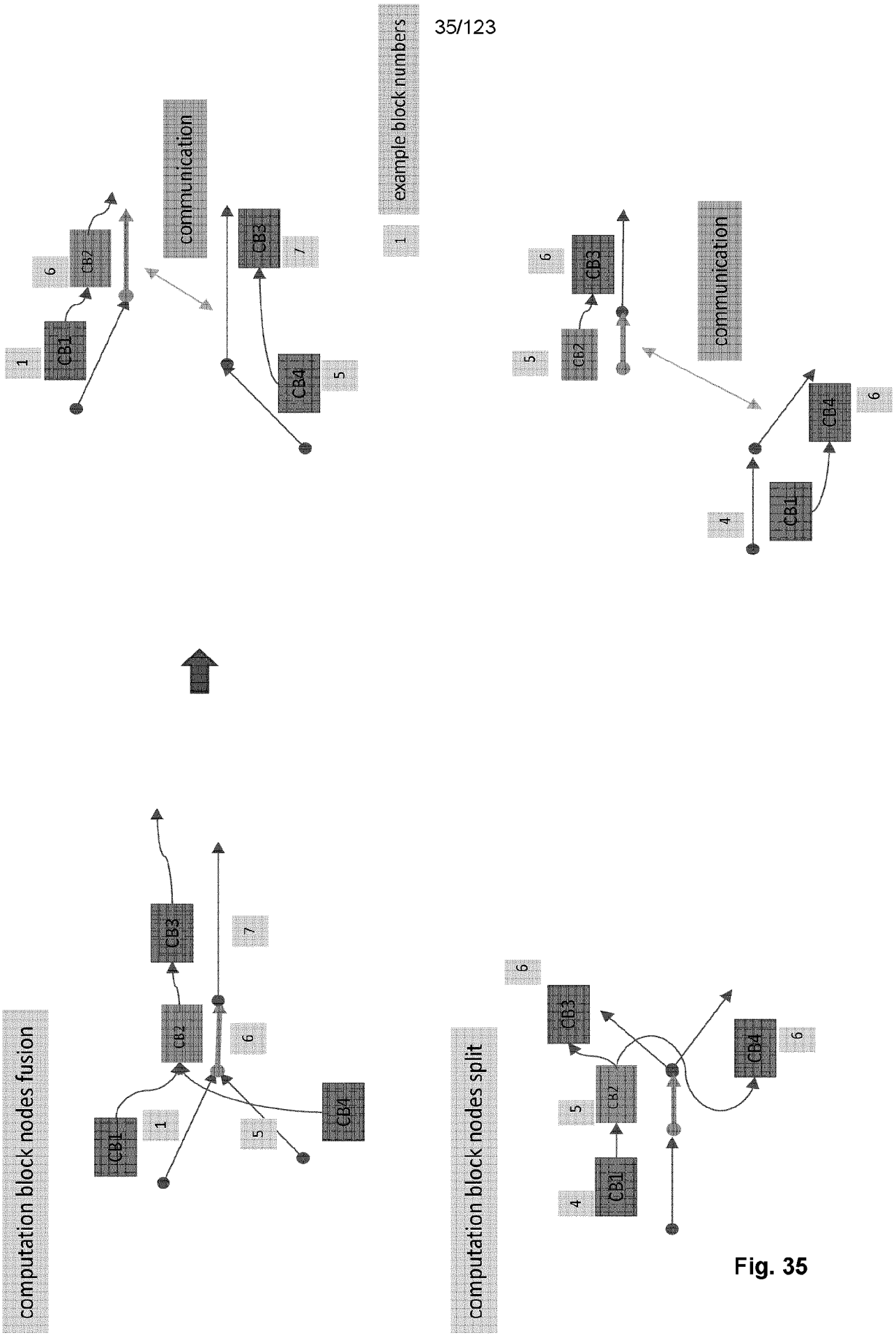

example block numbers
1
communication
CB1
CB2
CB3
CB4
1
6
7
5
communication
CB1
CB2
CB3
CB4
4
5
6
6
computation block nodes fusion
CB1
CB2
CB3
CB4
1
6
7
5
computation block nodes split
CB1
CB2
CB3
CB4
4
5
6
6

Fig. 35

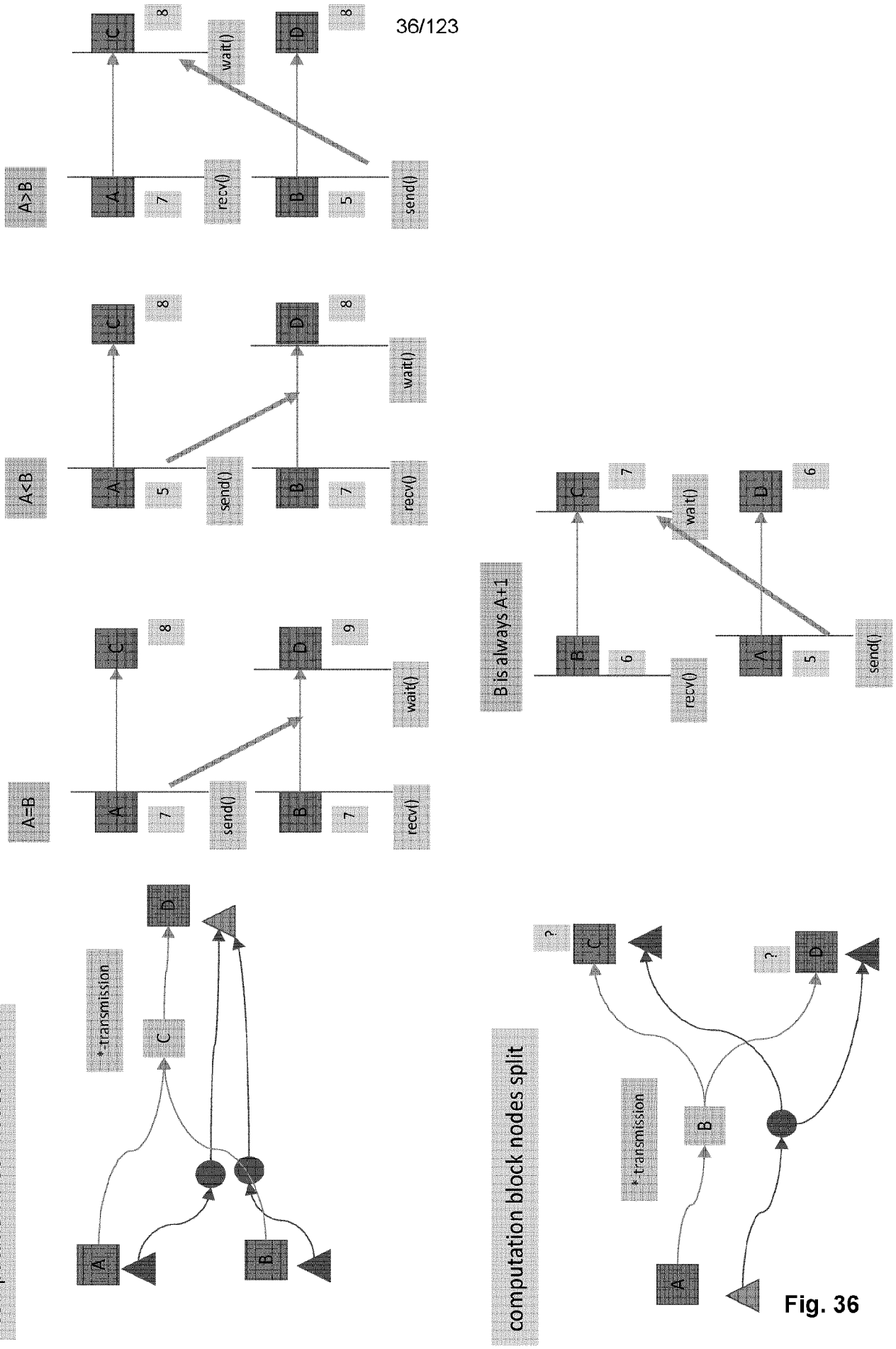
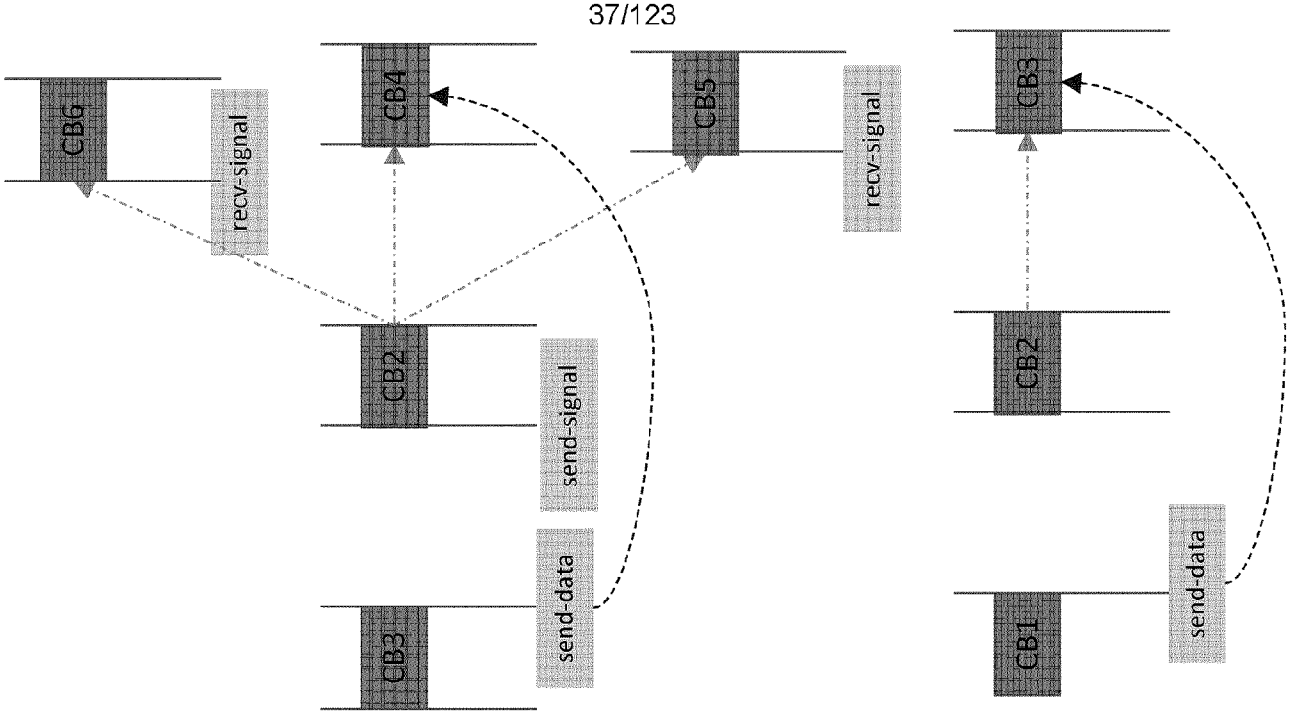
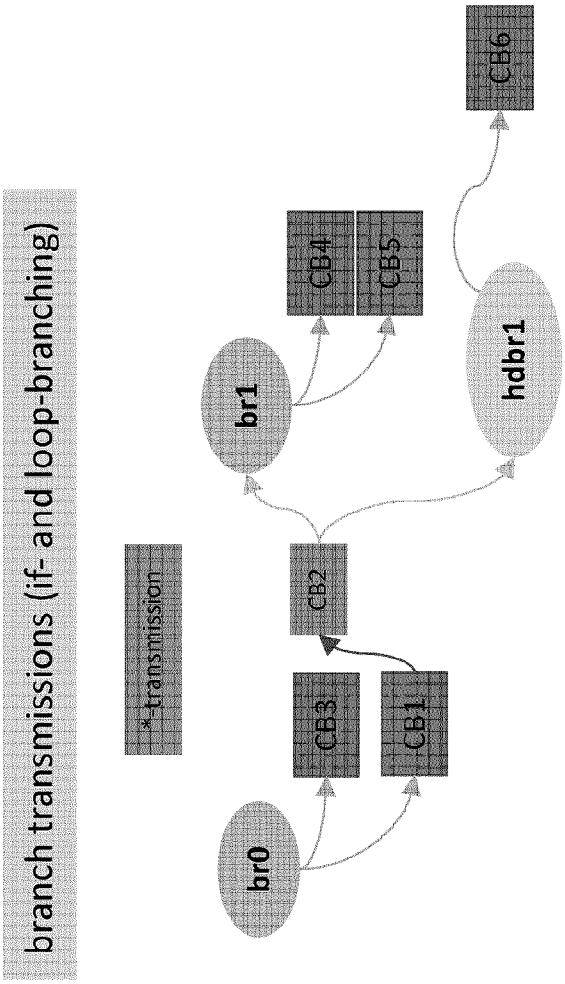

A>B
A
7
recv()
C
8
wait()
B
5
send()
D
8
A<B
A
5
send()
C
8
B
7
recv()
D
8
wait()
A=B
A
7
send()
C
8
B
7
recv()
D
9
wait()
B is always A+1
B
6
recv()
C
7
wait()
A
5
send()
D
6
computation block nodes fusion
A
B
* transmission
C
D
computation block nodes split
A
B
* transmission
C
?
D
?

Fig. 36


CB6
recv-signal
CB4
CB5
recv-signal
CB3
CB2
send-signal
send-data
CB3
CB2
CB1
send-data


branch transmissions (if- and loop-branching)
*-transmission
br0
CB3
CB1
CB2
br1
CB4
CB5
hdbr1
CB6

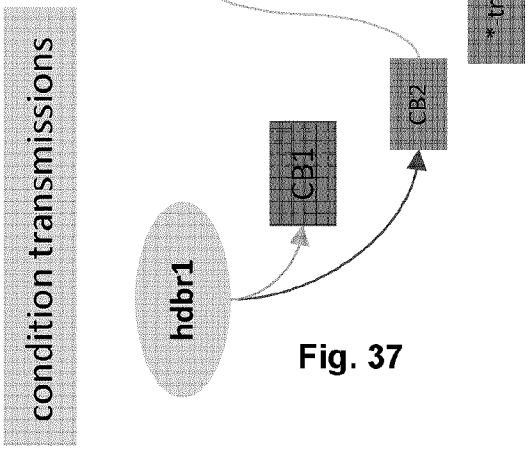

condition transmissions
hdbr1
CB1
CB2
br2
CB3
* transmission

Fig. 37

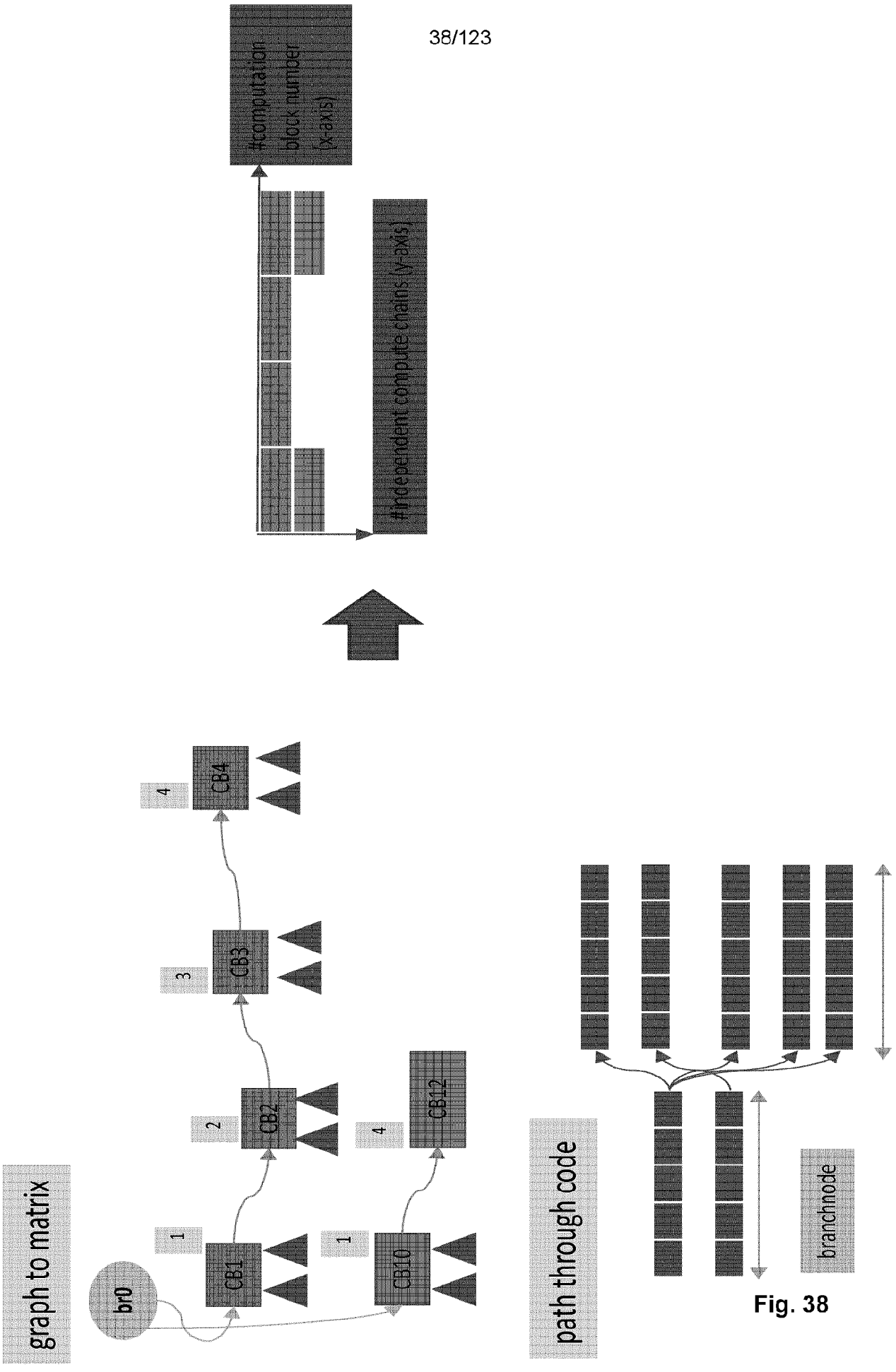

#computation block number (x-axis)
#independent compute chains (y-axis)
graph to matrix
br0
CB1
CB10
CB2
CB12
CB3
CB4
1
1
2
4
3
4
path through code
branchnode

Fig. 38

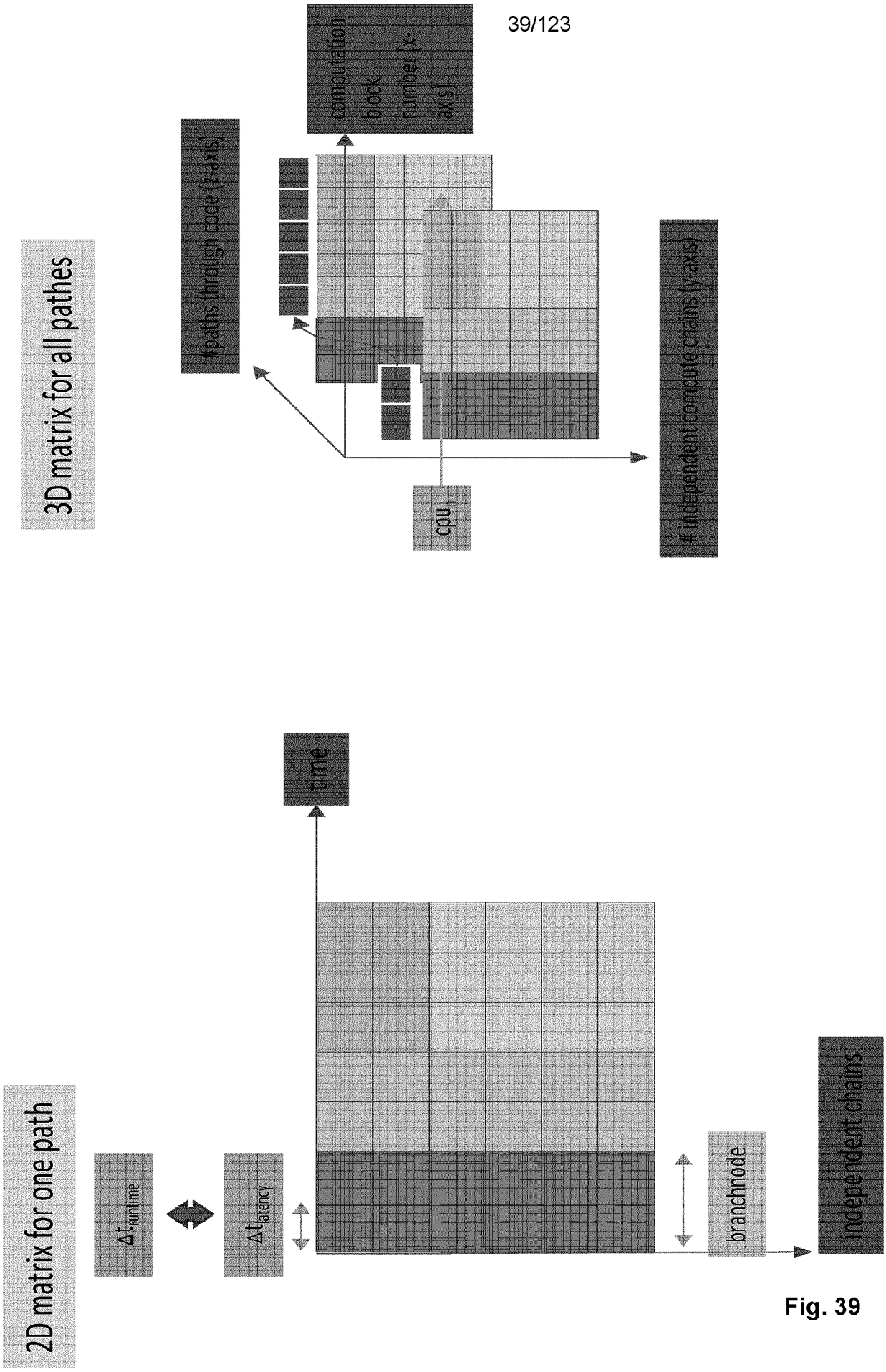

3D matrix for all pathes
#paths through code (z-axis)
computation block number (x-axis)
cpu$_n$
independent compute chains (y-axis)
2D matrix for one path
$\Delta t_{runtime}$
$\Delta t_{latency}$
time
branchrode
independent chains

Fig. 39

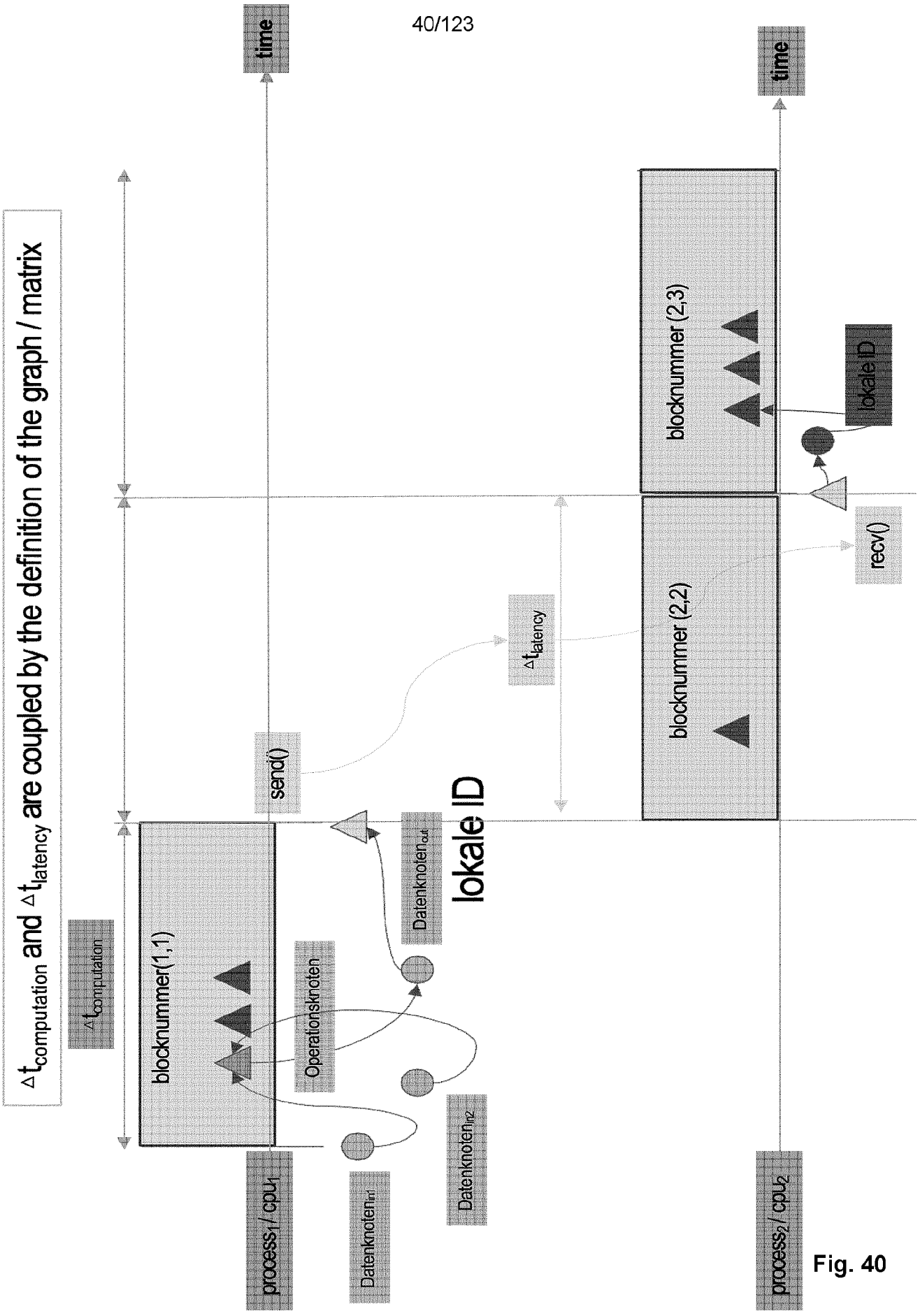

Δt computation and Δt latency are coupled by the definition of the graph / matrix
time
time
Δt computation
blocknummer(1,1)
Operationsknoten
Datenknoten out
send()
Δt latency
lokale ID
blocknummer (2,3)
blocknummer (2,2)
lokale ID
recv()
process1 / cpu1
Datenknoten in1
Datenknoten in2
process2 / cpu2

Fig. 40

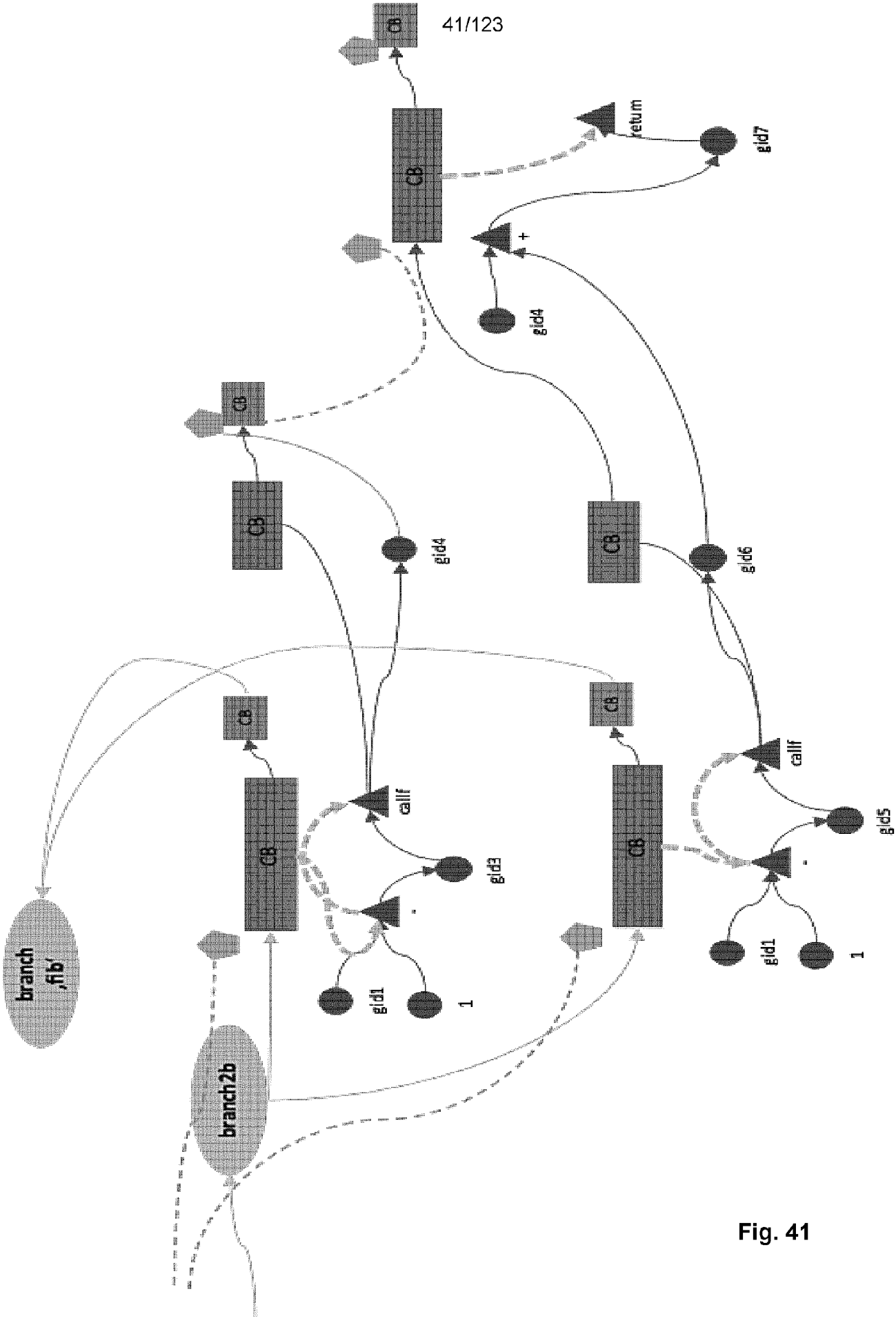

CB
return
gId7
+
gId4
CB
CB
gId4
CB
gId6
CB
CB
callf
gId3
.
gId1
1
branch
‚fib'
branch2b
CB
CB
callf
gId5
.
gId1
1

Fig. 41

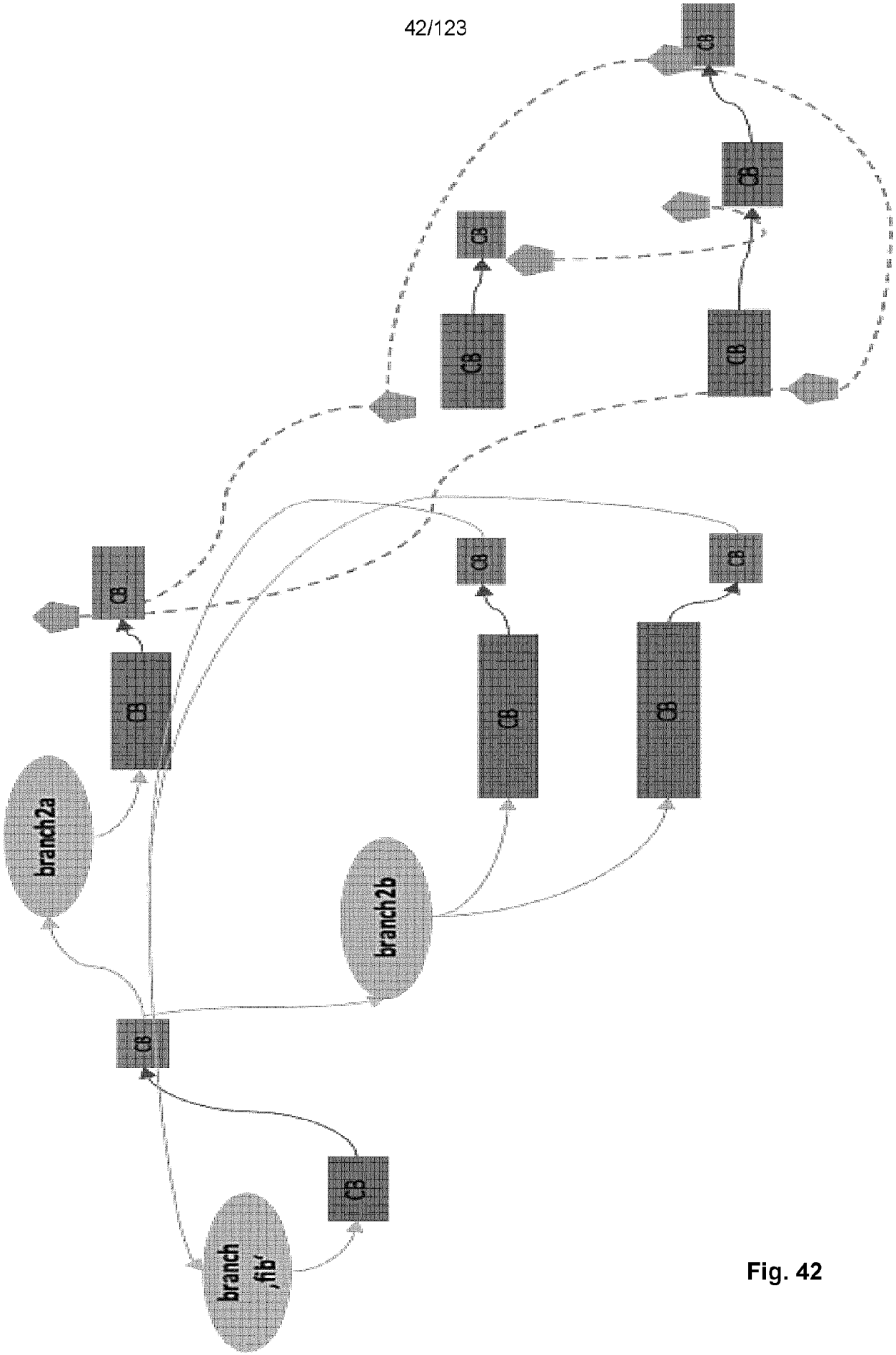

CB
CB
CB
CB
CB
CB
CB
CB
CB
CB
CB
CB
CB
branch2a
branch2b
branch
„fib"

Fig. 42

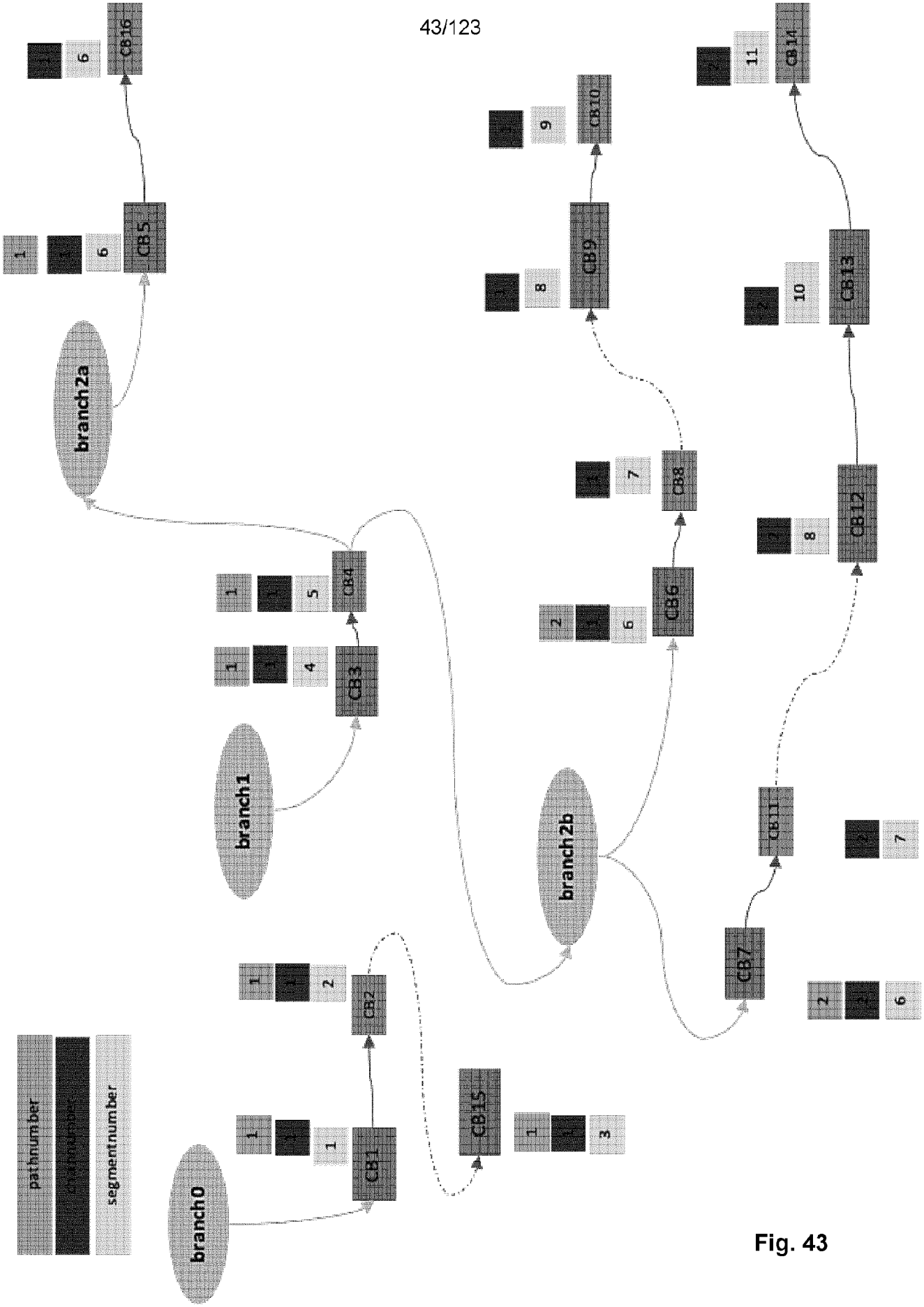

branch0
branch1
branch2a
branch2b
CB1
CB2
CB3
CB4
CB5
CB6
CB7
CB8
CB9
CB10
CB11
CB12
CB13
CB14
CB15
CB16
pathnumber
segmentnumber

Fig. 43

cbid	segm,chain,path	x	y	z
CB1	1,1,1	1	1	1,2
CB2	2,1,1	2	1	1,2
CB15	3,1,1	3	1	1,2
CB3	4,1,1	4	1	1,2
CB4	5,1,1	5	1	1
CB6	6,1,2	6	1	2
CB8	7,1,2	7	1	2
CB9	8,1,2	8	1	2
CB10	9,1,2	9	1	2
CB7	6,2,2	6	2	2
CB11	7,2,2,	7	2	2
CB12	8,2,2	8	2	2
CB13	10,2,2	10	2	2
CB14	11,2,2	11	2	2

Fig. 44

Transfer-Matrix

2	1	2	3	4	5	6	7	8	9	10	11
1	e: COM1->		s: ->COM10	e: SIG1-> COM2-> s: ->COM1 ->COM4 ->COM5		s: ->SIG1 ->COM2 e: COM4->		s: ->COM8	s: COM6->		
2						s: ->SIG1 ->COM2 e: COM5->		s: ->COM9		s: ->COM6	s: COM7->

1	1	2	3	4	5	6	7	8	9	10	11
1	e: COM1->		s: ->COM10	e: SIG1-> COM2-> s: ->COM1 ->COM4 ->COM5		s: ->SIG1 ->COM2 e: COM3->					
2											

Fig. 45

	source cbid		target cbid			direction
	start	end	start	end	variable	
COM1		CB1 - (1,1,1)	CB3 - (4,1,1)		gid8, addr CB15	forward
SIG1		CB3 - (4,1,1)	CB5 - (6,1,1), CB6 - (6,1,2), CB7 - (6,2,2)		gid2	forward
COM2		CB3 - (4,1,1)	CB5 - (6,1,1), CB6 - (6,1,2), CB7 - (6,2,2)		gid1	forward
COM3		CB5 - (6,1,1)	CB15 / CB9 / CB12		gid1	
COM4		CB6 - (6,1,2)	CB3 - (4,1,1)		gid3, addr CB9	backward
COM5		CB7 - (6,2,2)	CB3 - (4,1,1)		gid5, addr CB12	backward
COM6		CB9 - (9,1,2)	CB10 - (10,2,2)		gid4	forward
COM7		CB13 - (10,2,2)	CB15 / CB9 / CB12		gid7	
COM8		LL	CB9 - (8,1,2)		gid4	
COM9		LL	CB12 - (8,2,2)		gid6	
COM10		LL	CB15 - (3,1,1)		gid9	

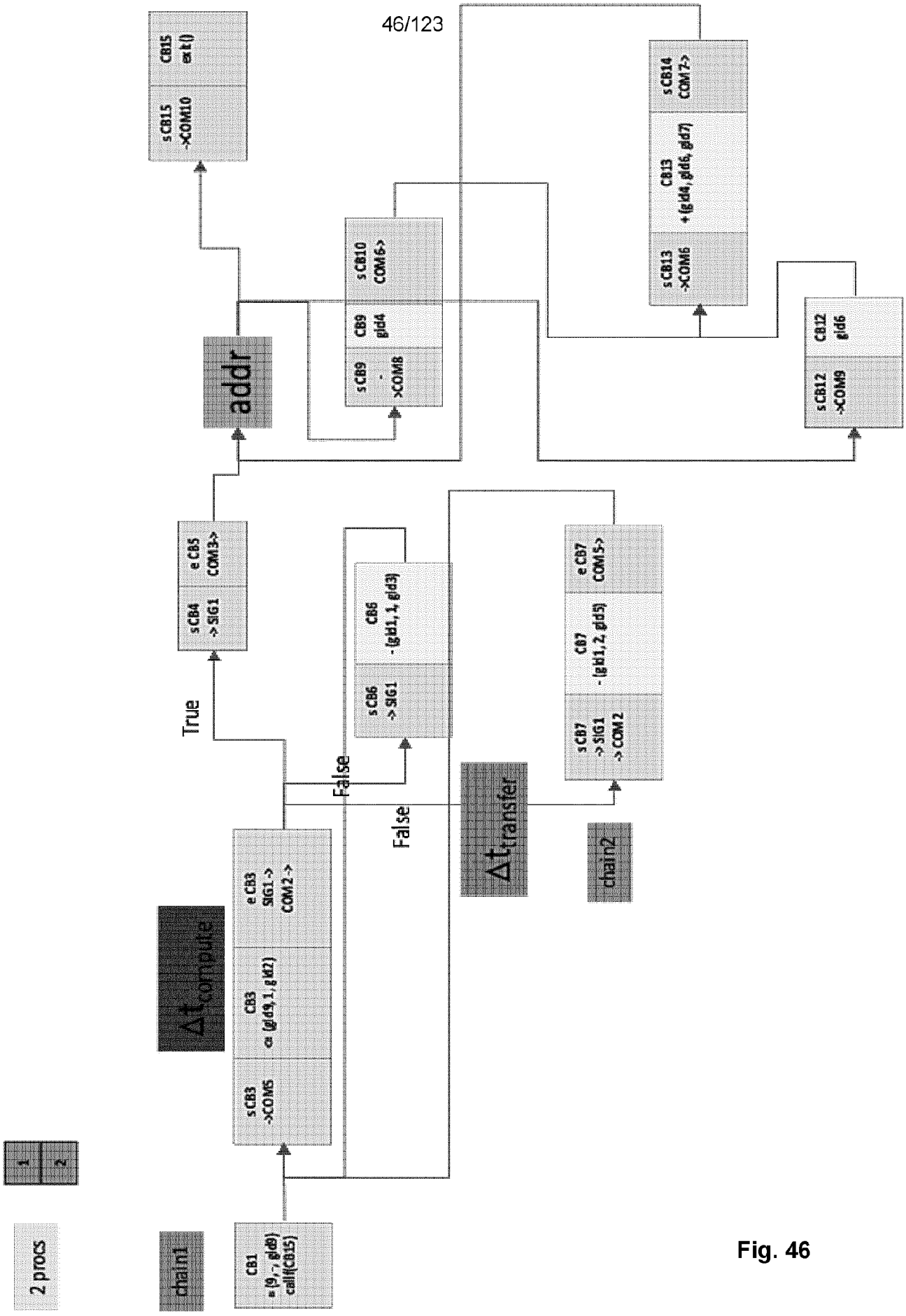

2 procs
1
2
chain1
CB1
chain2
CB15
ext()
sCB15
->COM10
addr
sCB10
COM6->
CB9
gId4
sCB9
->COM8
sCB14
COM7->
CB13
+ (gId4, gId6, gId7)
sCB13
->COM6
CB12
gId6
sCB12
->COM9
eCB5
COM3->
sCB4
->SIG1
True
CB6
-(gId1, 1, gId3)
sCB6
->SIG1
False
eCB7
COM5->
CB7
-(gId1, 2, gId5)
sCB7
->SIG1
->COM2
False
eCB3
SIG1->
COM2->
CB3
α (gId3, 1, gId2)
sCB3
->COM5
Δt_compute
Δt_transfer

Fig. 46

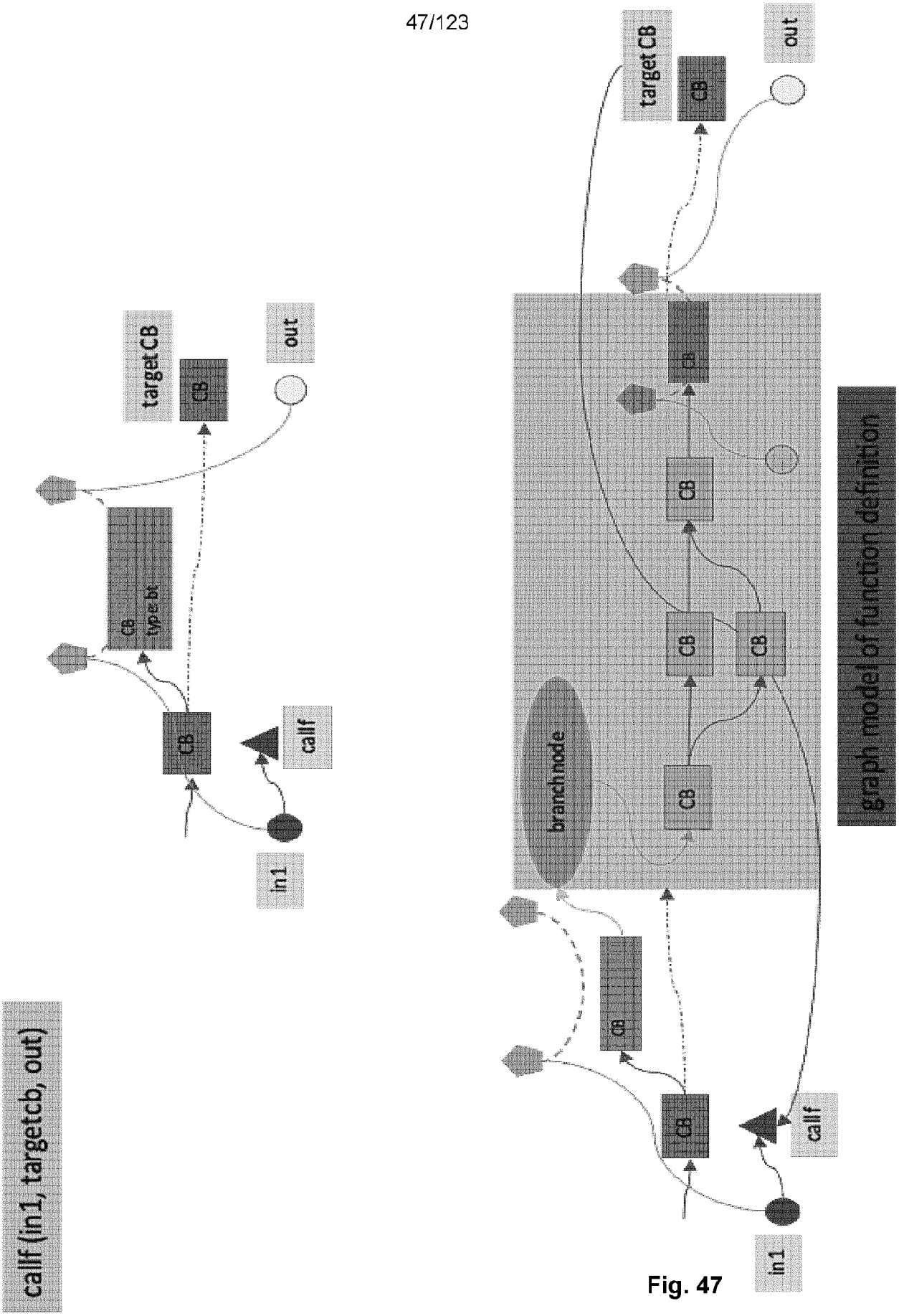

callf (in1, targetcb, out)
target CB
CB
out
CB
type bt
CB
callf
in1
target CB
CB
out
CB
CB
CB
CB
branchnode
CB
CB
callf
in1
graph model of function definition

Fig. 47

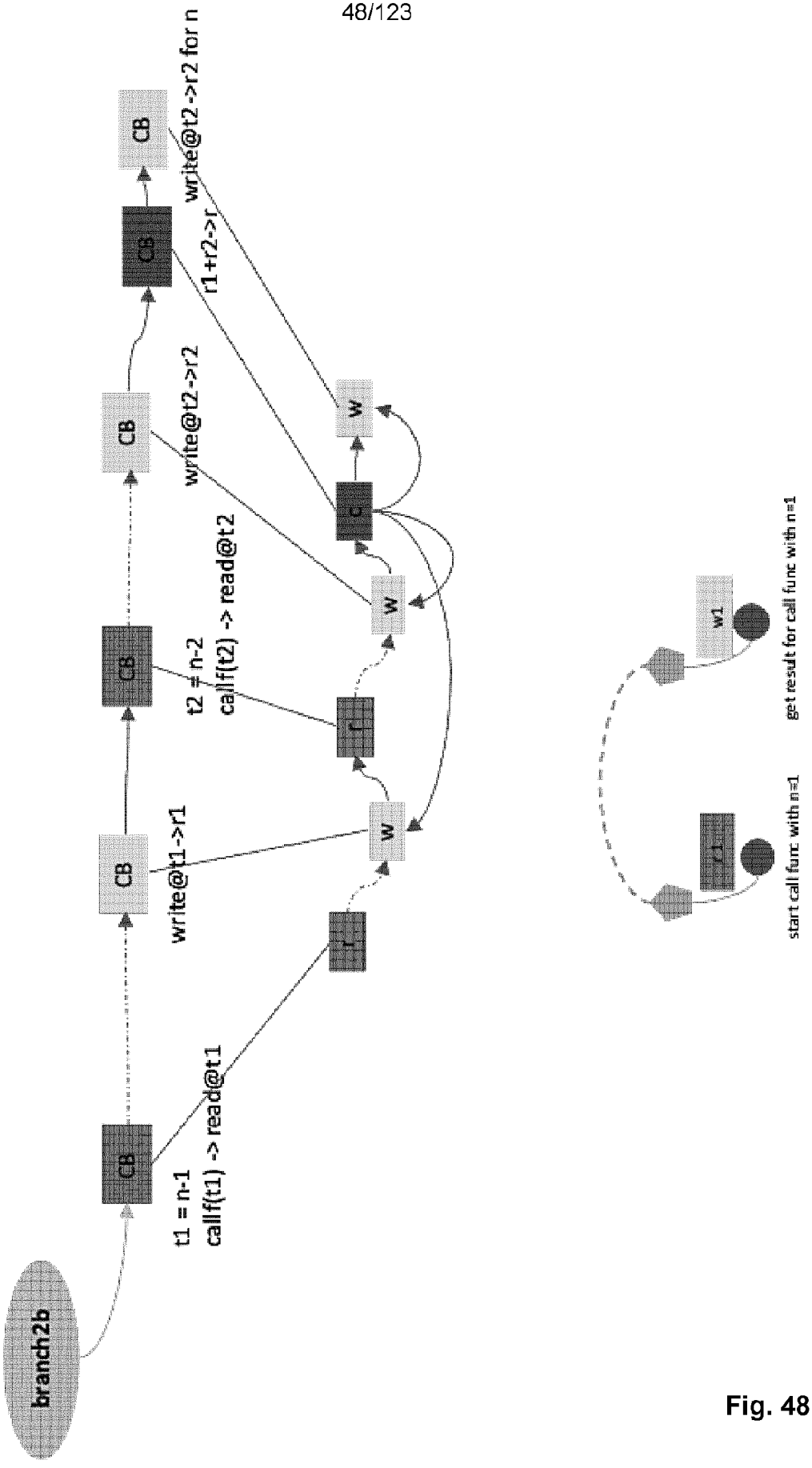

branch2b
CB
CB
CB
CB
CB
CB
t1 = n-1
callf(t1) -> read@t1
write@t1->r1
t2 ≠ n-2
callf(t2) -> read@t2
write@t2->r2
r1+r2->r
write@t2->r2 for n
r
w
w1
start call func with n=1
get result for call func with n=1

Fig. 48

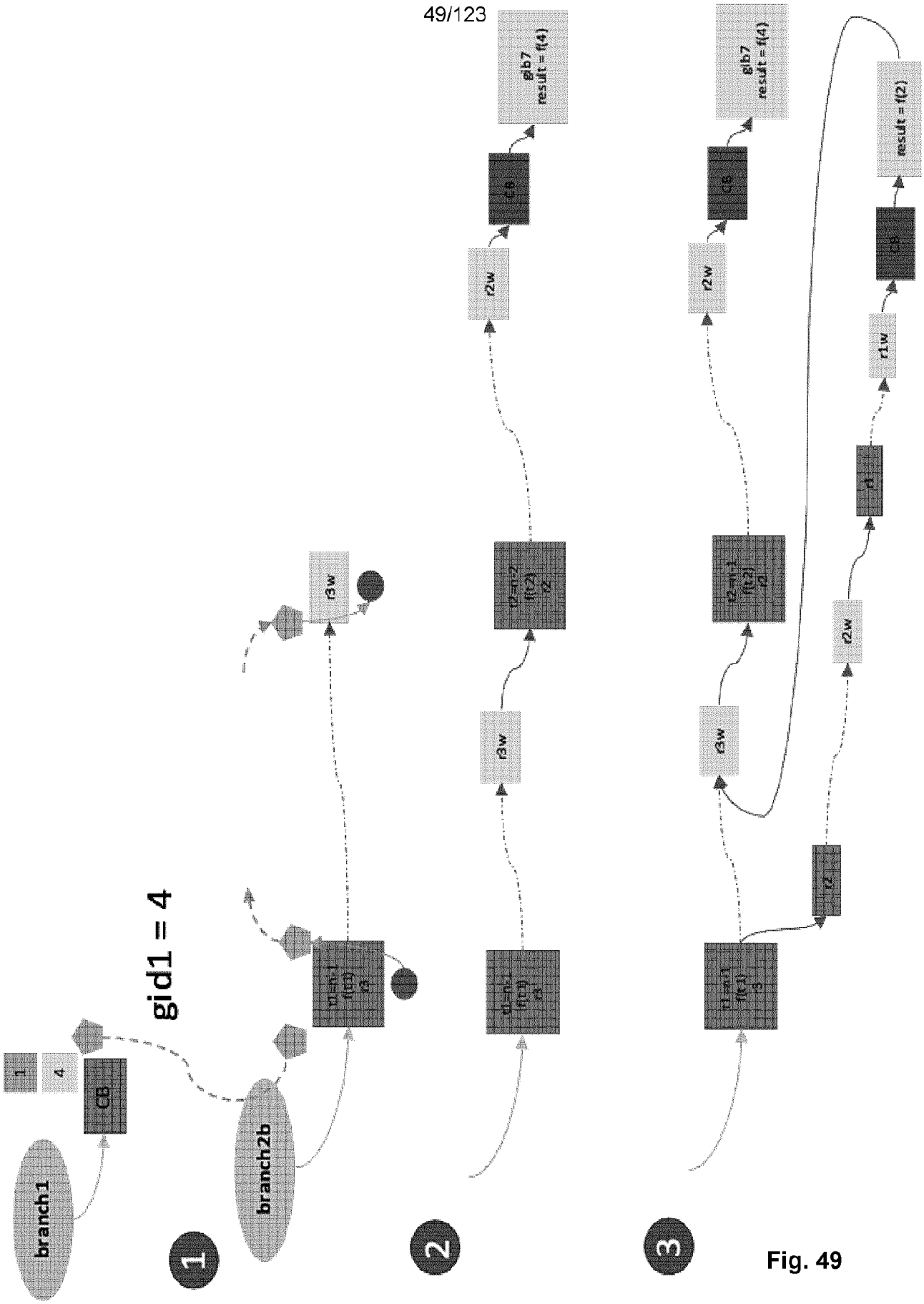

gid1 = 4
branch1
branch2b
1
4
CB
r3.w
r2.w
r1.w
r1
r2
r3
result = f(2)
result = f(4)
1
2
3

Fig. 49

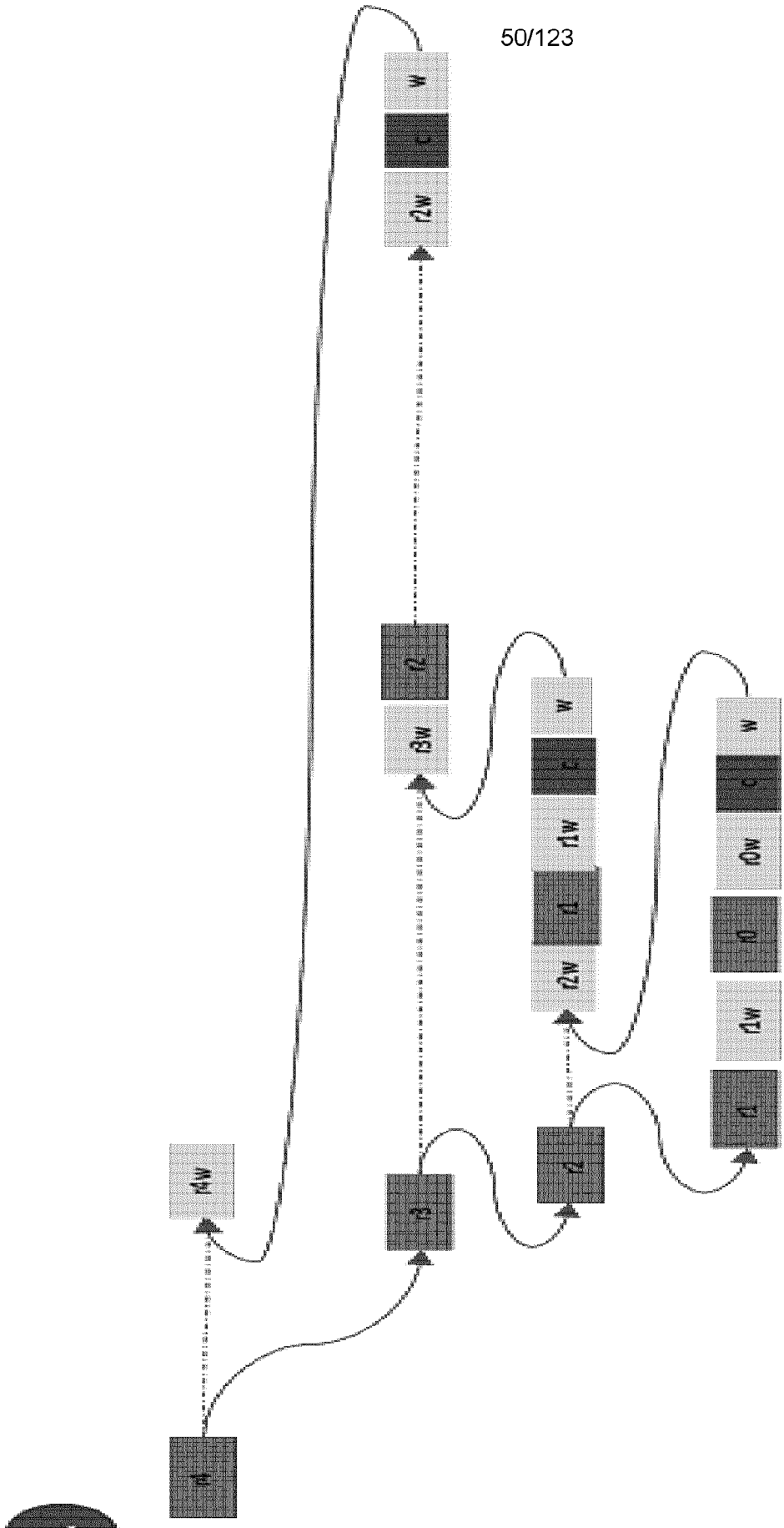


Fig. 50

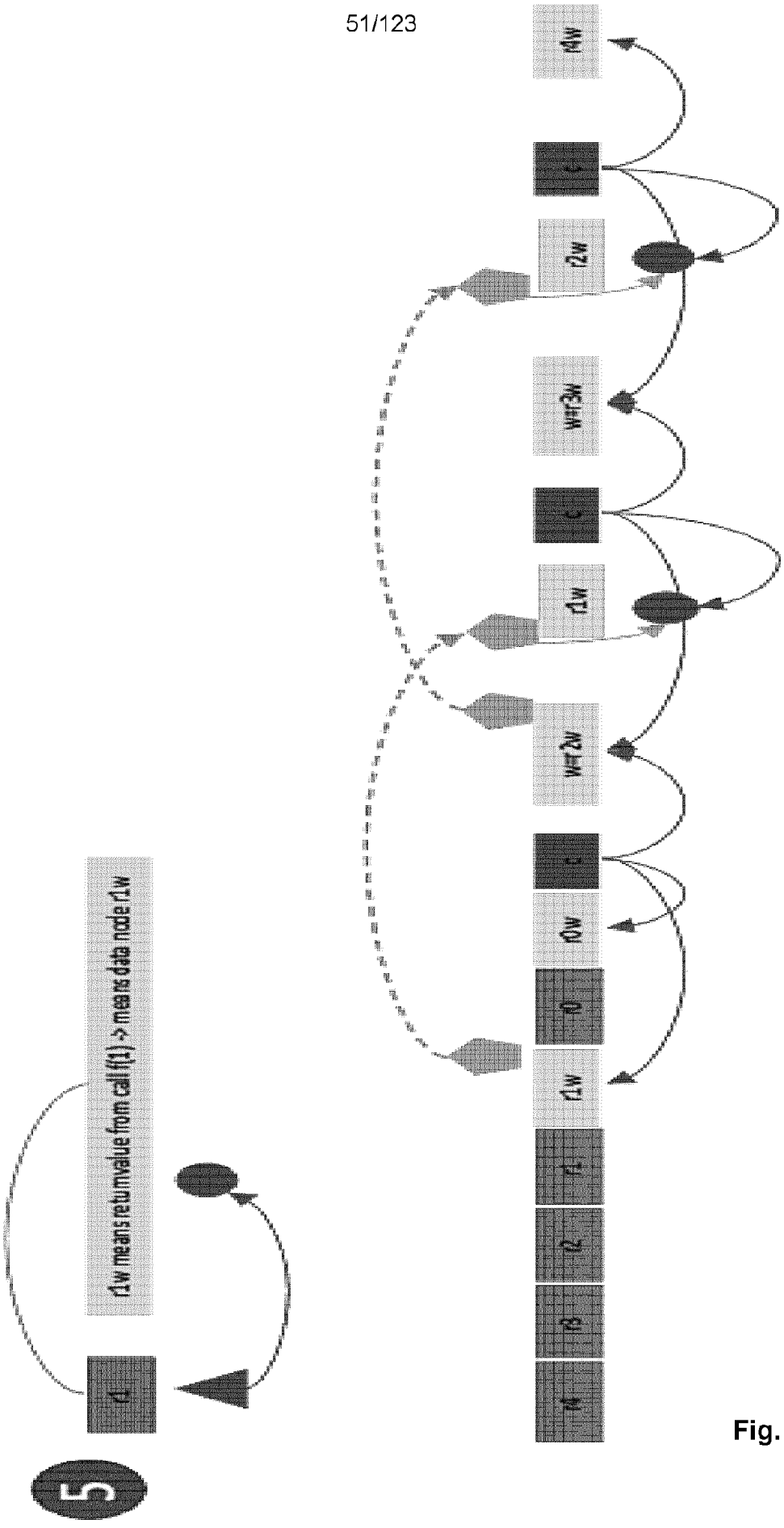

r4w
r2
r2w
w=r3w
r3
r1w
w=r2w
r2
r0w
r0
r1w
r1
r2
r3
r4
r1w means retumvalue from call f(1) -> means data node r1w
r1
5

Fig. 51

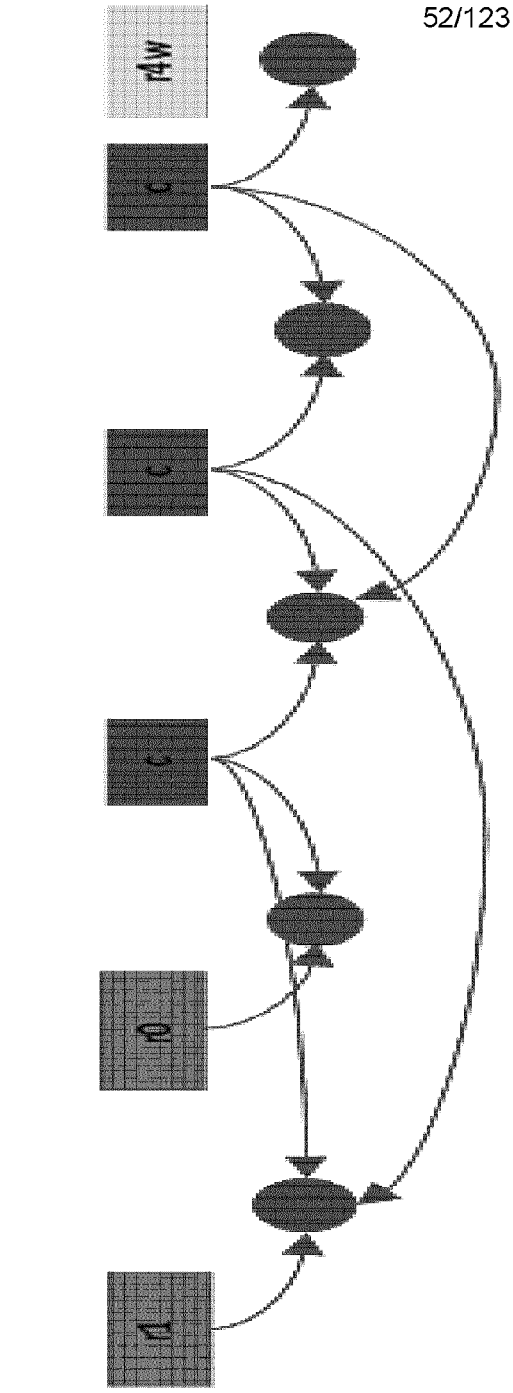


Fig. 52

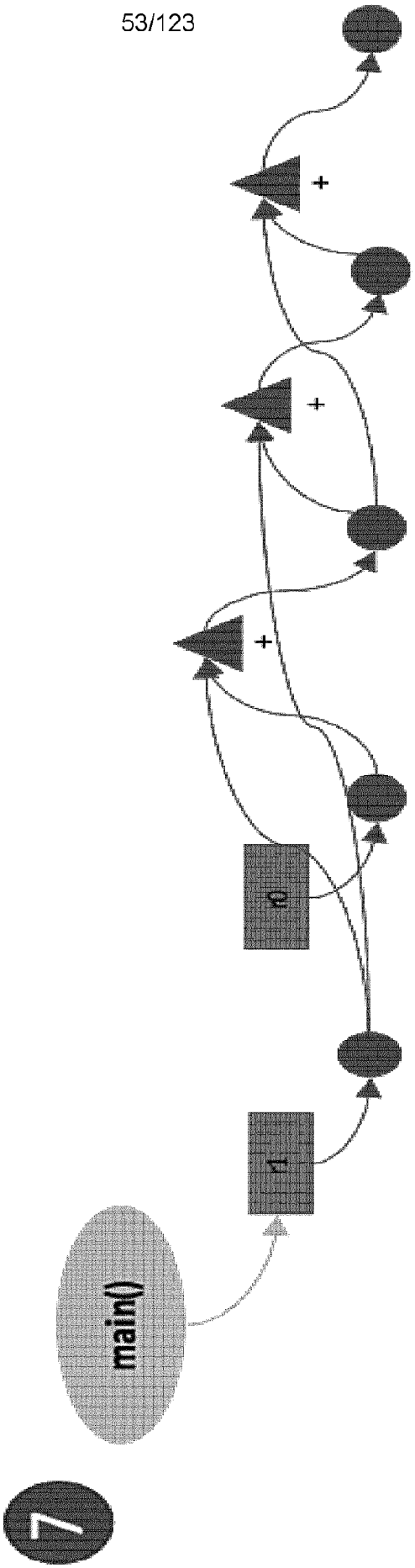


Fig. 53

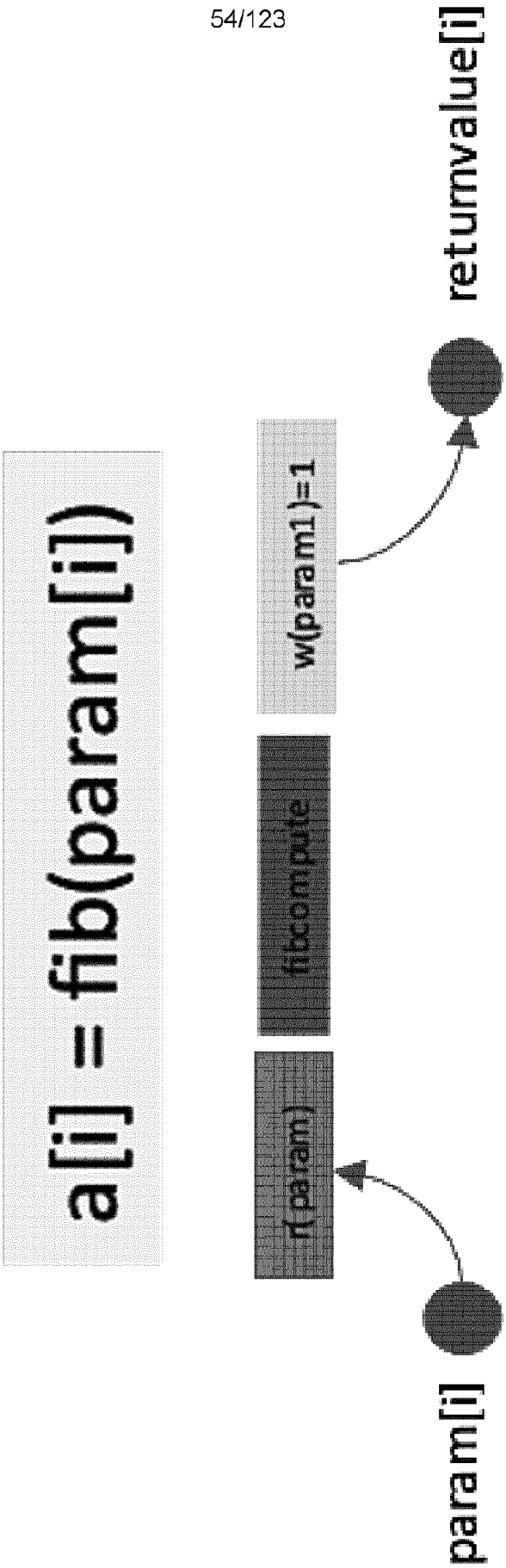

a[i] = fib(param[i])
w(param1)=1
fibcompute
rf(param)
param[i]
returnvalue[i]

Fig. 54

initial block

```
# Set the initial condition
u.fill(u_initial)
```

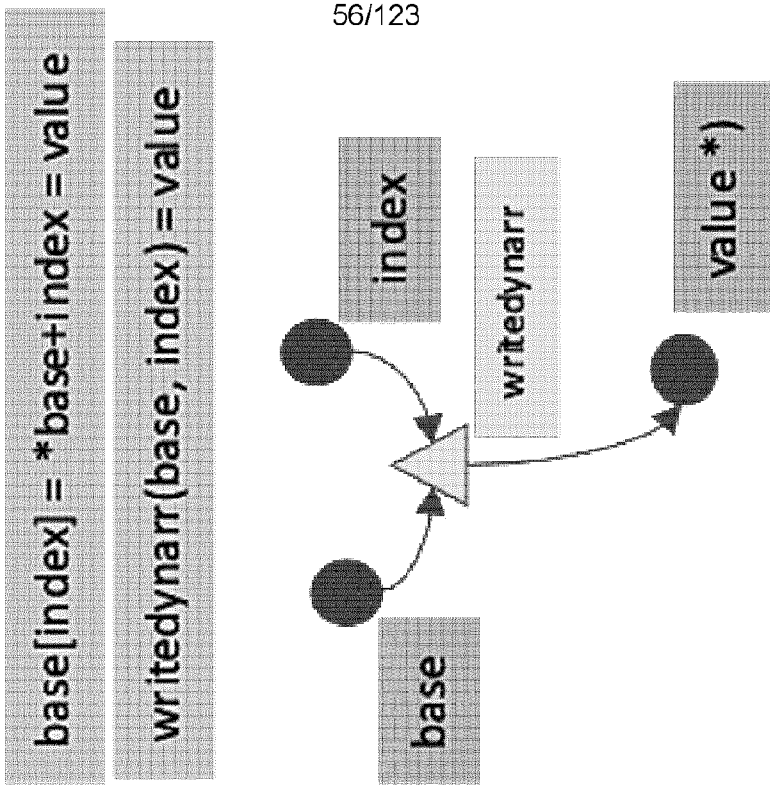
boundary conditions

```
# Set the boundary conditions
u[:, (plate_length-1):, :] = u_top
u[:, :, :1] = u_left
u[:, :1, 1:] = u_bottom
u[:, :, (plate_length-1):] = u_right
```

computation block

```
for k in range(0, max_iter_time-1, 1):
    for i in range(1, plate_length-1, delta_x):
        for j in range(1, plate_length-1, delta_x):
            u[k + 1, i, j] = gamma * (u[k][i+1][j] + u[k][i-1][j] + u[k][i][j+1] + u[k][i][j-1] - 4*u[k][i][j]) + u[k][i][j]
```

Fig. 55


base[index] = *base+index = value
writedynarr(base, index) = value
index
writedynarr
value *)
base

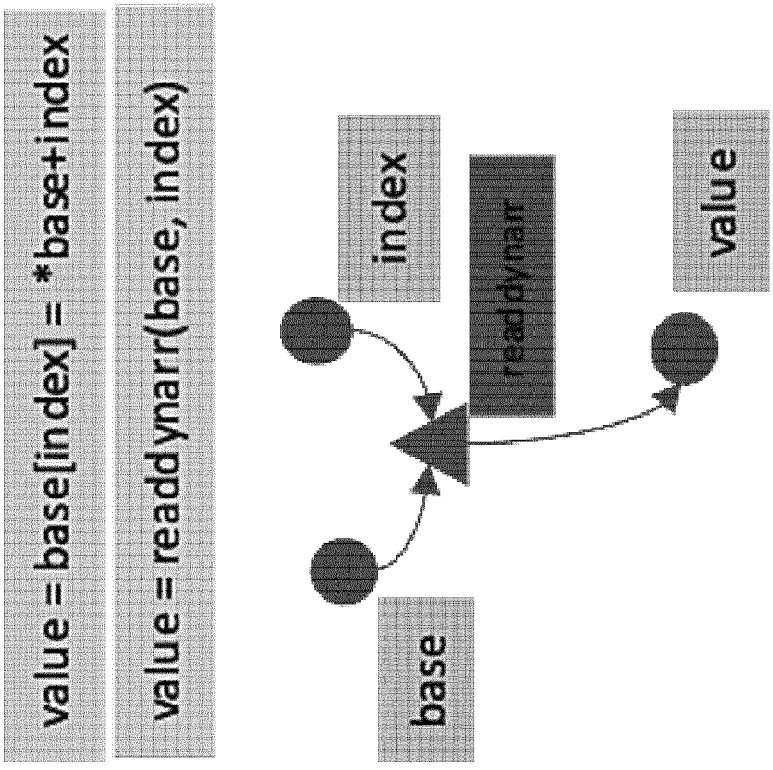

value = base[index] = *base+index
value = readdynarr(base, index)
index
readdynarr
value
base

Fig. 56

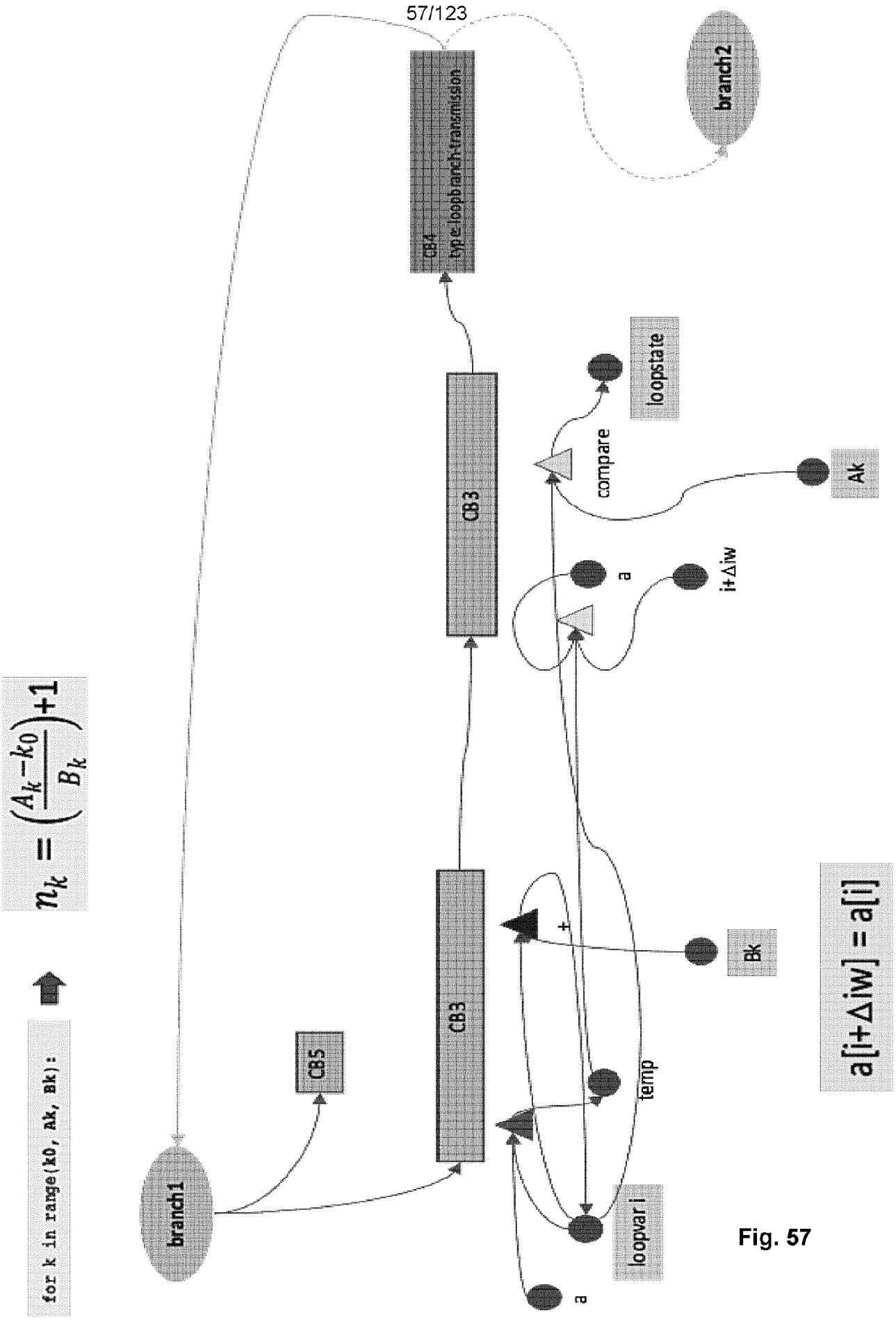

for k in range(k0, Ak, Bk):
$n_k = \left(\frac{A_k - k_0}{B_k}\right) + 1$
branch1
CB5
CB3
CB3
CB4
type=loopbranch-transmission
branch2
loopstate
compare
Ak
a
i+Δiw
Bk
+
temp
loopvar i
a
a[i+Δiw] = a[i]

Fig. 57

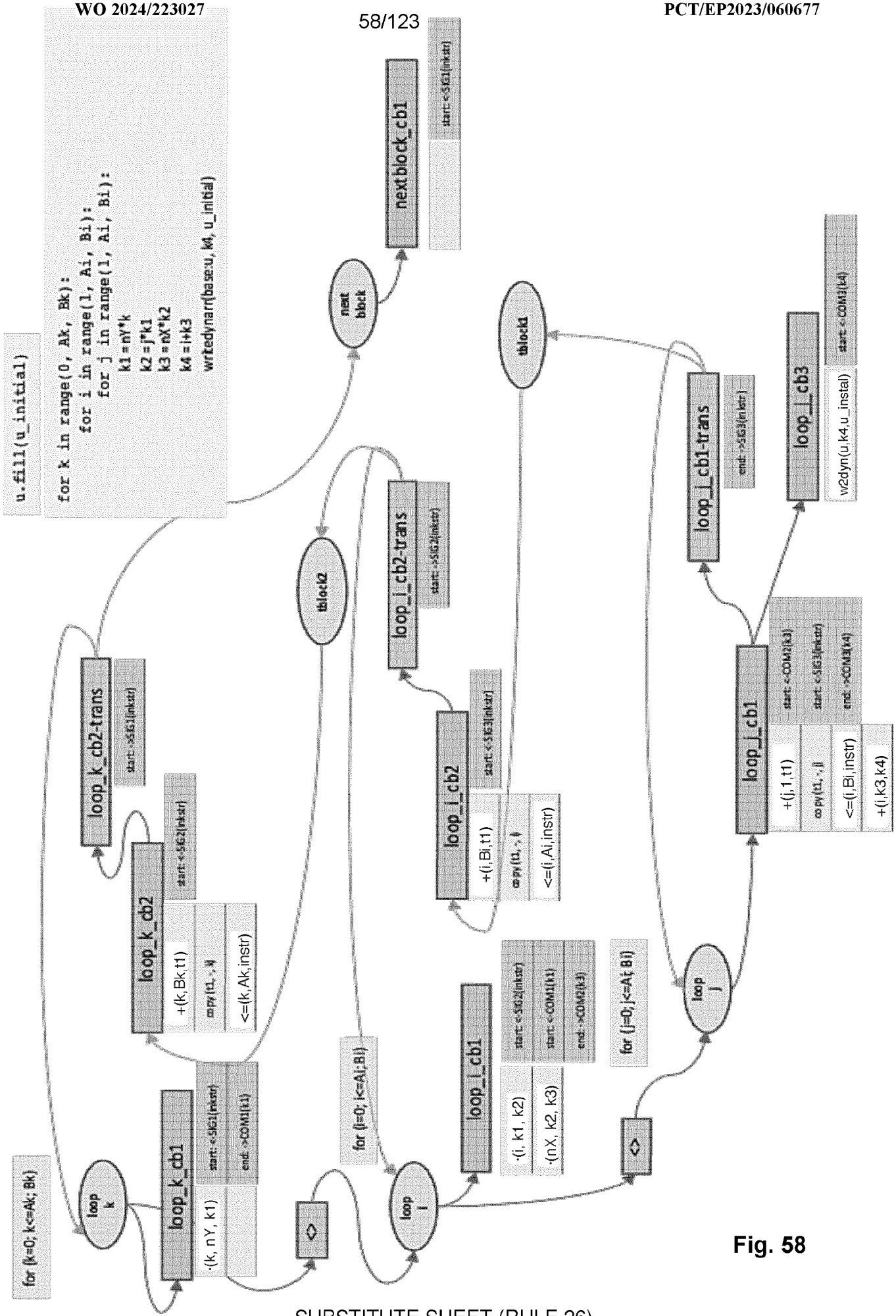

u.fill(u_initial)
for k in range(0, Ak, Bk):
for i in range(1, Ai, Bi):
for j in range(1, Aj, Bj):
k1 = nY*k
k2 = j*k1
k3 = nX*k2
k4 = i+k3
wrtedynarr(base:u, k4, u_initial)
next block_cb1
start: ->SG1(instr)
next bbck
tblock1
tblock2
loop_k_cb2-trans
start: ->SG1(instr)
loop_k_cb2
start: ->SG2(instr)
+(k, Bk, t1)
copy (t1, *, k)
<=(k, Ak, instr)
loop_k_cb1
start: ->SG1(instr)
end: ->COM1(t1)
·(k, nY, k1)
loop k
for (k=0; k<=Ak; Bk)
loop_i_cb2-trans
start: ->SG2(instr)
loop_i_cb2
start: ->SG3(instr)
+(i, Bi, t1)
copy (t1, *, j)
<=(i, Ai, instr)
loop_i_cb1
start: ->SG2(instr)
start: ->COM1(k1)
end: ->COM2(k3)
·(i, k1, k2)
·(nX, k2, k3)
loop i
for (i=0; i<=Ai; Bi)
loop_j_cb1-trans
end: ->SG3(instr)
loop_j_cb1
start: ->COM2(k3)
start: ->SG3(instr)
end: ->COM3(k4)
+(j, 1, t1)
copy (t1, *, j)
<=(i, Bi, instr)
+(i, k3, k4)
loop j
for (j=0; j<=Aj; Bj)
loop_j_cb3
start: ->COM3(k4)
w2dyn(u, k4, u_instal)

Fig. 58

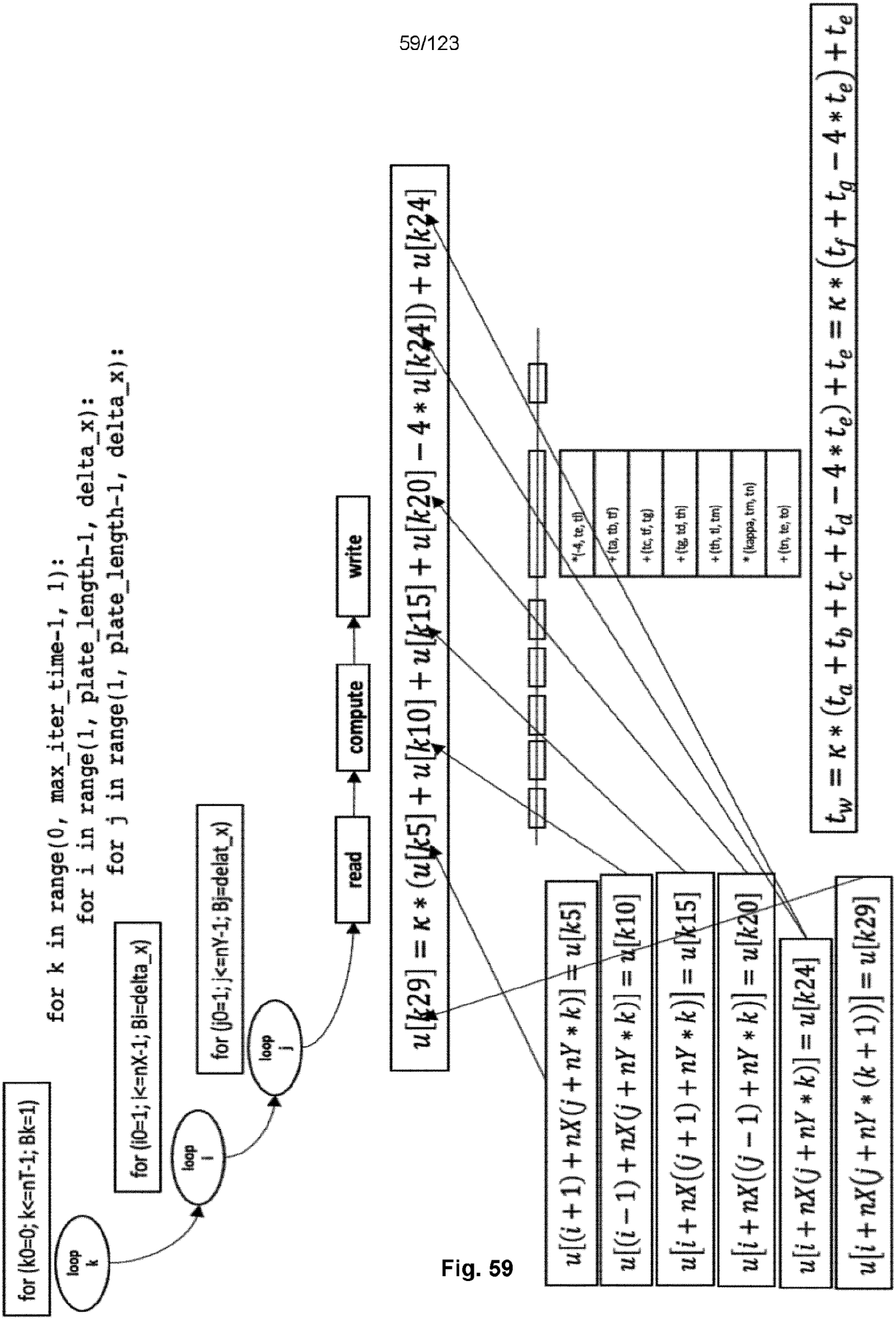

for (k0=0; k<=nT-1; Bk=1)
loop k
for (i0=1; i<=nX-1; Bi=delta_x)
loop i
for (j0=1; j<=nY-1; Bj=delat_x)
loop j
read
compute
write
for k in range(0, max_iter_time-1, 1):
for i in range(1, plate_length-1, delta_x):
for j in range(1, plate_length-1, delta_x):
$u[k29] = \kappa * (u[k5] + u[k10] + u[k15] + u[k20] - 4 * u[k24]) + u[k24]$
$u[(i+1) + nX(j+nY*k)] = u[k5]$
$u[(i-1) + nX(j+nY*k)] = u[k10]$
$u[i + nX((j+1) + nY*k)] = u[k15]$
$u[i + nX((j-1) + nY*k)] = u[k20]$
$u[i + nX(j+nY*k)] = u[k24]$
$u[i + nX(j+nY*(k+1))] = u[k29]$
*(-4, te, tl)
+ (ta, tb, tf)
+ (tc, tf, tg)
+ (tg, td, th)
+ (th, tl, tm)
* (kappa, tm, tn)
+ (tn, te, to)
$t_w = \kappa * (t_a + t_b + t_c + t_d - 4 * t_e) + t_e = \kappa * (t_f + t_g - 4 * t_e) + t_e$

Fig. 59

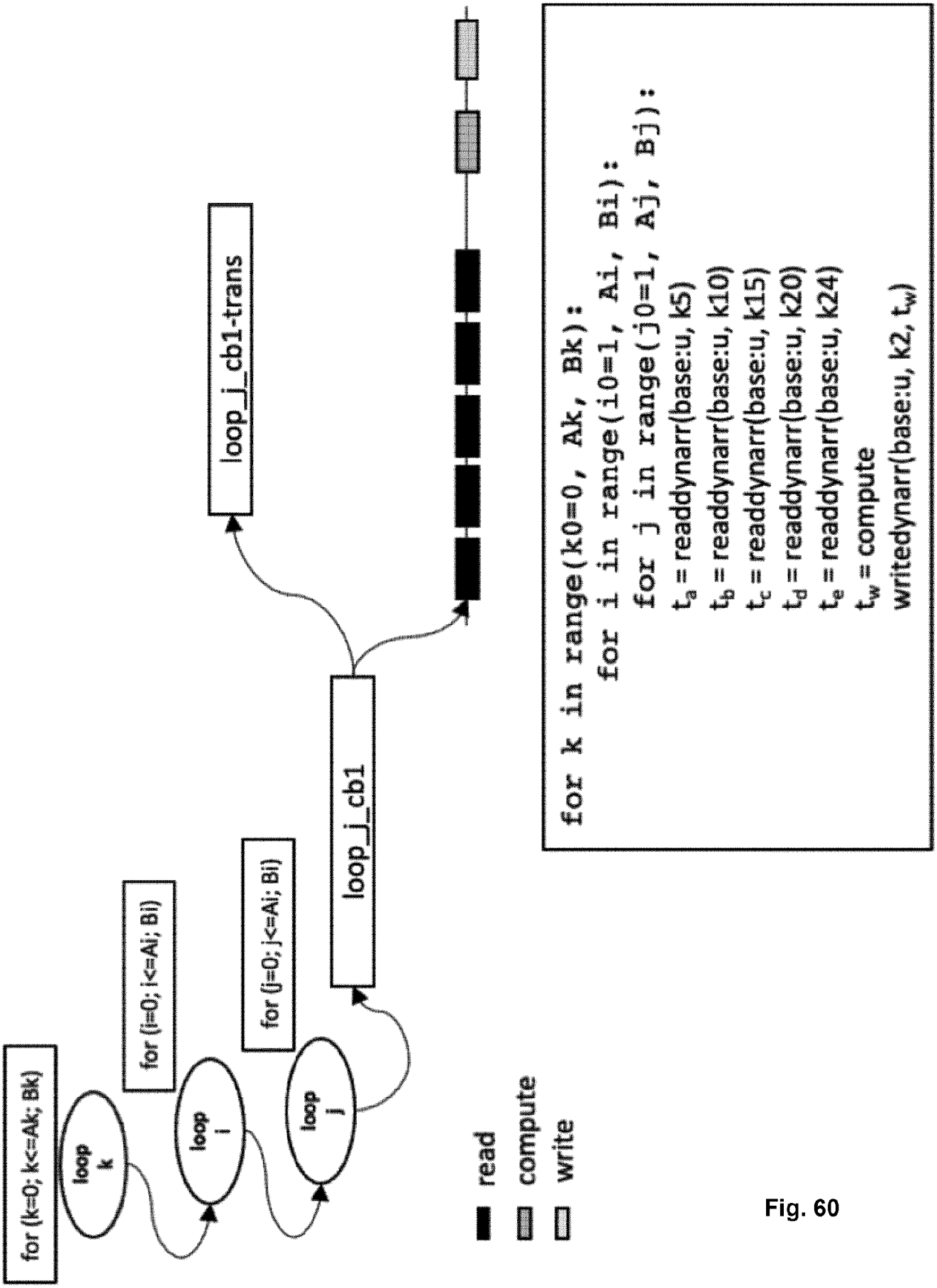

for (k=0; k<=Ak; Bk)
loop k
for (i=0; i<=Ai; Bi)
loop i
for (j=0; j<=Aj; Bj)
loop j
loop_j_cb1
loop_j_cb1-trans
read
compute
write
for k in range(k0=0, Ak, Bk):
for i in range(i0=1, Ai, Bi):
for j in range(j0=1, Aj, Bj):
t_a = readdynarr(base:u, k5)
t_b = readdynarr(base:u, k10)
t_c = readdynarr(base:u, k15)
t_d = readdynarr(base:u, k20)
t_e = readdynarr(base:u, k24)
t_w = compute
writedynarr(base:u, k2, t_w)

Fig. 60

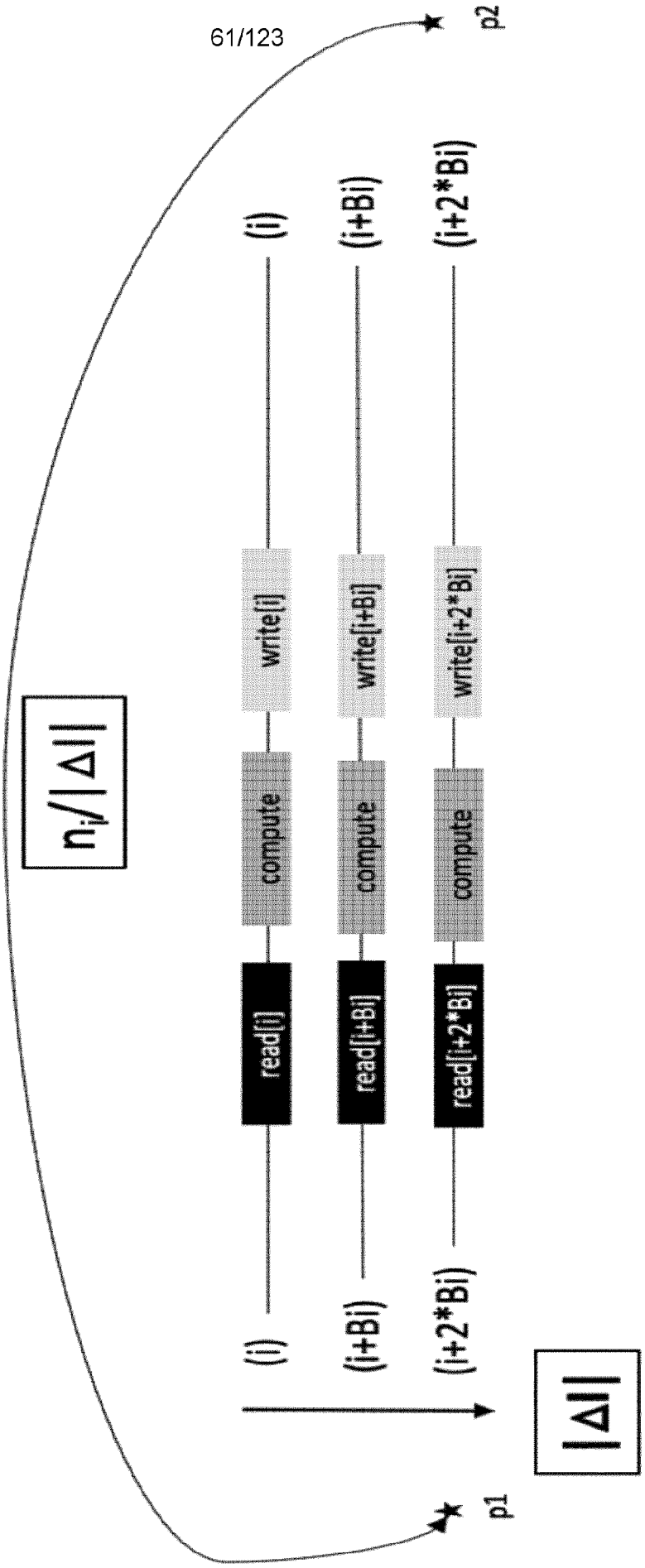
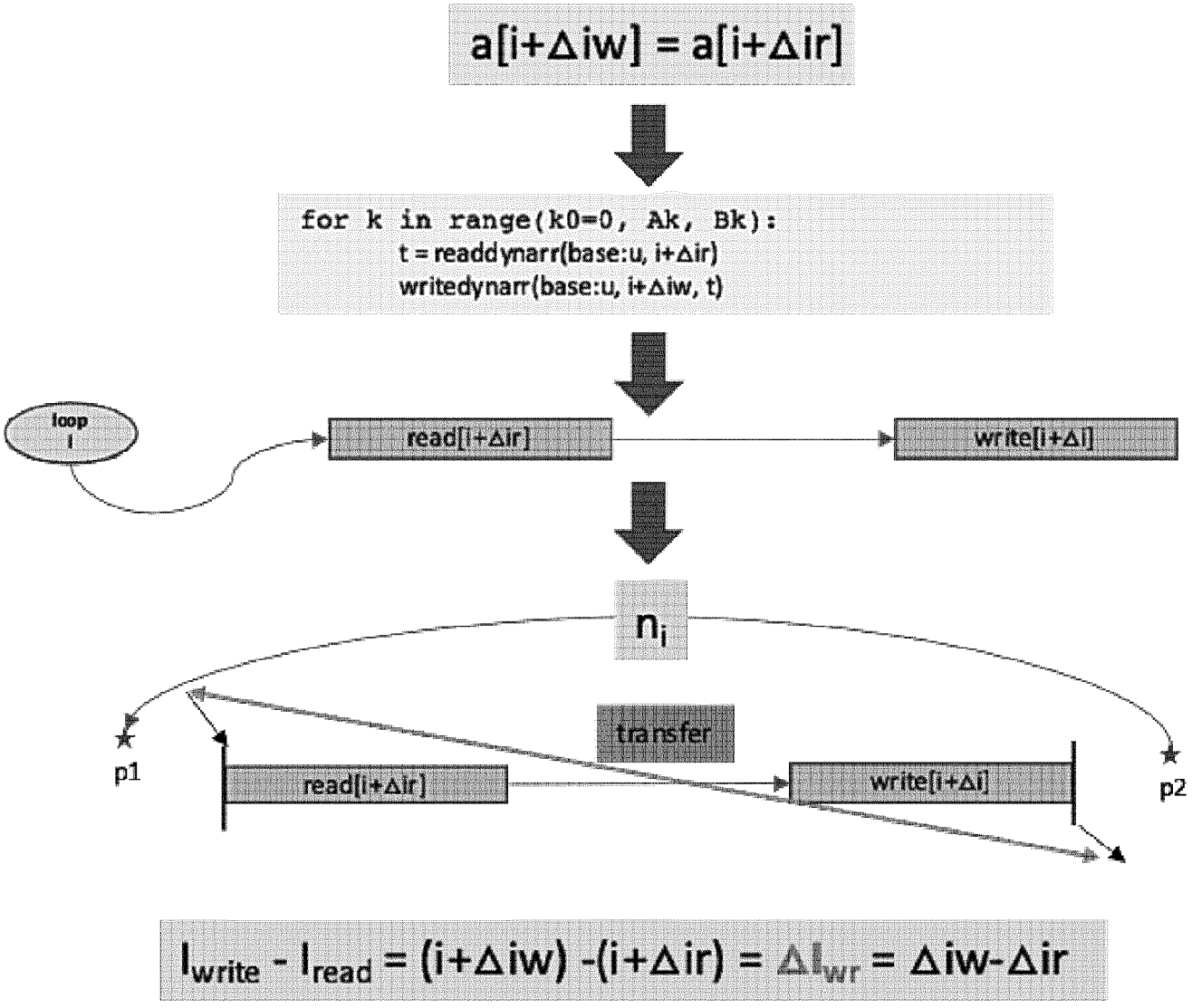

p2
(i)
(i+Bi)
(i+2*Bi)
write[i]
write[i+Bi]
write[i+2*Bi]
compute
compute
compute
read[i]
read[i+Bi]
read[i+2*Bi]
(i)
(i+Bi)
(i+2*Bi)
p1
|ΔI|
ni/|ΔI|

Fig. 61



$$I_{write} - I_{read} = (i+\Delta iw) - (i+\Delta ir) = \Delta I_{wr} = \Delta iw - \Delta ir$$

Fig. 62

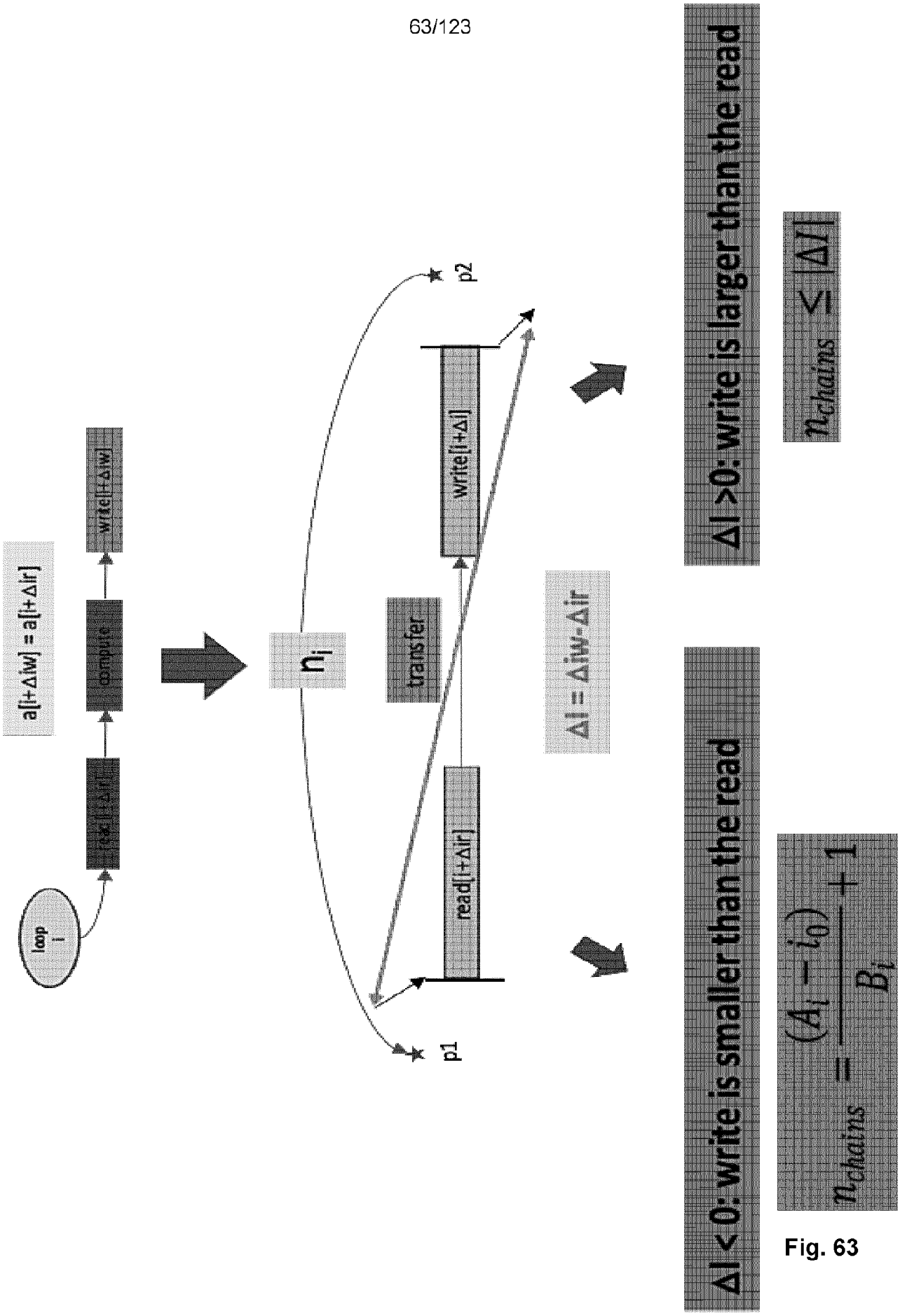

loop i
read[i+Δir]
compute
write[i+Δiw]
a[i+Δiw] = a[i+Δir]
p1
p2
read[i+Δir]
transfer
write[i+Δi]
Δi = Δiw-Δir
Δi <0: write is smaller than the read
Δi >0: write is larger than the read
$\frac{(A_i - i_0)}{n_{chains}} = \frac{B_i}{n_{chains}} + 1$
$n_{chains} \leq |\Delta i|$

Fig. 63

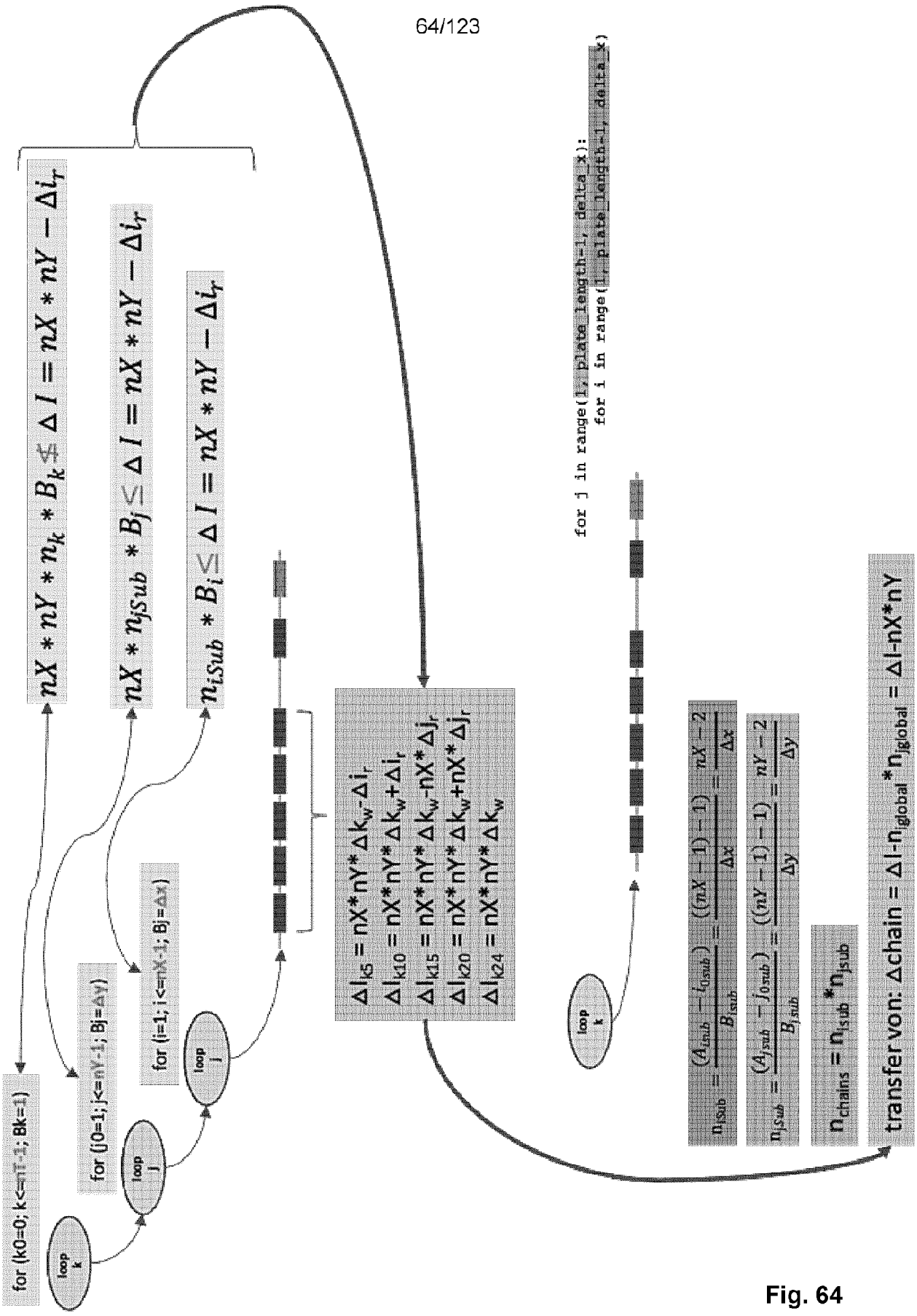

for (k0=0; k<=nT-1; Bk=1)
loop k
for (j0=1; j<=nY-1; Bj=ΔY)
loop j
for (i=1; i <=nX-1; Bj=Δx)
loop j
loop k
for j in range(1, plate_length-1, delta_x):
for i in range(1, plate_length-1, delta_x)
transfer von: Δchain = Δl-nglobal * nJglobal = Δl-nX*nY
nchains = niSub *njSub

Fig. 64

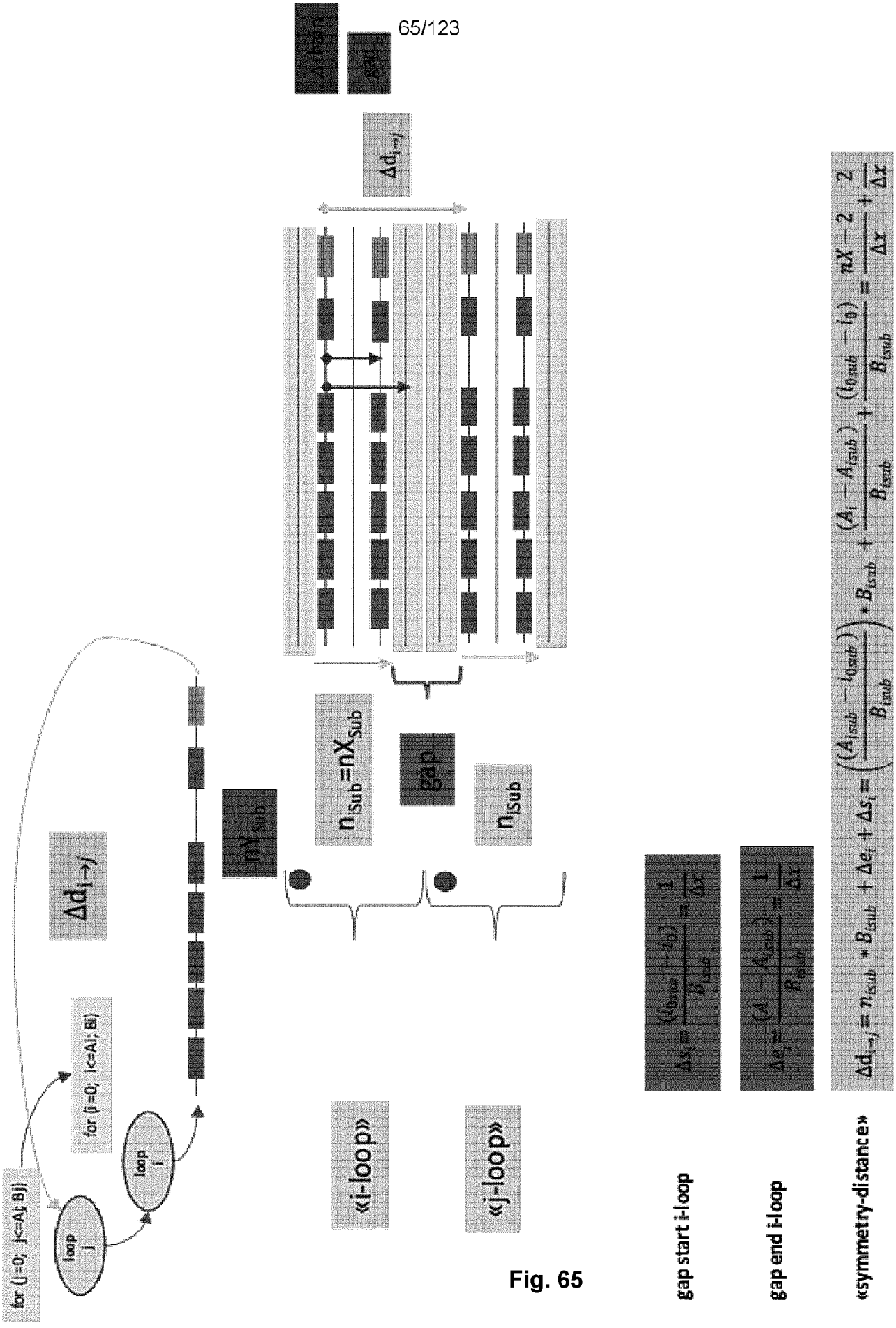

for (j=0; j<=Aj; Bj)
loop j
for (i=0; i<=Ai; Bi)
loop i
Δd_{i→j}
nY_Sub
n_iSub=nX_Sub
gap
n_iSub
«i-loop»
«j-loop»
Δd_{i→j}
Δ chain
Δ x
gap start i-loop
gap end i-loop
«symmetry-distance»

Fig. 65

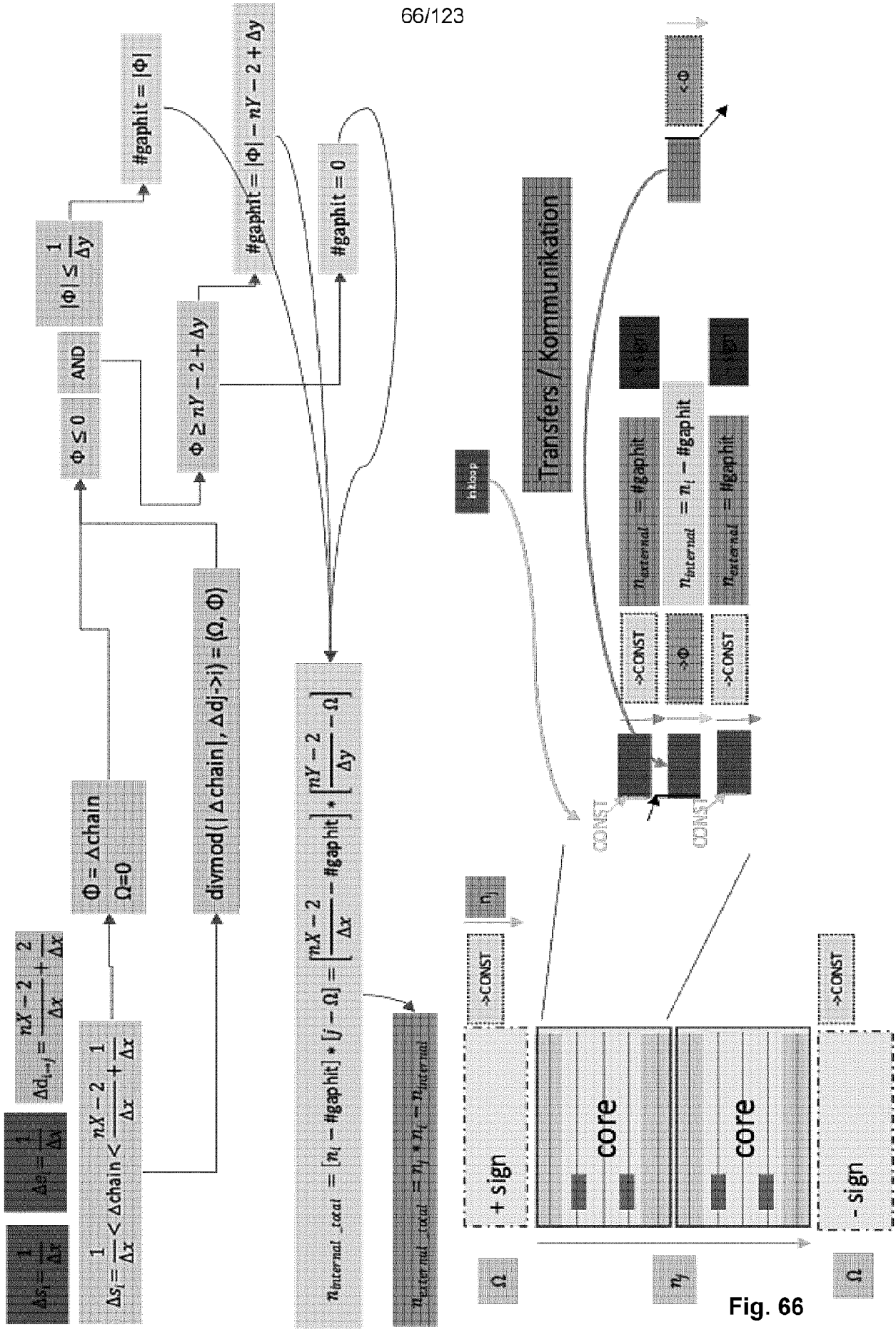

Transfers / Kommunikation
core
core
+ sign
- sign
CONST
CONST
AND
#gaphit = |Φ|
#gaphit = |Φ| − nY − 2 + Δy
#gaphit = 0
Φ = Δchain
Ω=0
divmod(|Δchain|, Δdj->i) = (Ω, Φ)
Φ ≤ 0
Φ ≥ nY − 2 + Δy
|Φ| ≤ 1/Δy
Δs_i = 1/Δx
Δe = 1/Δx
Δd_i→j = (nX − 2)/Δx + 2/Δx
1/Δx ≤ Δchain < (nX − 2)/Δx + 1/Δx
n_internal_total = [n_i − #gaphit] * [j − Ω] = ⌊(nX − 2)/Δx − #gaphit⌋ * ⌊(nY − 2)/Δy − Ω⌋
n_external_total = n_j * n_i − n_internal
n_internal = #gaphit
n_external = n_i − #gaphit
n_external = #gaphit
->CONST
->Φ
->CONST
->CONST
<-Φ
n_i
n_j
Ω
Ω

Fig. 66

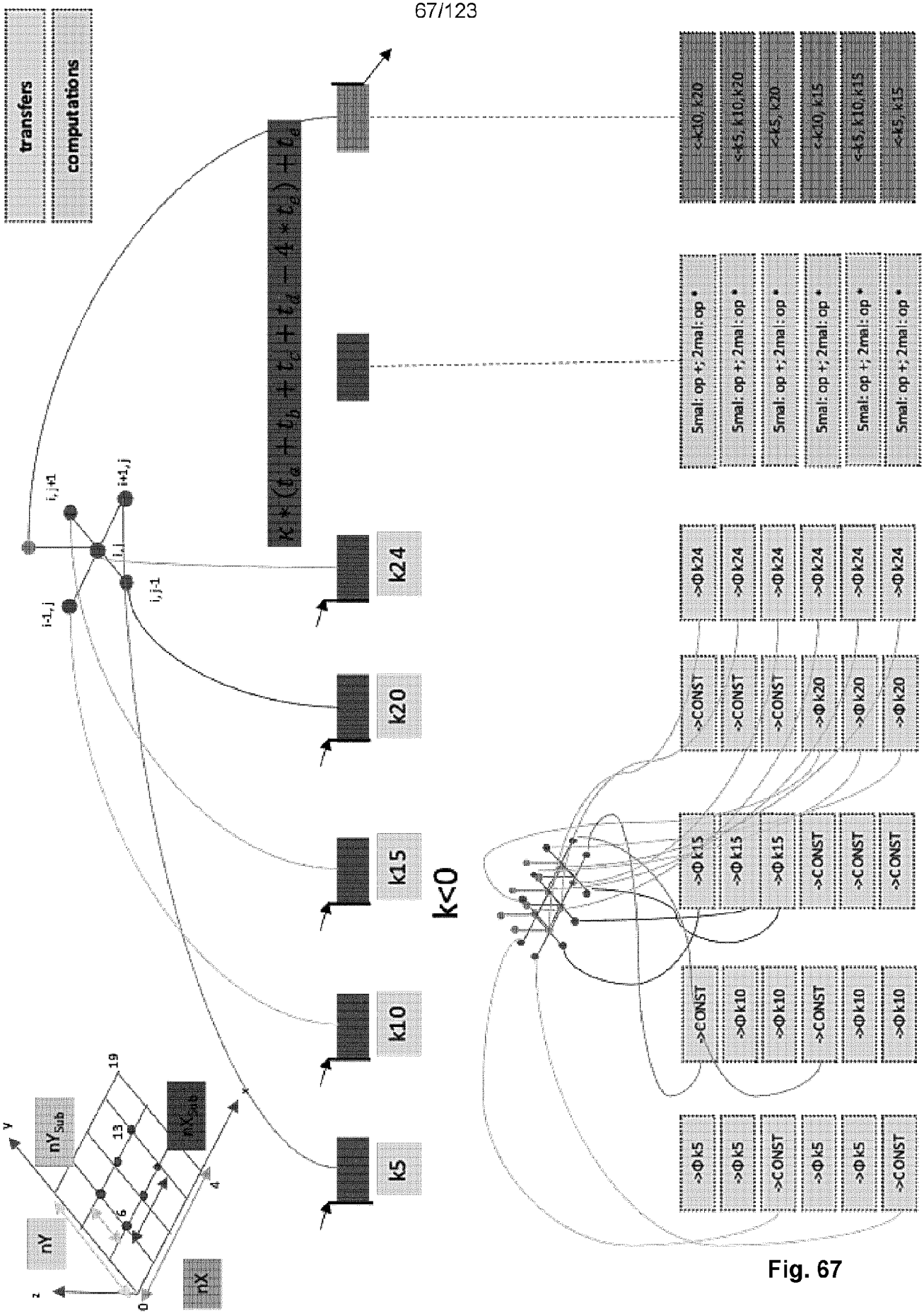

transfers
computations
$\kappa * (t_a + t_b + t_c + t_d + 4 * t_e) + t_a$
k24
k20
k15
k10
k5
k<0
i-1, j
i, j+1
i+1, j
i, j-1
nY
nY_Sub
nX
nK_Sub
19
13
6
4
0
z
y
x
5mal: op +; 2mal: op *
<k10, k20
<k5, k10,k20
<k5, k20
<k10, k15
<k5, k10,k15
<k5, k15
->Φ k24
->Φ k20
->Φ k15
->Φ k10
->Φ k5
->CONST

Fig. 67

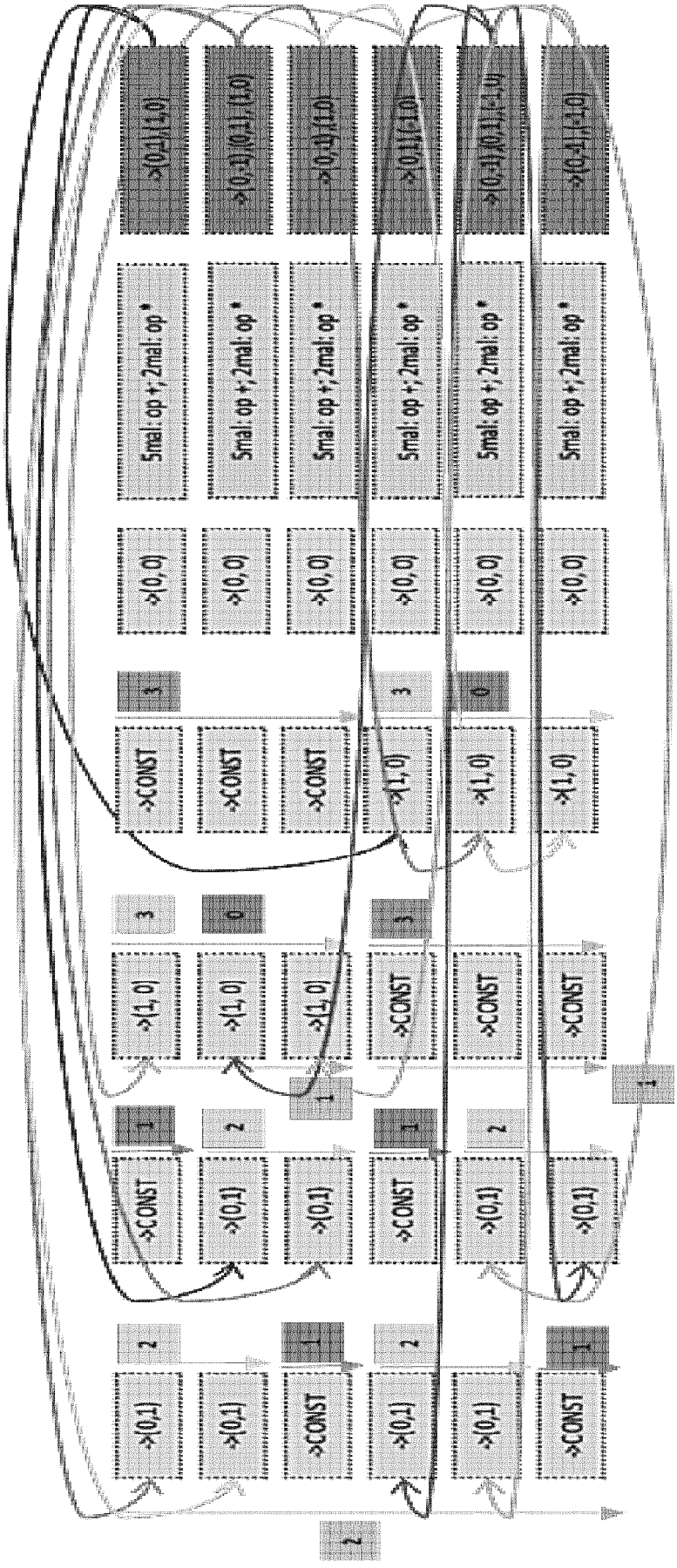


Fig. 68

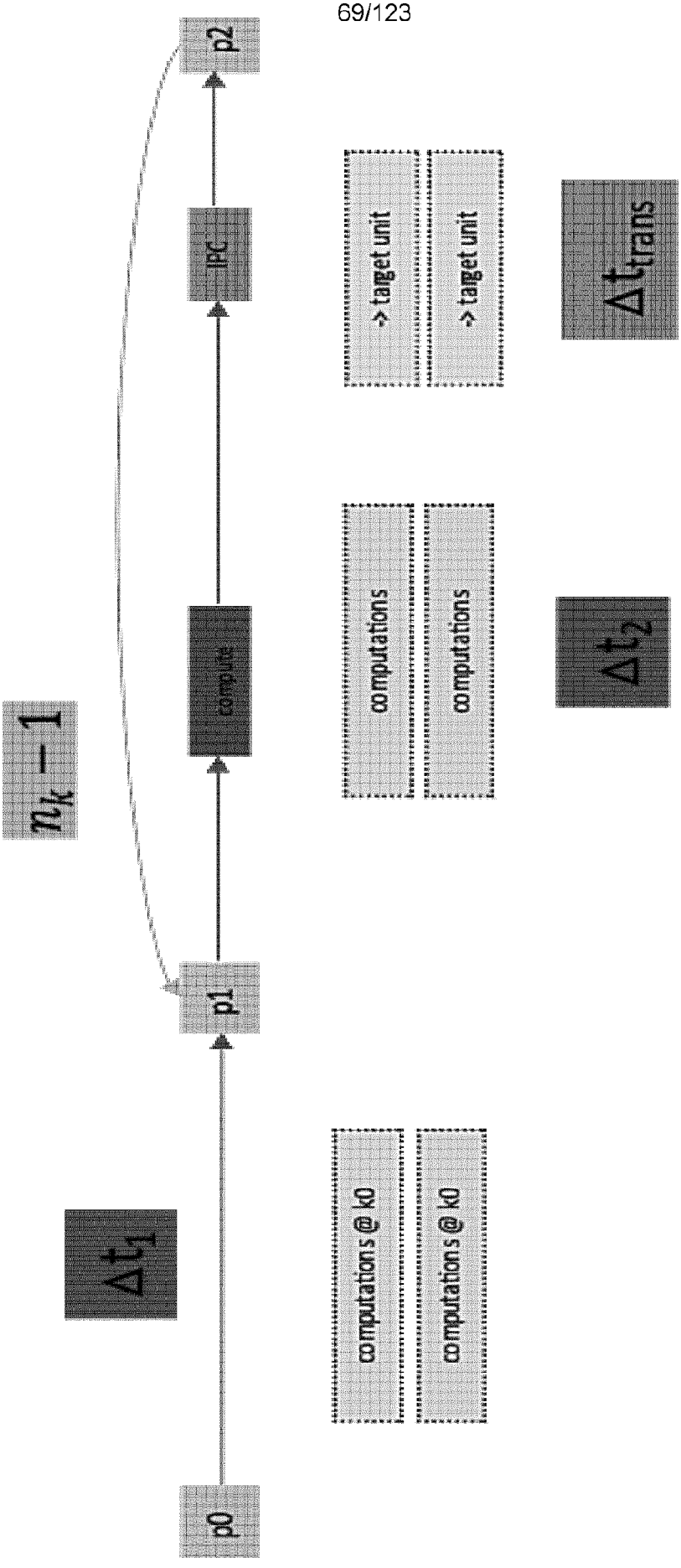

p2
IPC
compute
p1
p0
$n_k - 1$
Δt_1
Δt_2
Δt_{trans}
computations @ k0
computations @ k0
computations
computations
-> target unit
-> target unit

Fig. 69

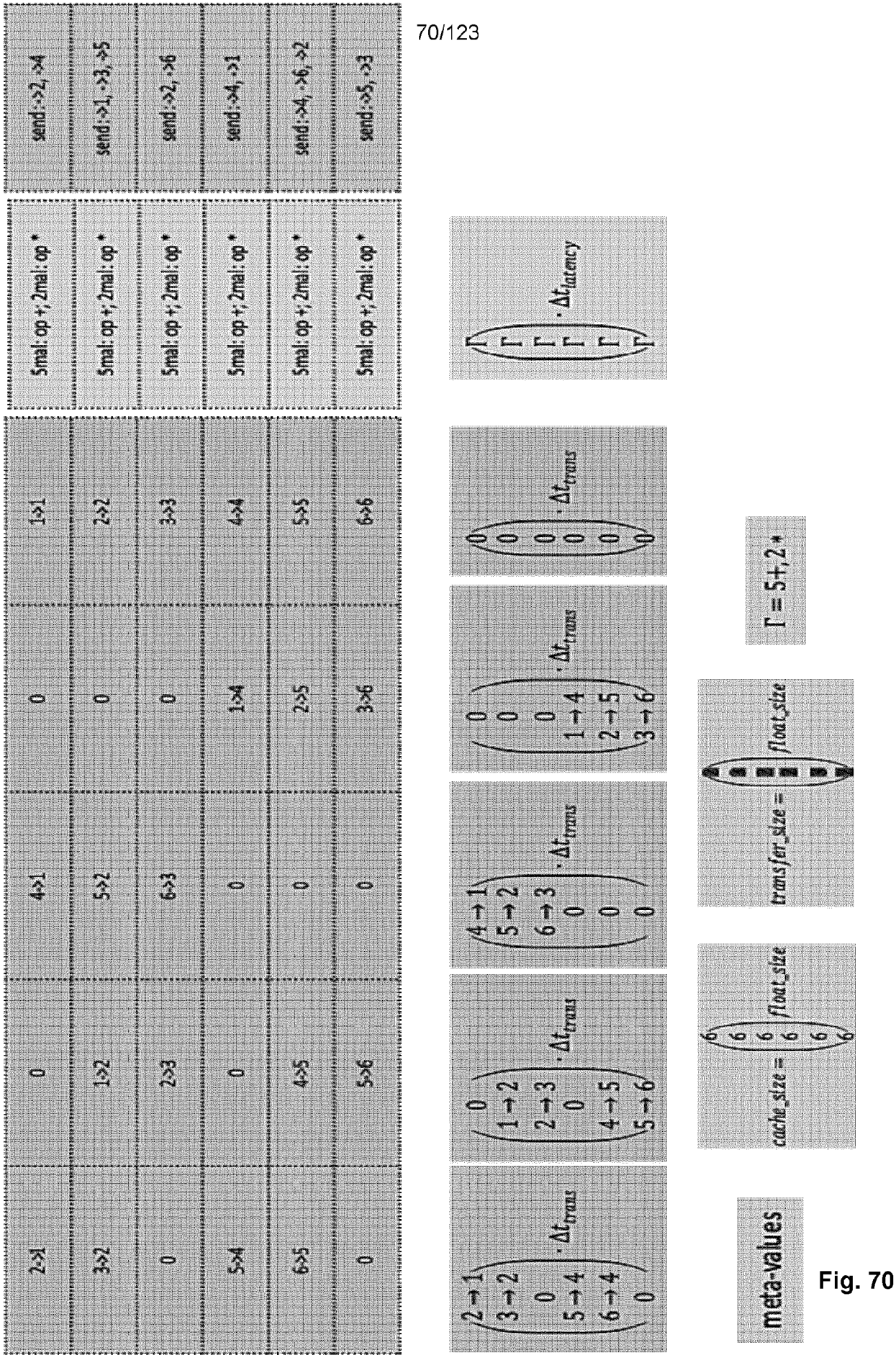

meta-values
cache_size = (6 6 6 6 6 6) float_size
transfer_size = (1 1 1 1 1 1) float_size
Γ = 5+, 2*
2->1 3->2 0 6->5
0 1->2 2->3 4->5
4->1 5->2 6->3 0
0 1->4 2->5 3->6
1->1 2->2 3->3 4->4 5->5 6->6
5mal: op +; 2mal: op *
send:->2, ->4
send:->1, ->3, ->5
send:->2, ->6
send:->4, ->1
send:->4, ->6, ->2
send:->5, ->3

Fig. 70

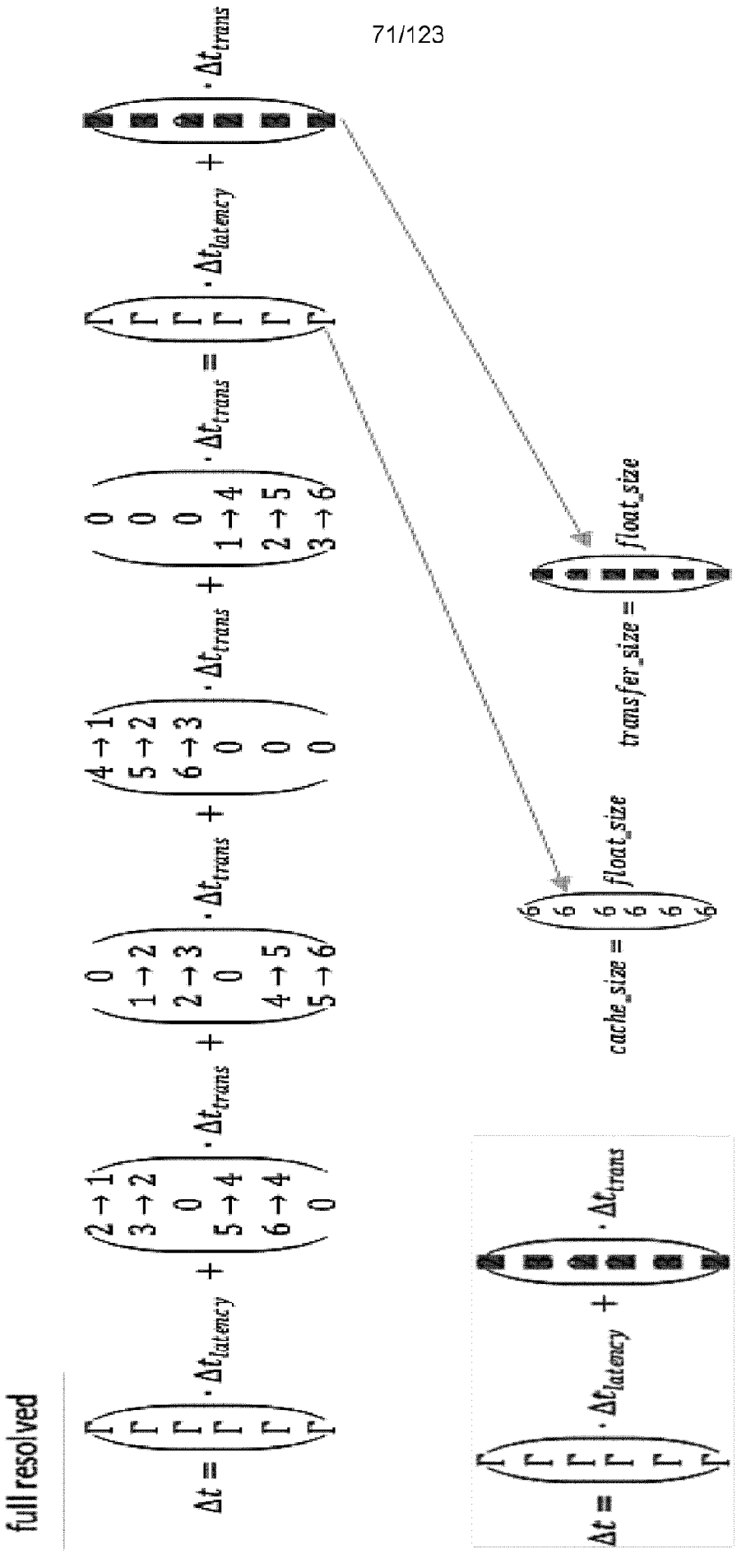
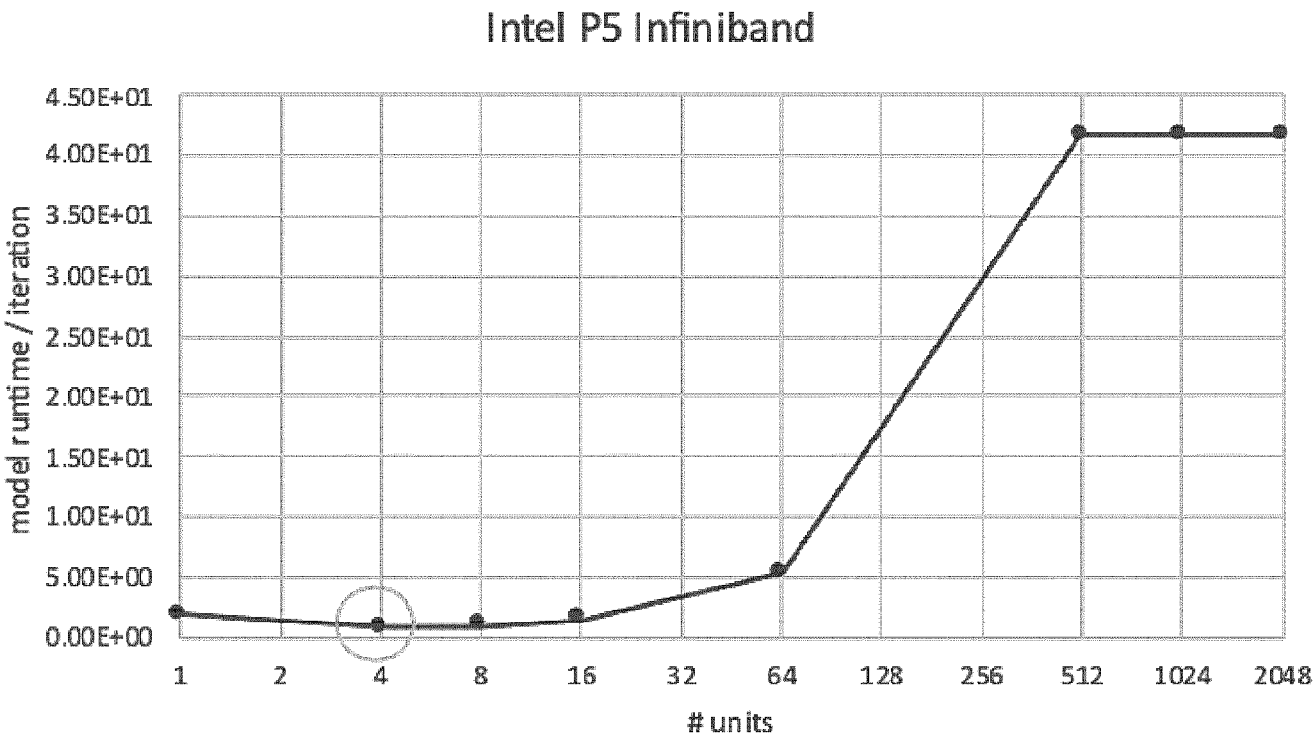

full resolved
$\Delta t = \Gamma \cdot \Delta t_{latency} + \begin{pmatrix} 2\to1 & & \\ 3\to2 & 0 & \\ & 5\to4 & 6\to4 \\ & & 0 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 0 & & \\ 1\to2 & 2\to3 & \\ & 0 & 4\to5 \\ & & 5\to6 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 4\to1 & & \\ 5\to2 & 6\to3 & \\ & 0 & 0 \\ & & 0 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 0 & & \\ & 0 & \\ & & 1\to4 \\ & & 2\to5 \\ & & 3\to6 \end{pmatrix} \cdot \Delta t_{trans} = \Gamma \cdot \Delta t_{latency} + \begin{pmatrix} & & \\ & & \\ & & \\ & & \\ & & \end{pmatrix} \cdot \Delta t_{trans}$
$cache_size = \begin{pmatrix} 6 & 6 & 6 \\ 6 & 6 & 6 \end{pmatrix} float_size$
$transfer_size = \begin{pmatrix} & & \\ & & \end{pmatrix} float_size$
$\Delta t = \Gamma \cdot \Delta t_{latency} + \begin{pmatrix} & & \\ & & \\ & & \\ & & \\ & & \end{pmatrix} \cdot \Delta t_{trans}$

Fig. 71


Intel P5 Infiniband
model runtime / iteration
4.50E+01
4.00E+01
3.50E+01
3.00E+01
2.50E+01
2.00E+01
1.50E+01
1.00E+01
5.00E+00
0.00E+00
1
2
4
8
16
32
64
128
256
512
1024
2048
units

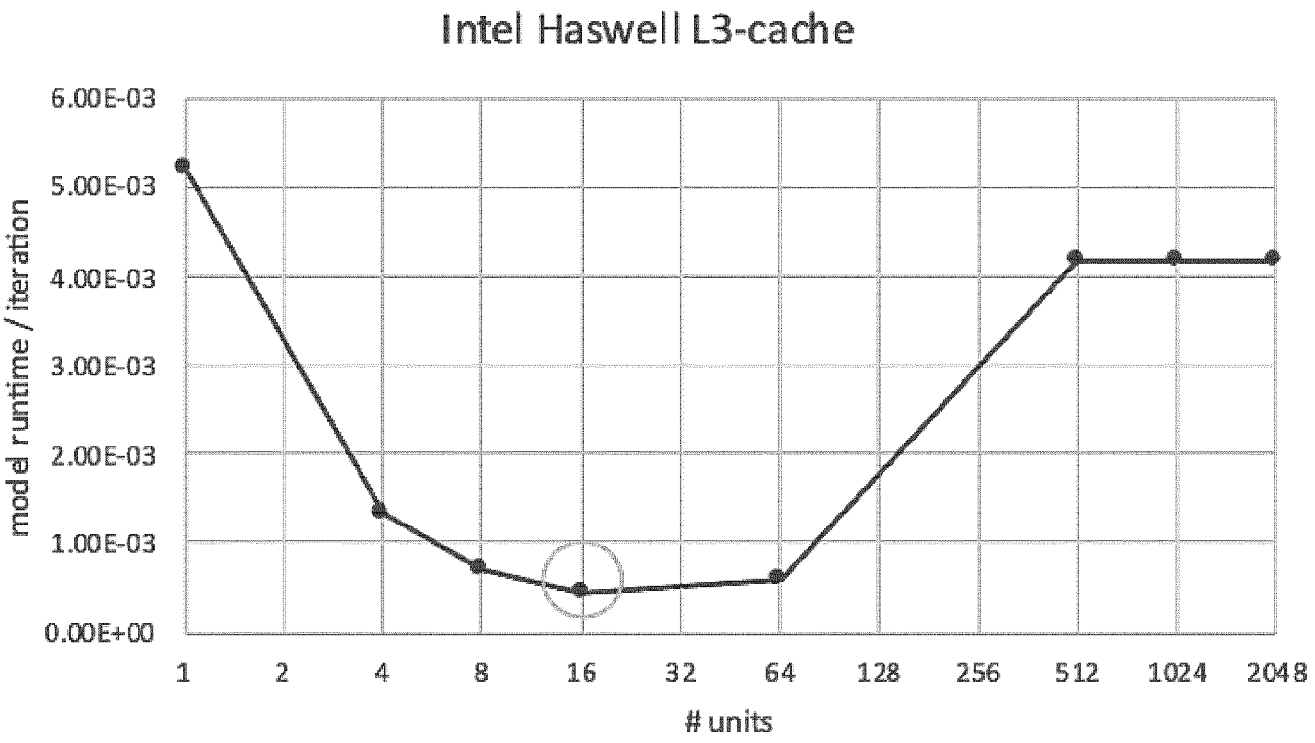
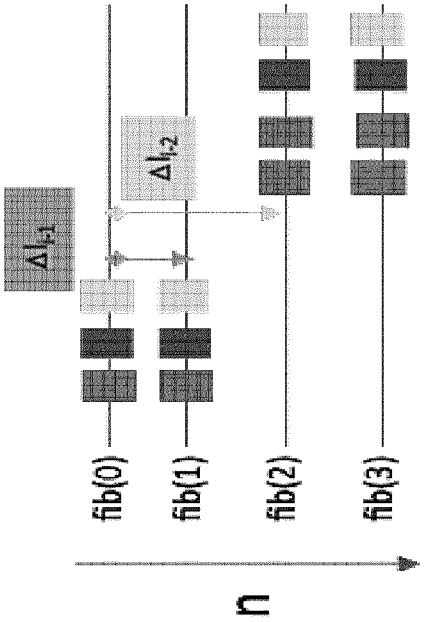

Intel Haswell L3-cache
model runtime / iteration
6.00E-03
5.00E-03
4.00E-03
3.00E-03
2.00E-03
1.00E-03
0.00E+00
1
2
4
8
16
32
64
128
256
512
1024
2048
units

Fig. 72



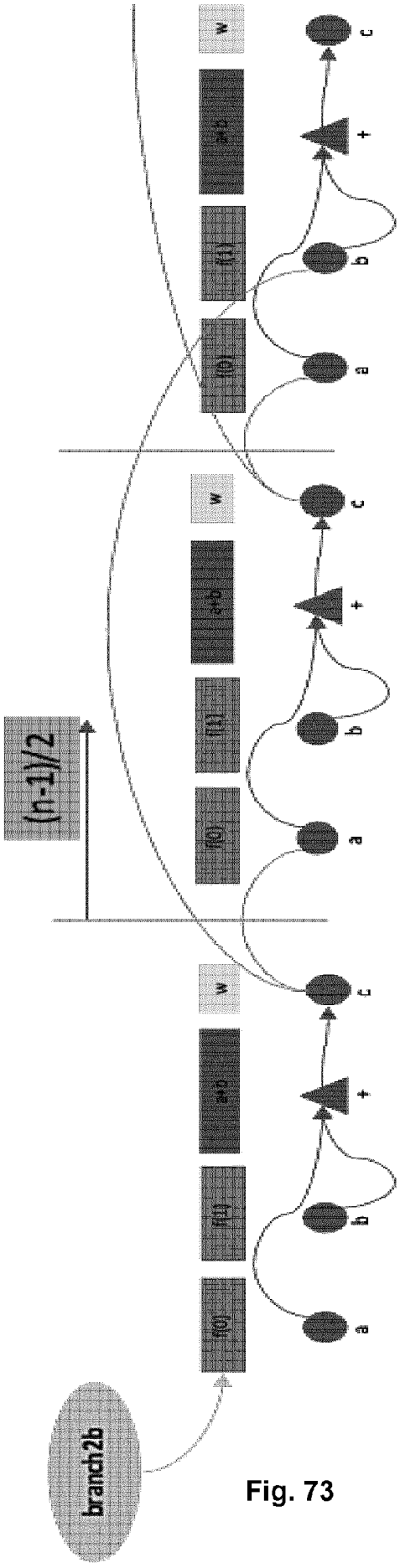


Fig. 73

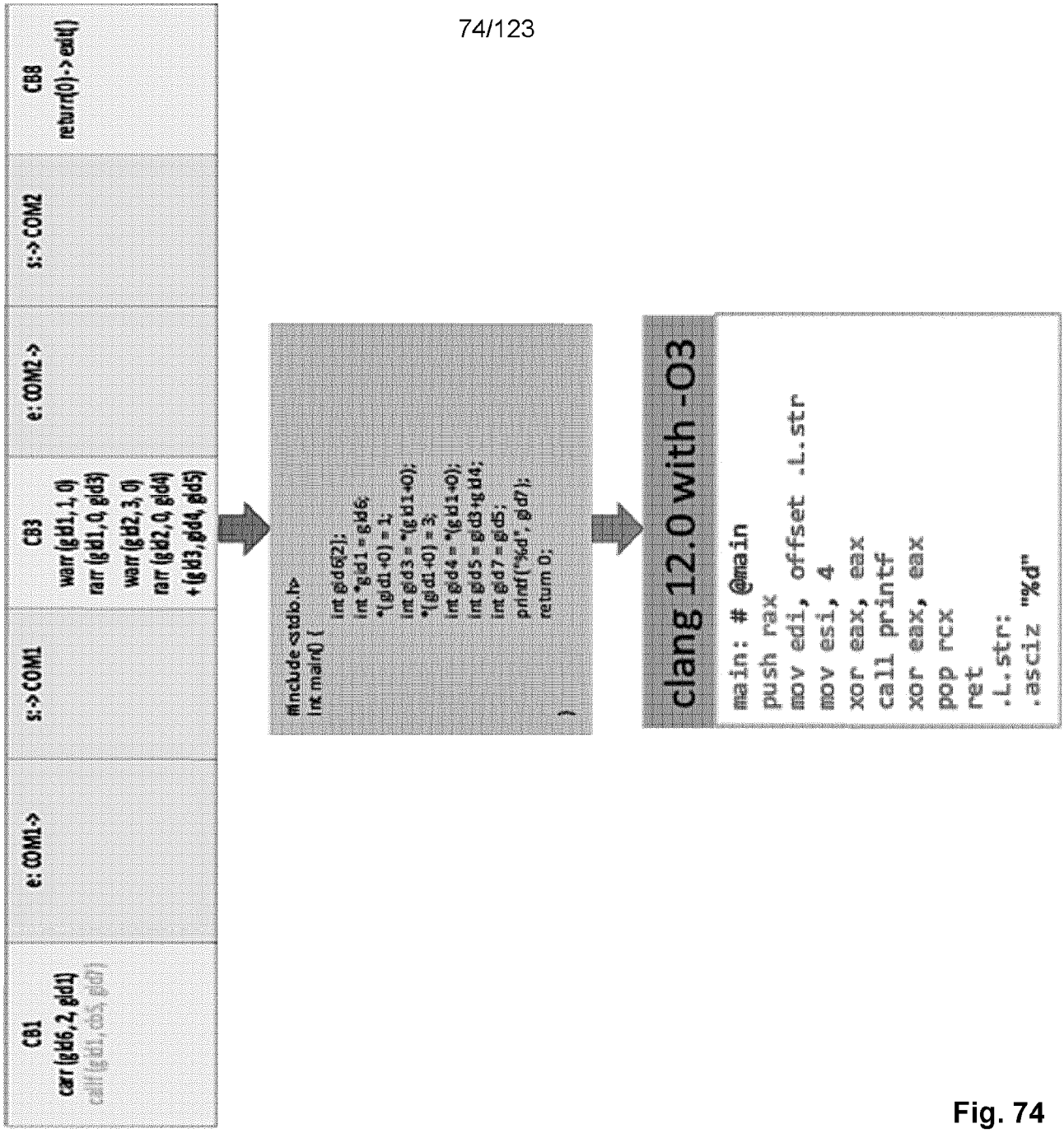

CB1
e: COM1->
S:->COM1
CB3
e: COM2->
S:->COM2
CB8
clang 12.0 with -O3
clang 12.0 with -O3
method

Fig. 74

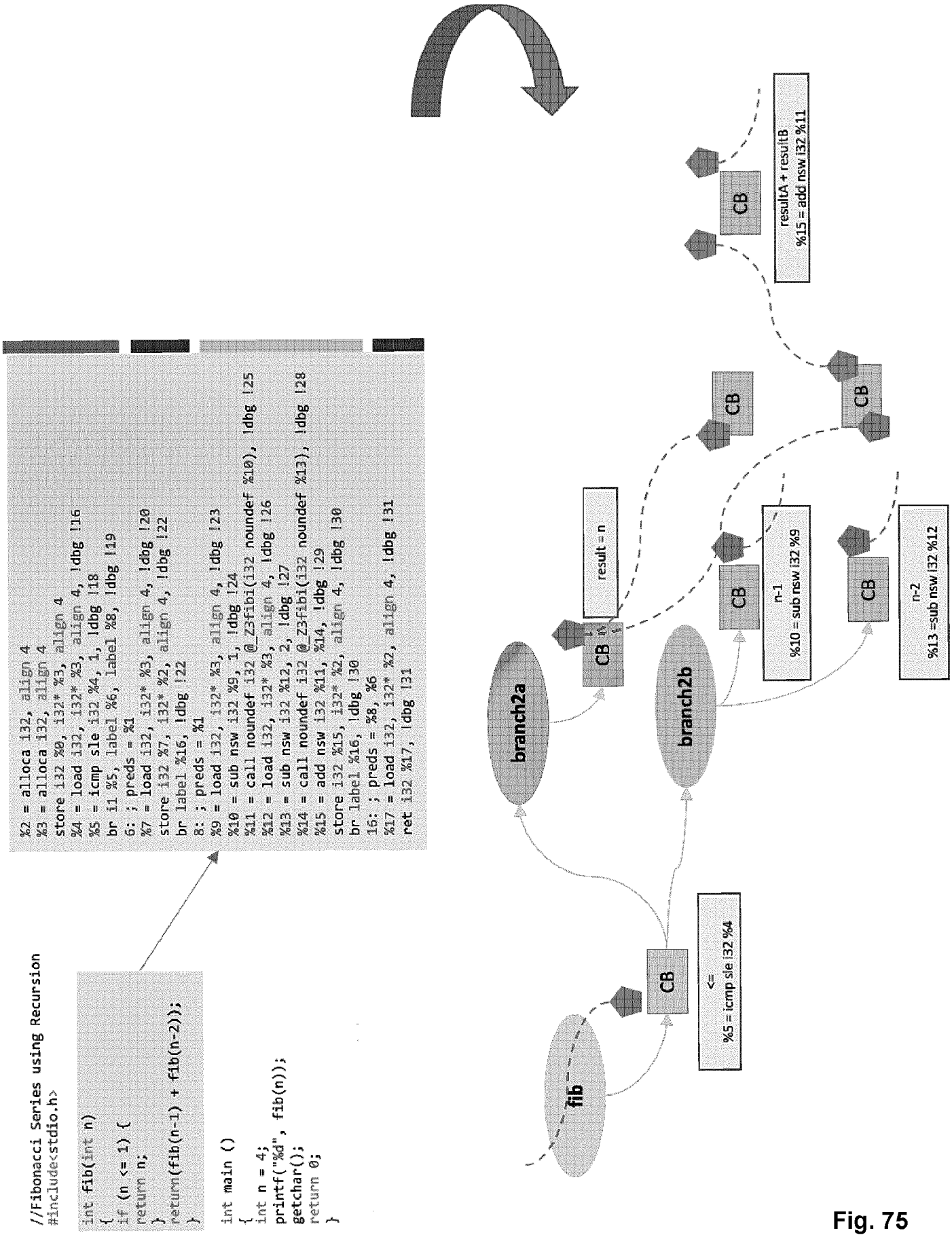

//Fibonacci Series using Recursion
#include<stdio.h>
int fib(int n)
{
if (n <= 1) {
return n;
}
return(fib(n-1) + fib(n-2));
}
int main ()
{
int n = 4;
printf("%d", fib(n));
getchar();
return 0;
}
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 %0, i32* %3, align 4
%4 = load i32, i32* %3, align 4, !dbg !16
%5 = icmp sle i32 %4, 1, !dbg !18
br i1 %5, label %6, label %8, !dbg !19
6: ; preds = %1
%7 = load i32, i32* %3, align 4, !dbg !20
store i32 %7, i32* %2, align 4, !dbg !22
br label %16, !dbg !22
8: ; preds = %1
%9 = load i32, i32* %3, align 4, !dbg !23
%10 = sub nsw i32 %9, 1, !dbg !24
%11 = call noundef i32 @_Z3fibi(i32 noundef %10), !dbg !25
%12 = load i32, i32* %3, align 4, !dbg !26
%13 = sub nsw i32 %12, 2, !dbg !27
%14 = call noundef i32 @_Z3fibi(i32 noundef %13), !dbg !28
%15 = add nsw i32 %11, %14, !dbg !29
store i32 %15, i32* %2, align 4, !dbg !30
br label %16, !dbg !30
16: ; preds = %8, %6
%17 = load i32, i32* %2, align 4, !dbg !31
ret i32 %17, !dbg !31
fib
CB
<=
%5 = icmp sle i32 %4
branch2a
branch2b
result = n
CB
CB
n-1
%10 = sub nsw i32 %9
CB
CB
n-2
%13 =sub nsw i32 %12
CB
resultA + resultB
%15 = add nsw i32 %11

Fig. 75

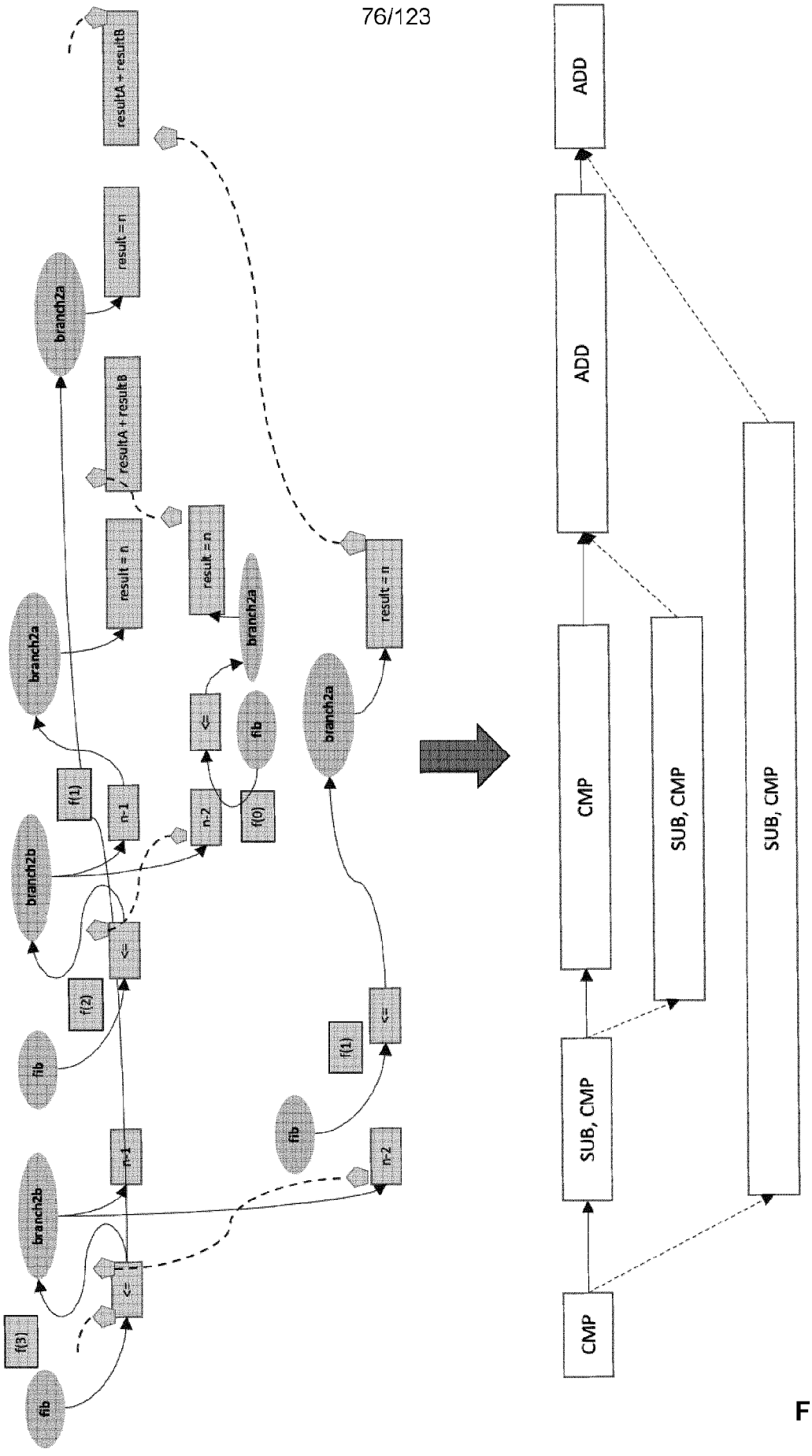
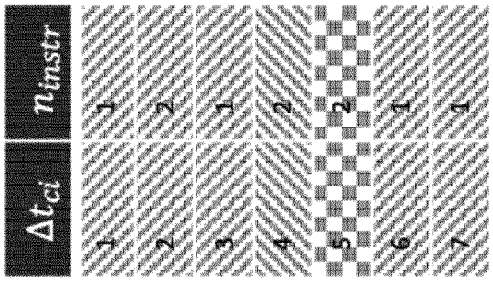

resultA + resultB
result = n
branch2a
resultA + resultB
result = n
branch2a
result = n
branch2a
result = n
branch2a
f(1)
n-1
branch2b
<=
f(2)
fib
n-1
branch2b
<=
f(3)
fib
<=
n-2
f(0)
fib
branch2a
<=
f(1)
fib
n-2
ADD
ADD
CMP
SUB, CMP
SUB, CMP
SUB, CMP
CMP

Fig. 76


Δt_{ci}
n_{instr}
1 2 3 4 5 6 7
1 2 1 2 2 1 1

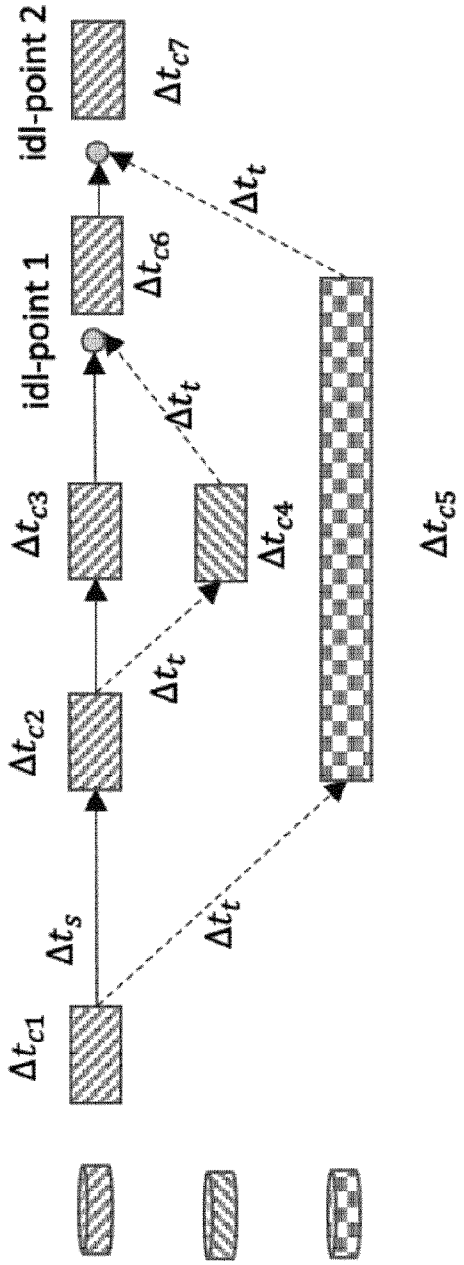
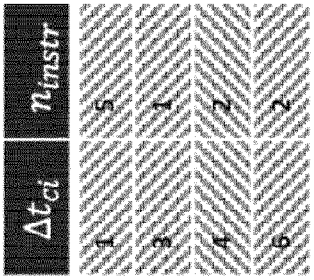
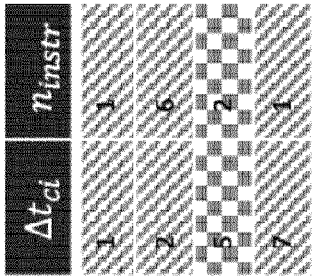
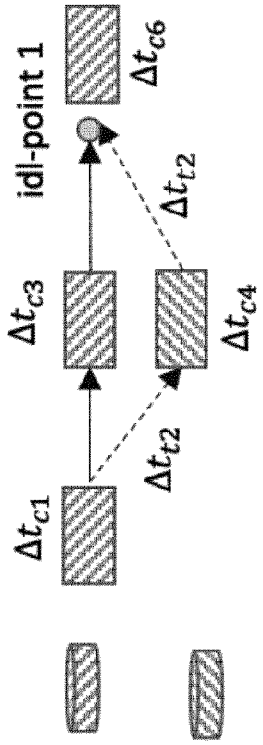

idl-point 2
idl-point 1
Δt_{c1}
Δt_{c2}
Δt_{c3}
Δt_{c4}
Δt_{c5}
Δt_{c6}
Δt_{c7}
Δt_s
Δt_t
Δt_t
Δt_t
Δt_t

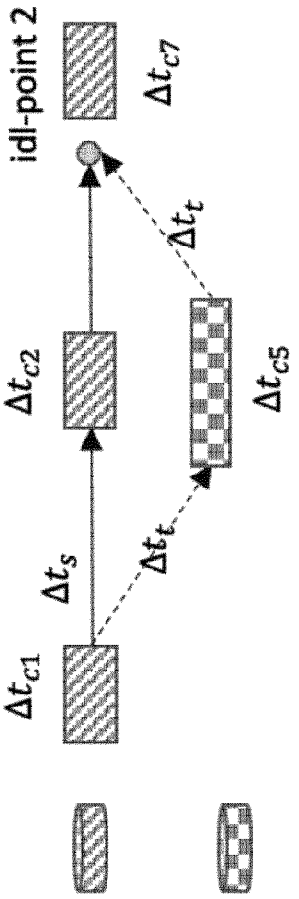
Fig. 77


Δt_{ci}
n_{instr}
1
3
4
6
5
1
2
2


Δt_{ci}
n_{instr}
1
2
5
7
1
6
2
1

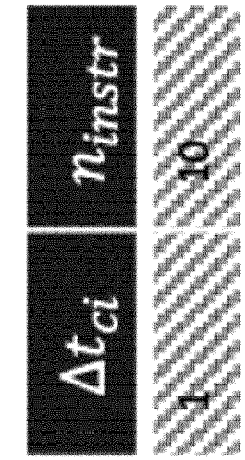

idl-point 1
Δt_{c1}
Δt_{c3}
Δt_{c6}
Δt_{t2}
Δt_{t2}
Δt_{c4}

c1


idl-point 2
Δt_{c1}
Δt_s
Δt_{c2}
Δt_t
Δt_t
Δt_{c5}
Δt_{c7}

c2

Fig. 78



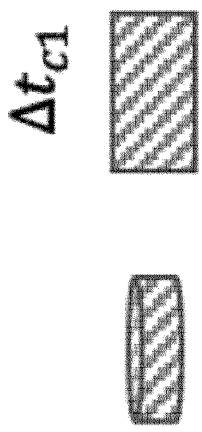


Fig. 79

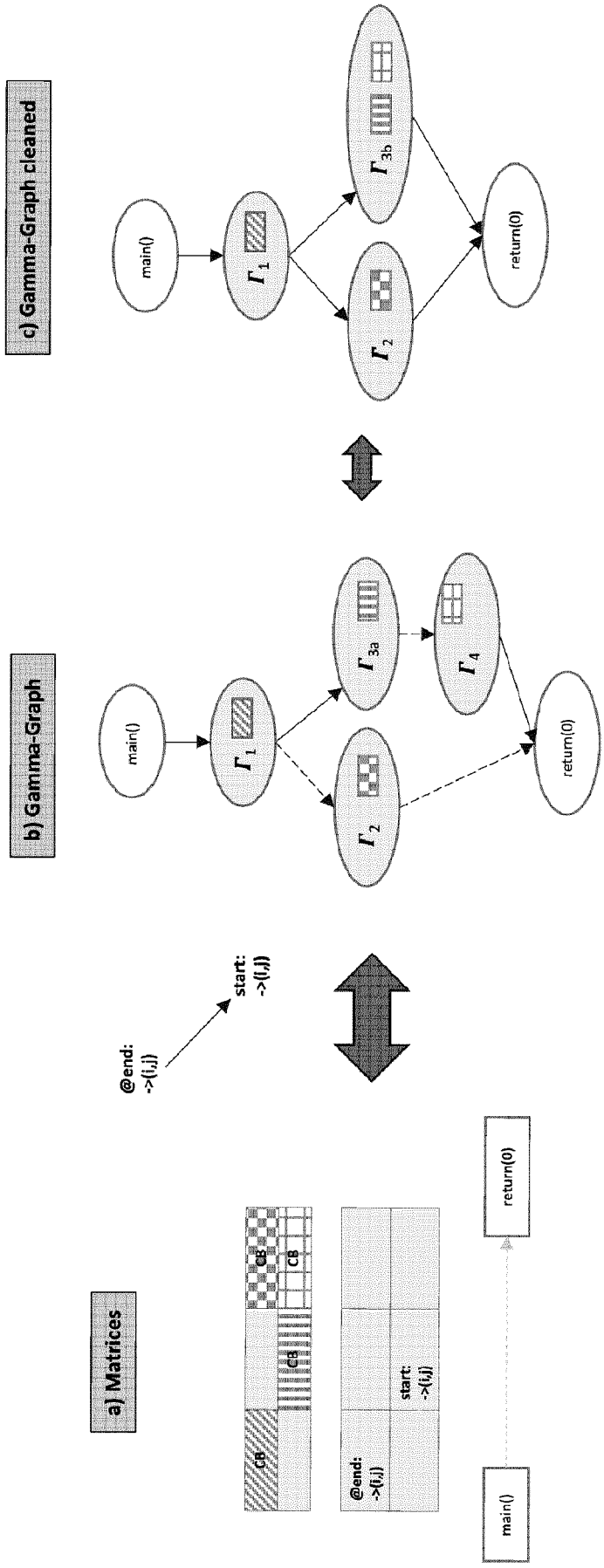

c) Gamma-Graph cleaned
main()
Γ1
Γ3b
Γ2
return(0)
b) Gamma-Graph
main()
Γ1
Γ3a
Γ4
Γ2
return(0)
@end:
->(i,j)
start:
->(i,j)
a) Matrices
CB
CB
CB
@end:
->(i,j)
start:
->(i,j)
return(0)
main()

Fig. 80

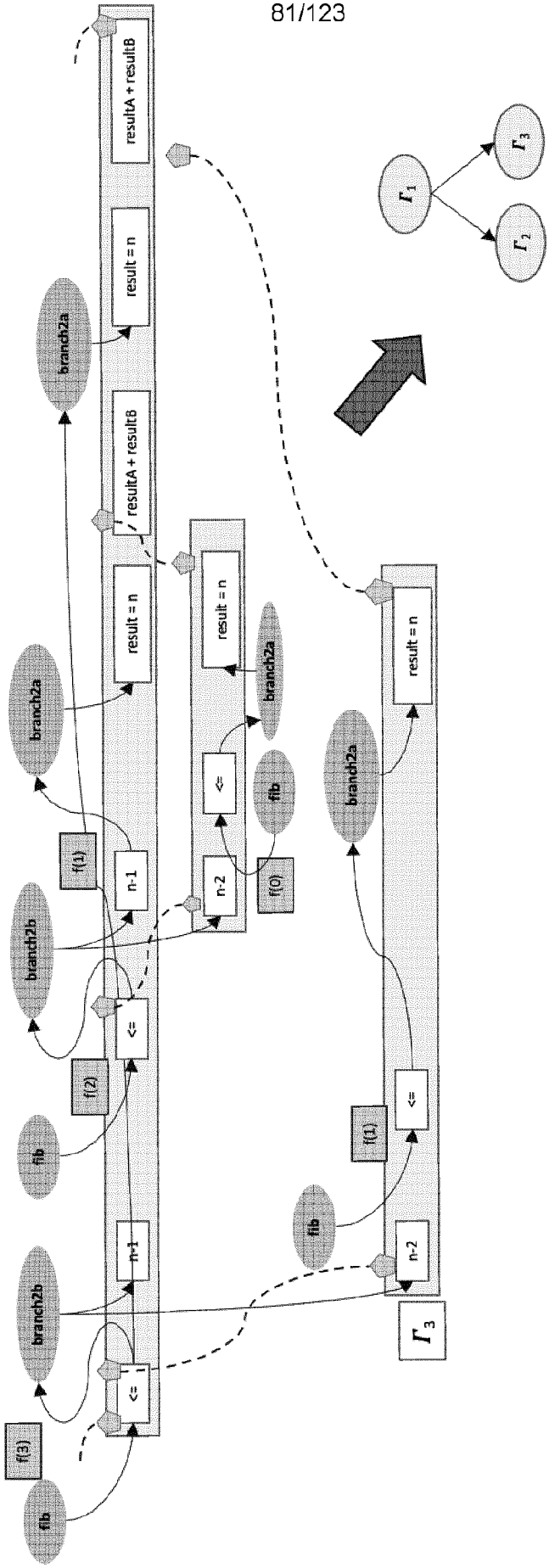

resultA + resultB
result = n
branch2a
resultA + resultB
result = n
branch2a
f(1)
n-1
branch2b
<=
f(2)
fib
n-1
branch2b
<=
f(3)
fib
result = n
branch2a
<=
n-2
fib
f(0)
result = n
branch2a
<=
f(1)
fib
n-2
Γ3
Γ1
Γ3
Γ2

Fig. 81

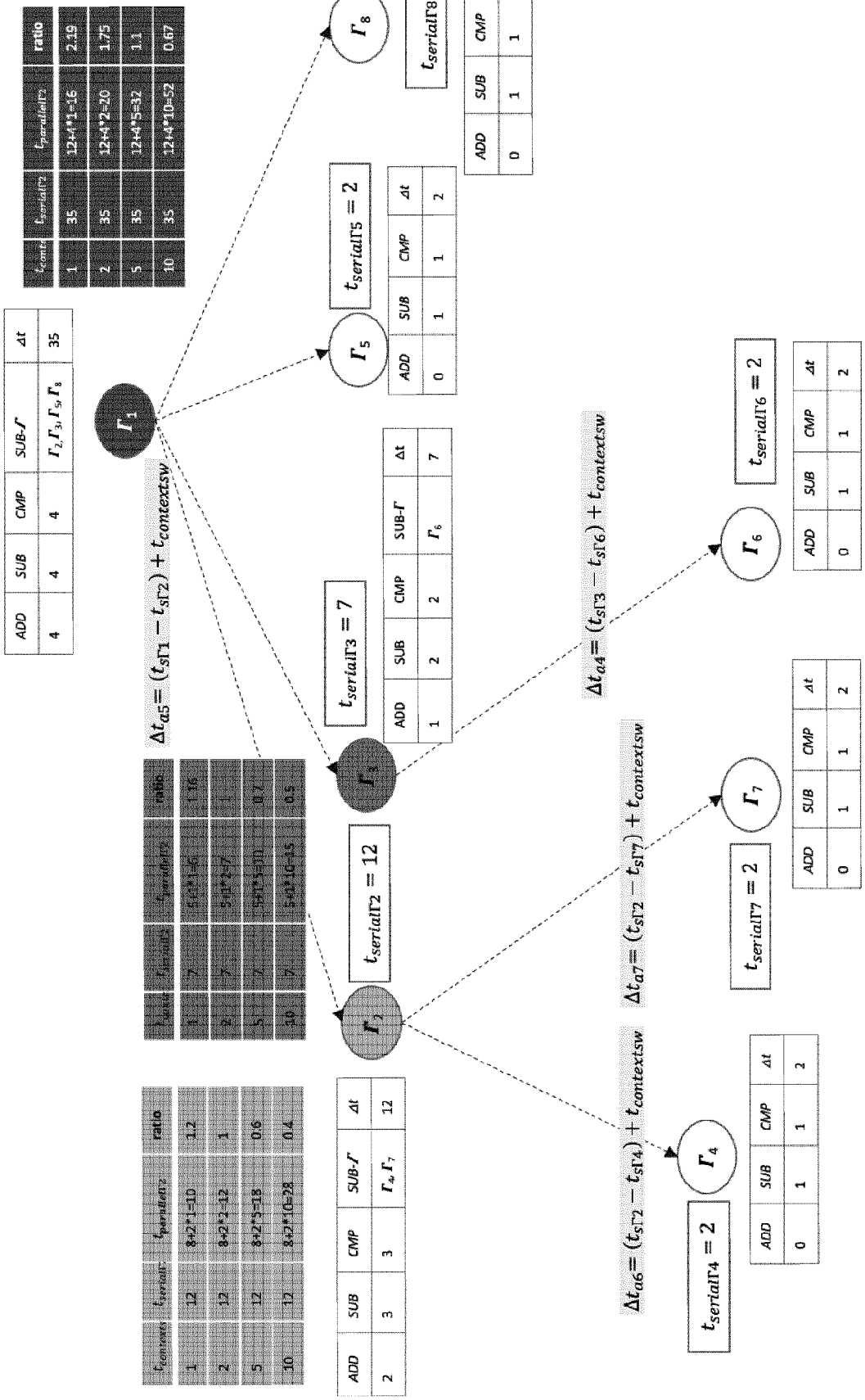

Γ_1
Γ_2
Γ_3
Γ_4
Γ_5
Γ_6
Γ_7
Γ_8
$\Delta t_{a5} = (t_{sT1} - t_{sT2}) + t_{contextsw}$
$t_{serialT3} = 7$
$t_{serialT2} = 12$
$\Delta t_{a4} = (t_{sT3} - t_{sT6}) + t_{contextsw}$
$\Delta t_{a6} = (t_{sT2} - t_{sT4}) + t_{contextsw}$
$\Delta t_{a7} = (t_{sT2} - t_{sT7}) + t_{contextsw}$
$t_{serialT4} = 2$
$t_{serialT7} = 2$
$t_{serialT6} = 2$
$t_{serialT5} = 2$
$t_{serialT8} = 2$
ADD SUB CMP SUB-Γ Δt
4 4 4 $\Gamma_2, \Gamma_3, \Gamma_5, \Gamma_8$ 35
$t_{contextsw}$ $t_{serialT2}$ $t_{parallelT2}$ ratio
1 35 12+4*1=16 2.19
2 35 12+4*2=20 1.75
5 35 12+4*5=32 1.1
10 35 12+4*10=52 0.67
$t_{contextsw}$ $t_{serialT3}$ $t_{parallelT3}$ ratio
1 7 8+2*1=10 1.2
2 7 8+2*2=12 1
5 7 8+2*3=18 0.6
10 7 8+2*10=28 0.4
ADD SUB CMP SUB-Γ Δt
2 3 3 Γ_4, Γ_7 12
ADD SUB CMP SUB-Γ Δt
1 2 2 Γ_6 7
ADD SUB CMP Δt
0 1 1 2
ADD SUB CMP Δt
0 1 1 2
ADD SUB CMP Δt
0 1 1 2
ADD SUB CMP Δt
0 1 1 2
ADD SUB CMP Δt
0 1 1 2

Fig. 82

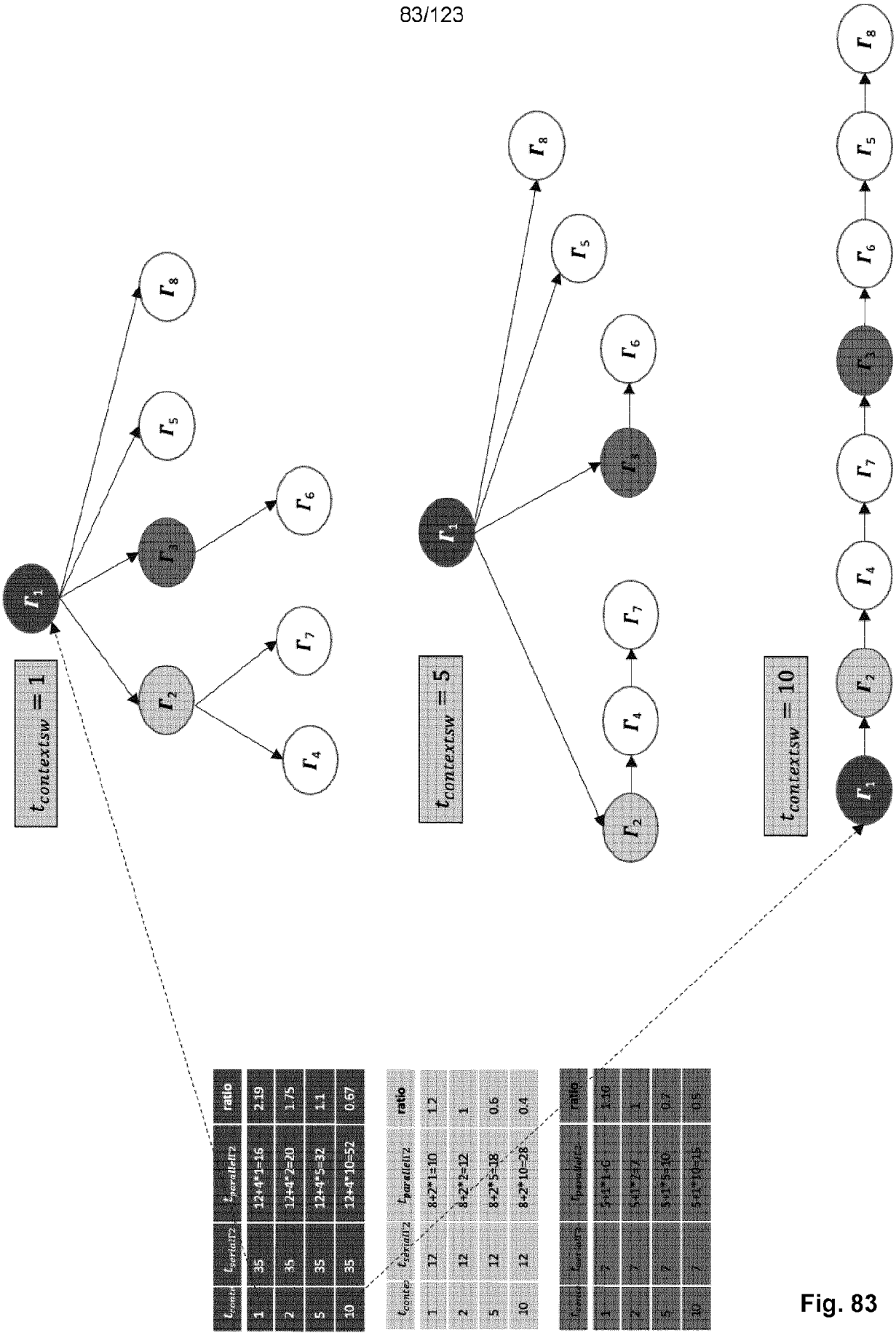

$t_{contextsw} = 1$
$t_{contextsw} = 5$
$t_{contextsw} = 10$
Γ_1
Γ_2
Γ_3
Γ_4
Γ_5
Γ_6
Γ_7
Γ_8
ratio
2.19
1.75
1.1
0.67
1.2
1
0.6
0.4
1.10
1
0.7
0.5

Fig. 83

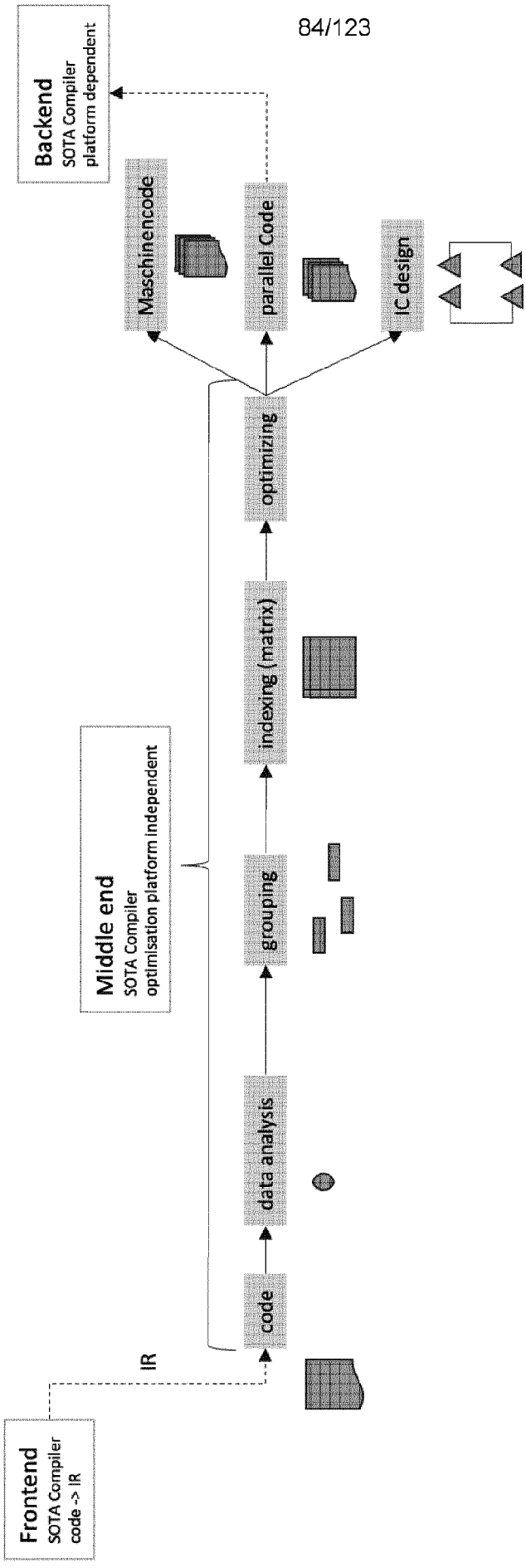

Backend
SOTA Compiler
platform dependent
Maschinencode
parallel Code
IC design
optimizing
indexing (matrix)
grouping
data analysis
code
IR
Middle end
SOTA Compiler
optimisation platform independent
Frontend
SOTA Compiler
code -> IR

Fig. 84

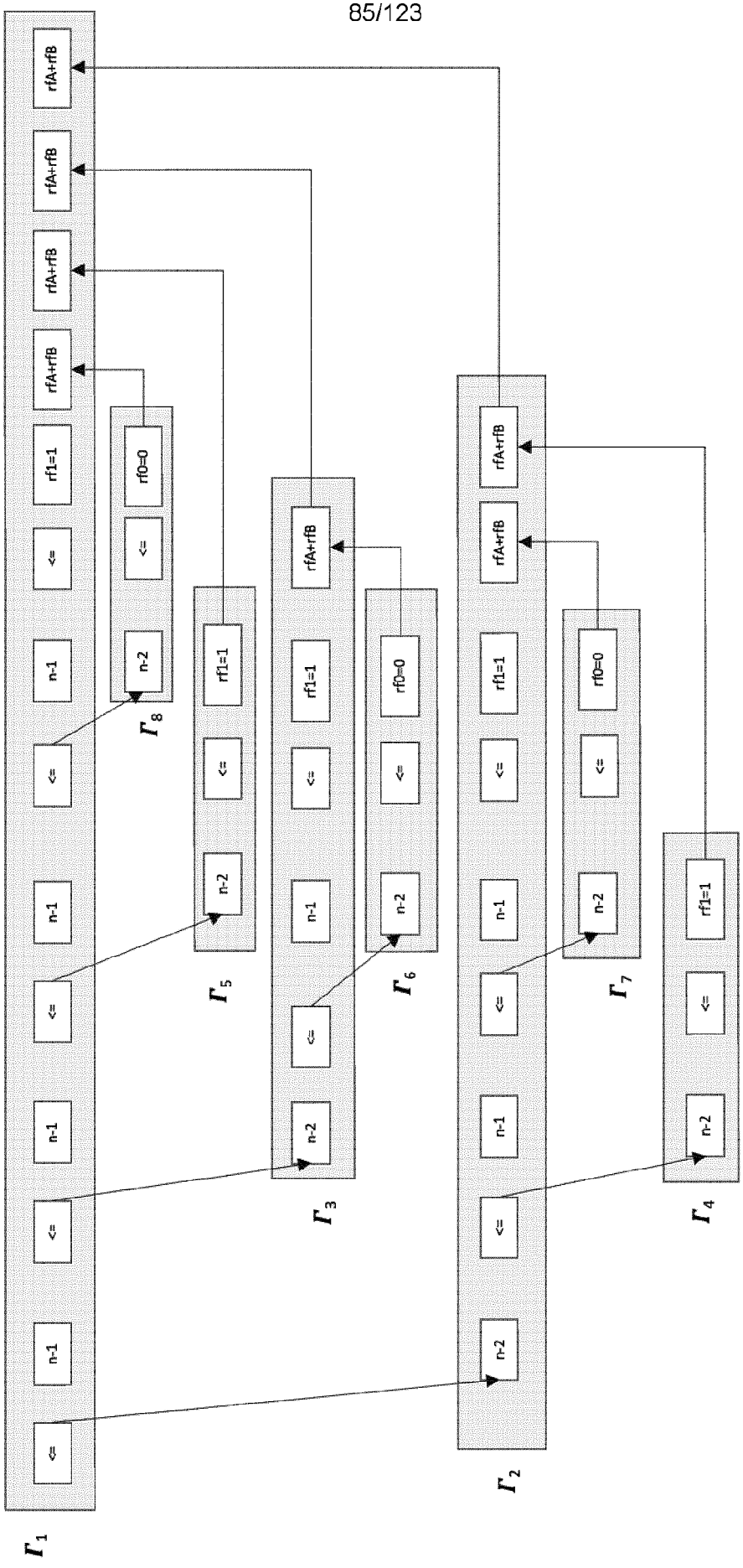
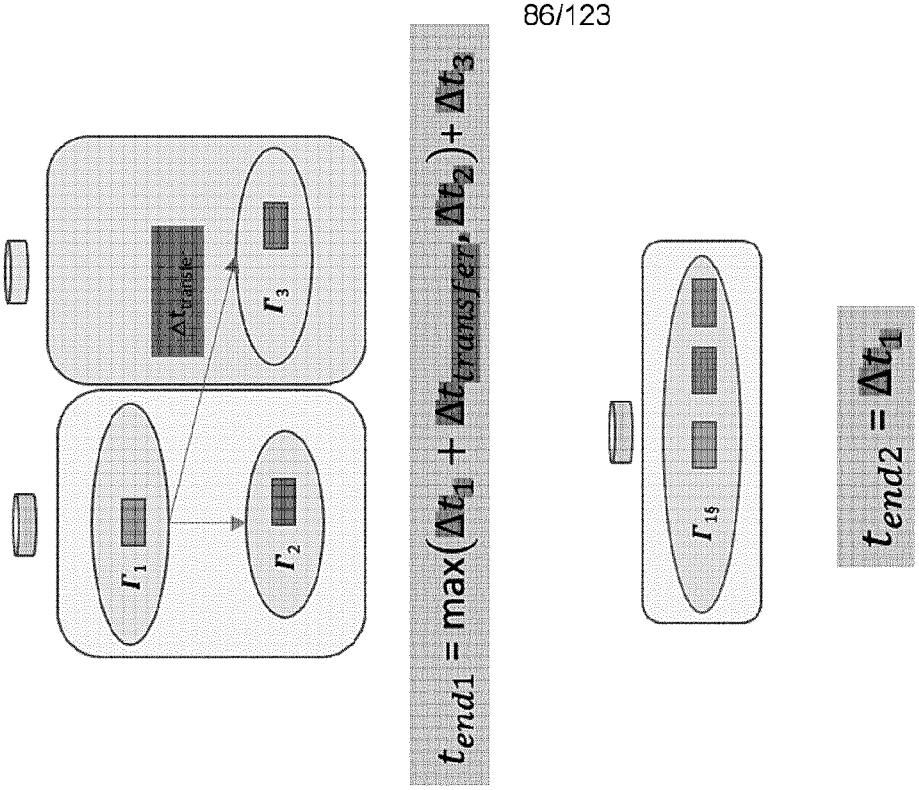


Fig. 85


Γ_1
Γ_2
Γ_3
$\Delta t_{transfer}$
$t_{end1} = \max(\Delta t_1 + \Delta t_{transfer}, \Delta t_2) + \Delta t_3$
$\Gamma_{1\S}$
$t_{end2} = \Delta t_1$

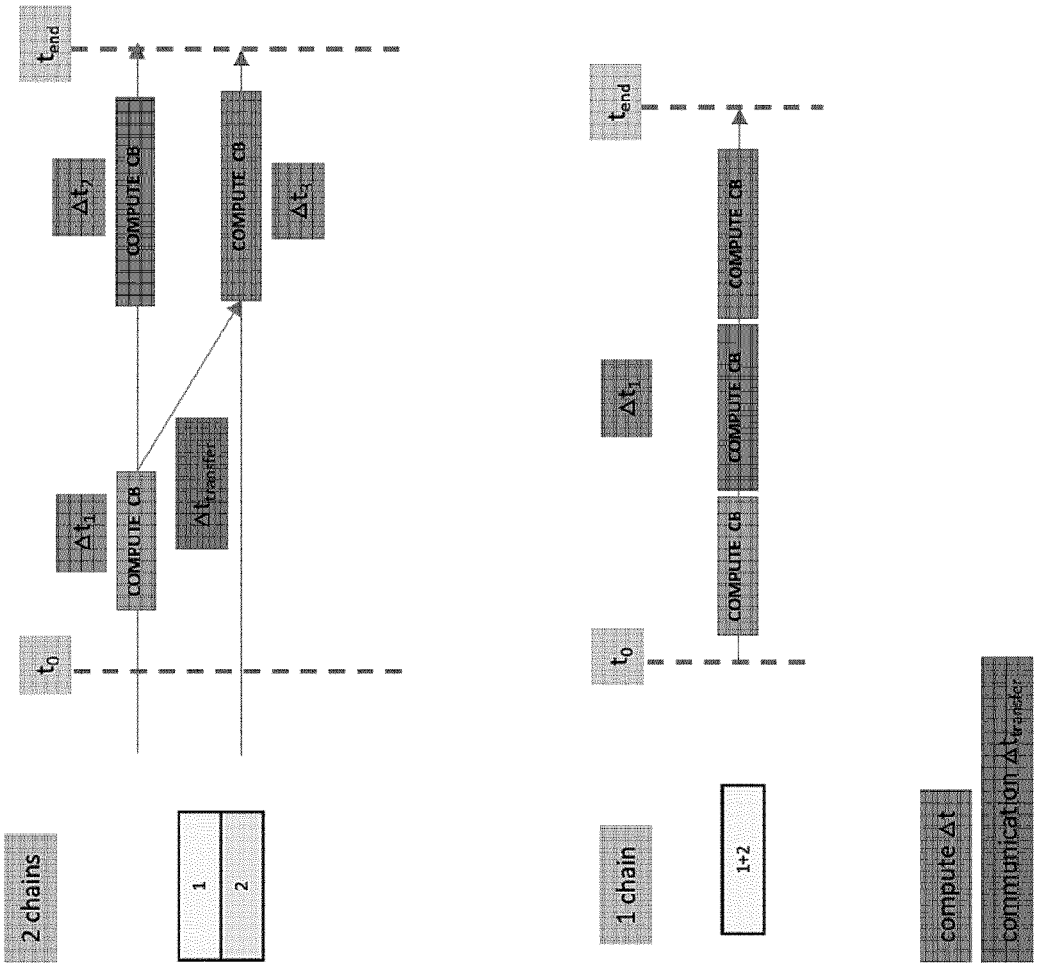

t_{end}
Δt_7
COMPUTE CB
COMPUTE CB
Δt_1
$\Delta t_{transfer}$
t_0
2 chains
1
2
t_{end}
COMPUTE CB
COMPUTE CB
COMPUTE CB
Δt_1
t_0
1 chain
1+2
compute Δt
communication $\Delta t_{transfer}$

Fig. 86

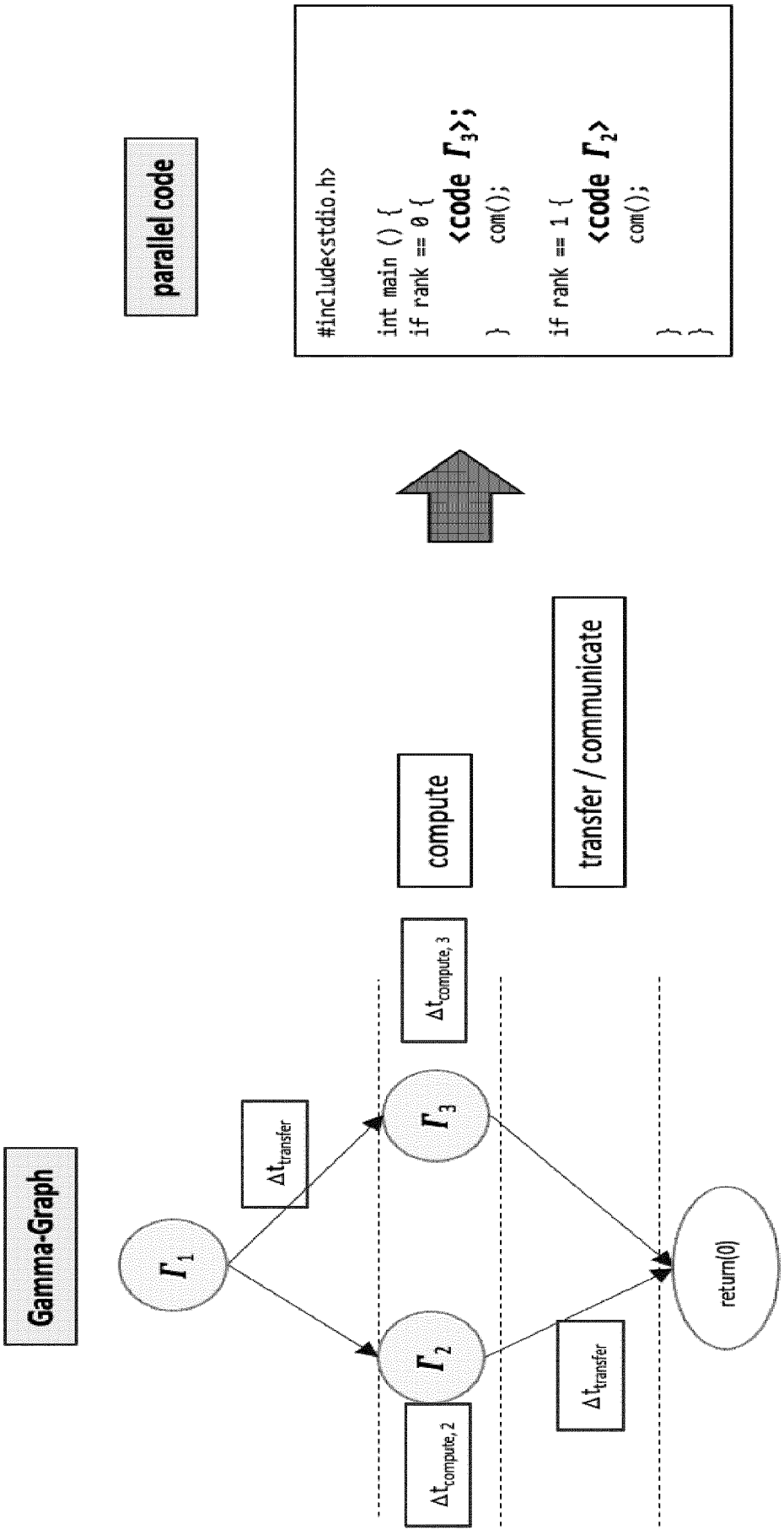

Gamma-Graph
Γ1
Δttransfer
Γ3
Γ2
Δtcompute, 3
Δtcompute, 2
Δttransfer
return(0)
compute
transfer / communicate
parallel code
#include<stdio.h>
int main () {
if rank == 0 {
<code Γ3>;
com();
}
if rank == 1 {
<code Γ2>
com();
}
}

Fig. 87

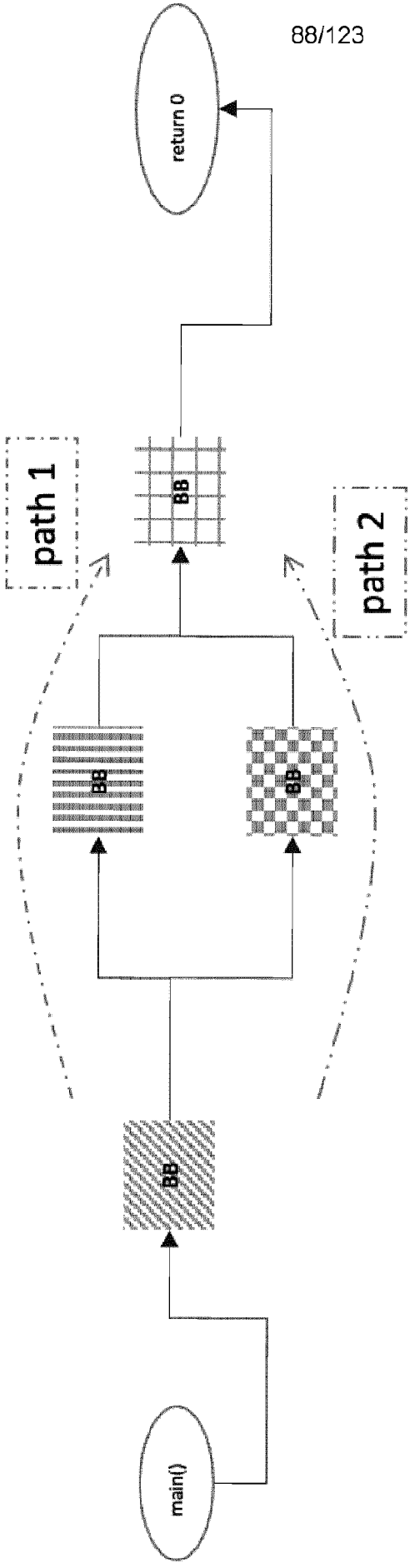


Fig. 88

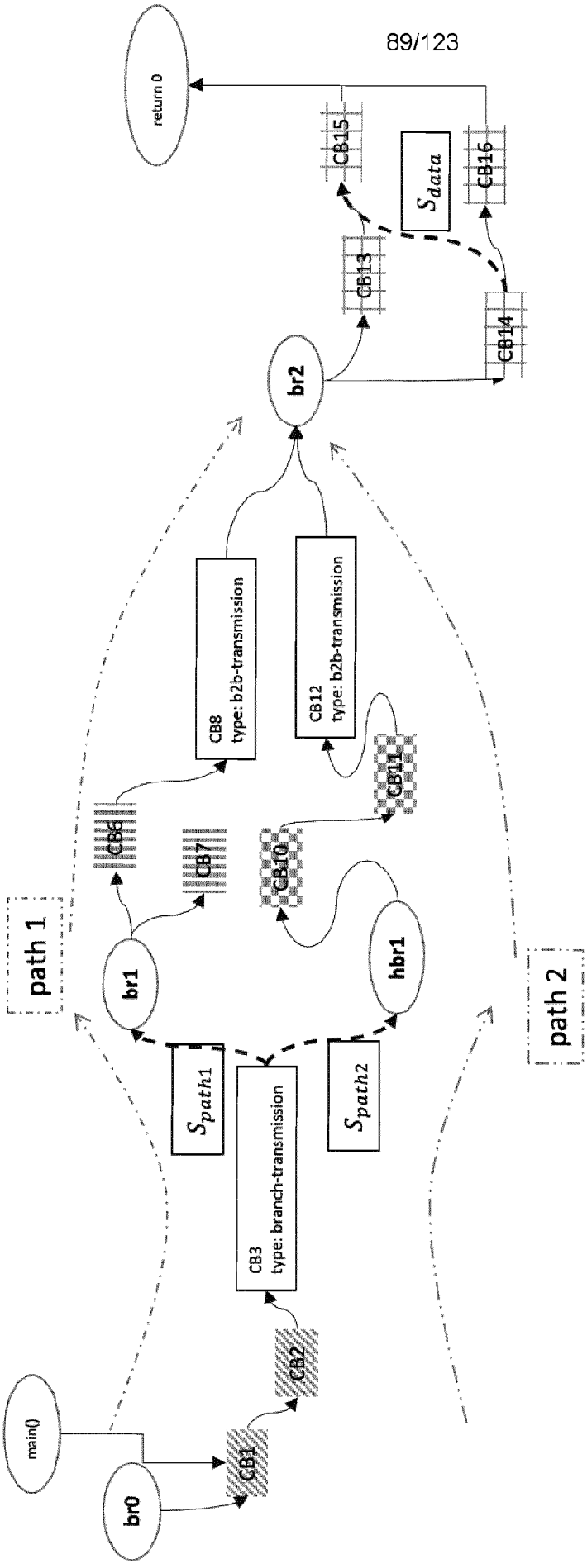

return 0
CB15
S_{data}
CB16
CB13
CB14
br2
CB8
type: b2b-transmission
CB12
type: b2b-transmission
CB11
CB6
CB7
CB10
path 1
br1
hbr1
path 2
S_{path1}
S_{path2}
CB3
type: branch-transmission
CB2
main()
CB1
br0

Fig. 89

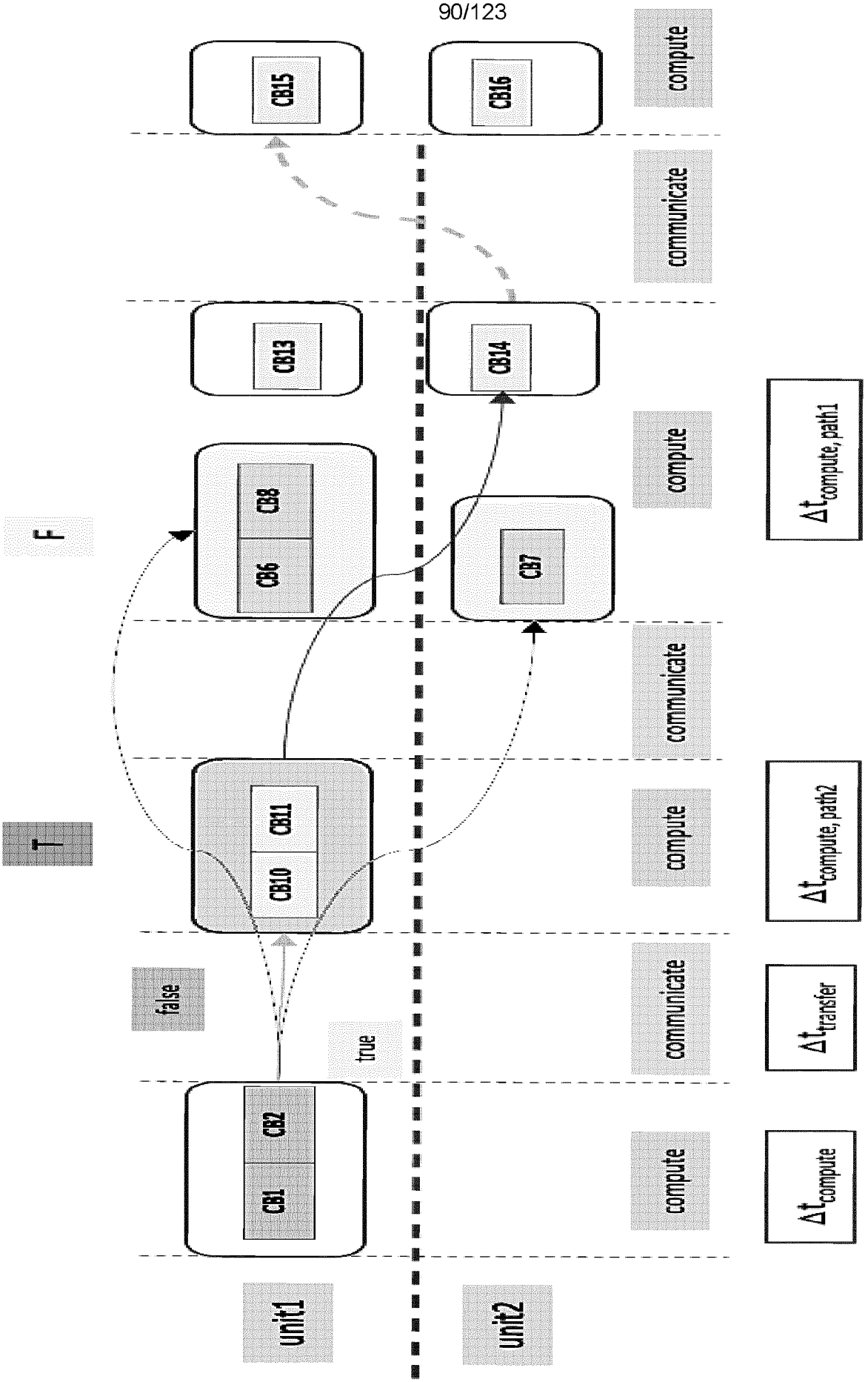

unit1
unit2
CB1
CB2
CB10
CB11
CB6
CB8
CB13
CB14
CB15
CB16
CB7
true
false
T
F
compute
communicate
compute
communicate
compute
communicate
compute
$\Delta t_{compute}$
$\Delta t_{transfer}$
$\Delta t_{compute, path2}$
$\Delta t_{compute, path1}$

Fig. 90

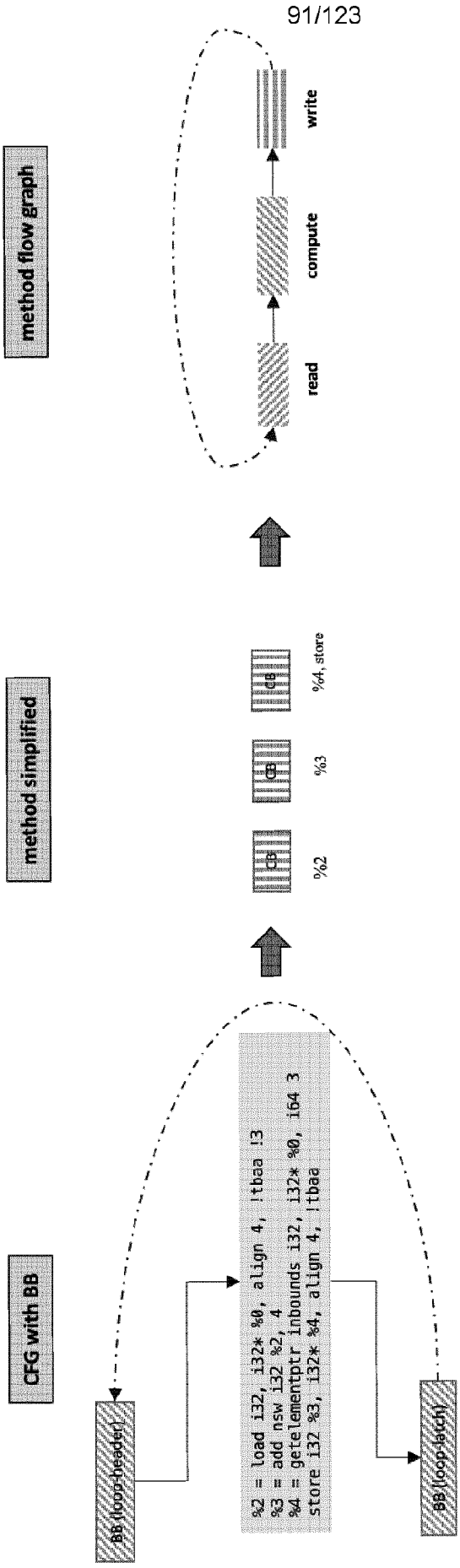

method flow graph
read
compute
write
method simplified
%2
%3
%4, store
CFG with BB
BB (loop.header)
%2 = load i32, i32* %0, align 4, !tbaa !3
%3 = add nsw i32 %2, 4
%4 = getelementptr inbounds i32, i32* %0, i64 3
store i32 %3, i32* %4, align 4, !tbaa
BB (loop.latch)

Fig. 91

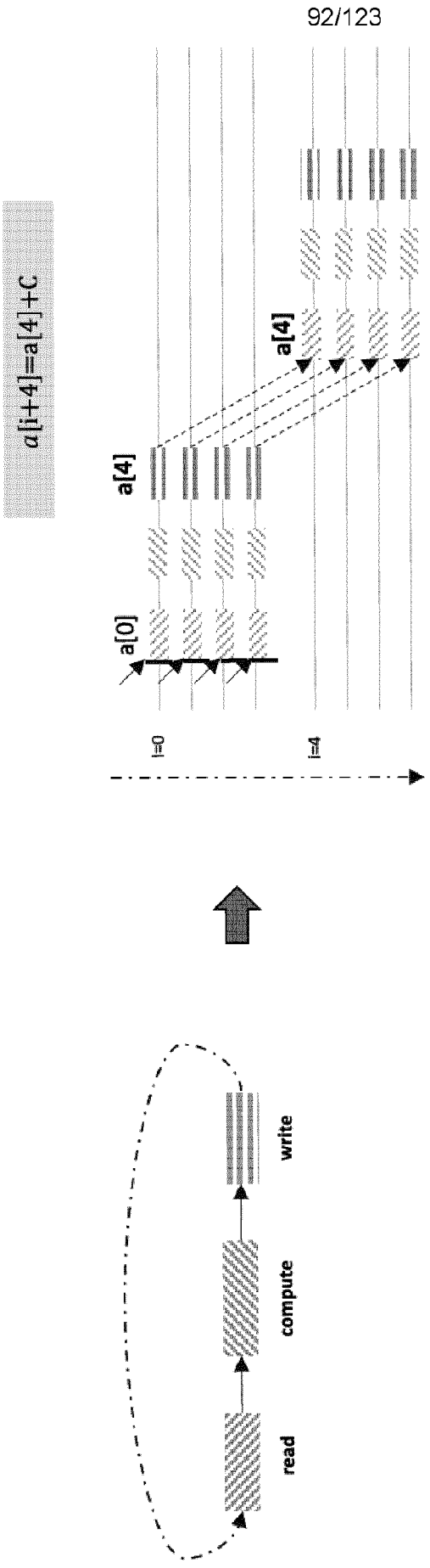


Fig. 92

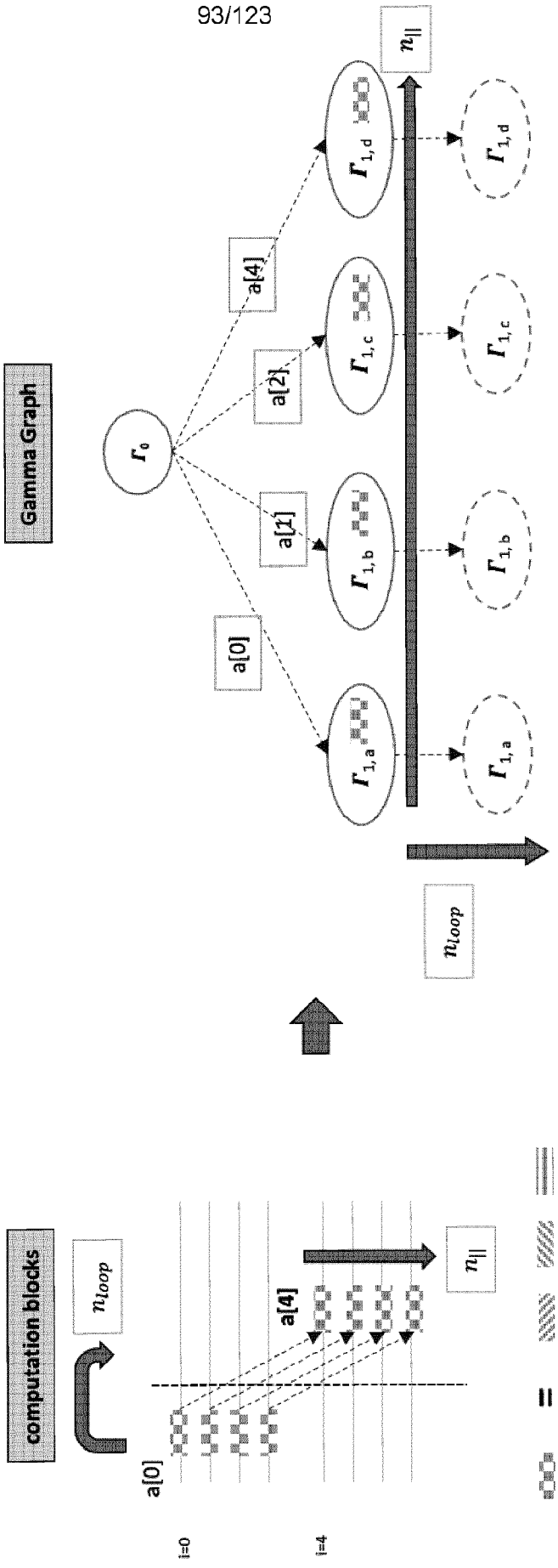

Gamma Graph
Γ_0
a[0]
a[1]
a[2]
a[4]
$\Gamma_{1,a}$
$\Gamma_{1,b}$
$\Gamma_{1,c}$
$\Gamma_{1,d}$
$n_{||}$
n_{loop}
computation blocks
n_{loop}
a[0]
a[4]
$n_{||}$
=
i=0
i=4

Fig. 93

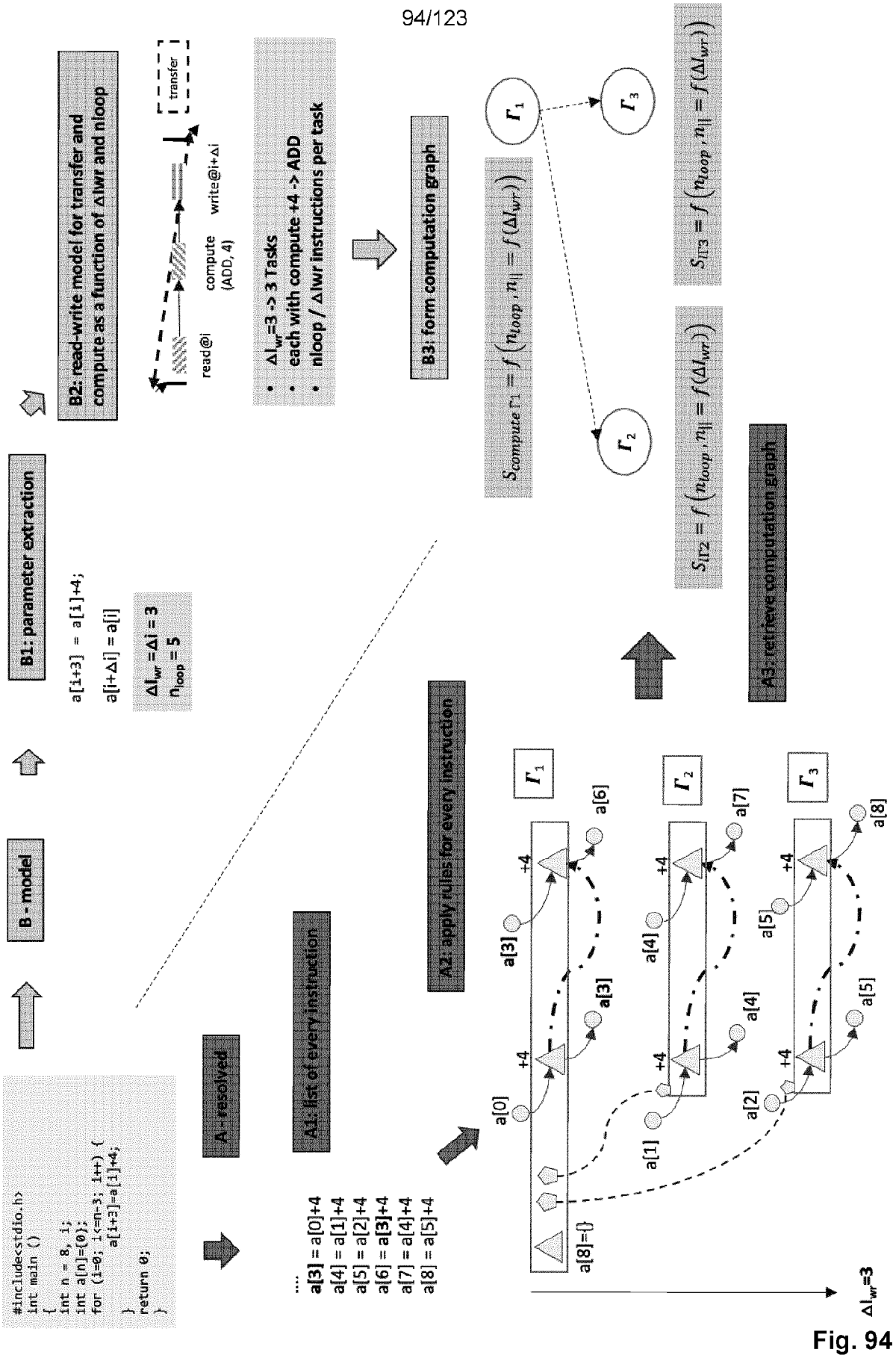

#include<stdio.h>
int main ()
{
int n = 8, i;
int a[n]={0};
for (i=0; i<=n-3; i++) {
a[i+3]=a[i]+4;
}
return 0;
}
A - resolved
B - model
A1: list of every instruction
a[3] = a[0]+4
a[4] = a[1]+4
a[5] = a[2]+4
a[6] = a[3]+4
a[7] = a[4]+4
a[8] = a[5]+4
A2: apply rules for every instruction
B1: parameter extraction
a[i+3] = a[i]+4;
a[i+Δi] = a[i]
ΔIwr =Δi = 3
nloop = 5
B2: read-write model for transfer and compute as a function of ΔIwr and nloop
read@i
compute (ADD, 4)
write@i+Δi
transfer
ΔIwr=3 -> 3 Tasks
each with compute +4 -> ADD
nloop / ΔIwr instructions per task
B3: form computation graph
A3: retrieve computation graph
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[8]={}
+4
Γ1
Γ2
Γ3
ΔIwr=3

Fig. 94

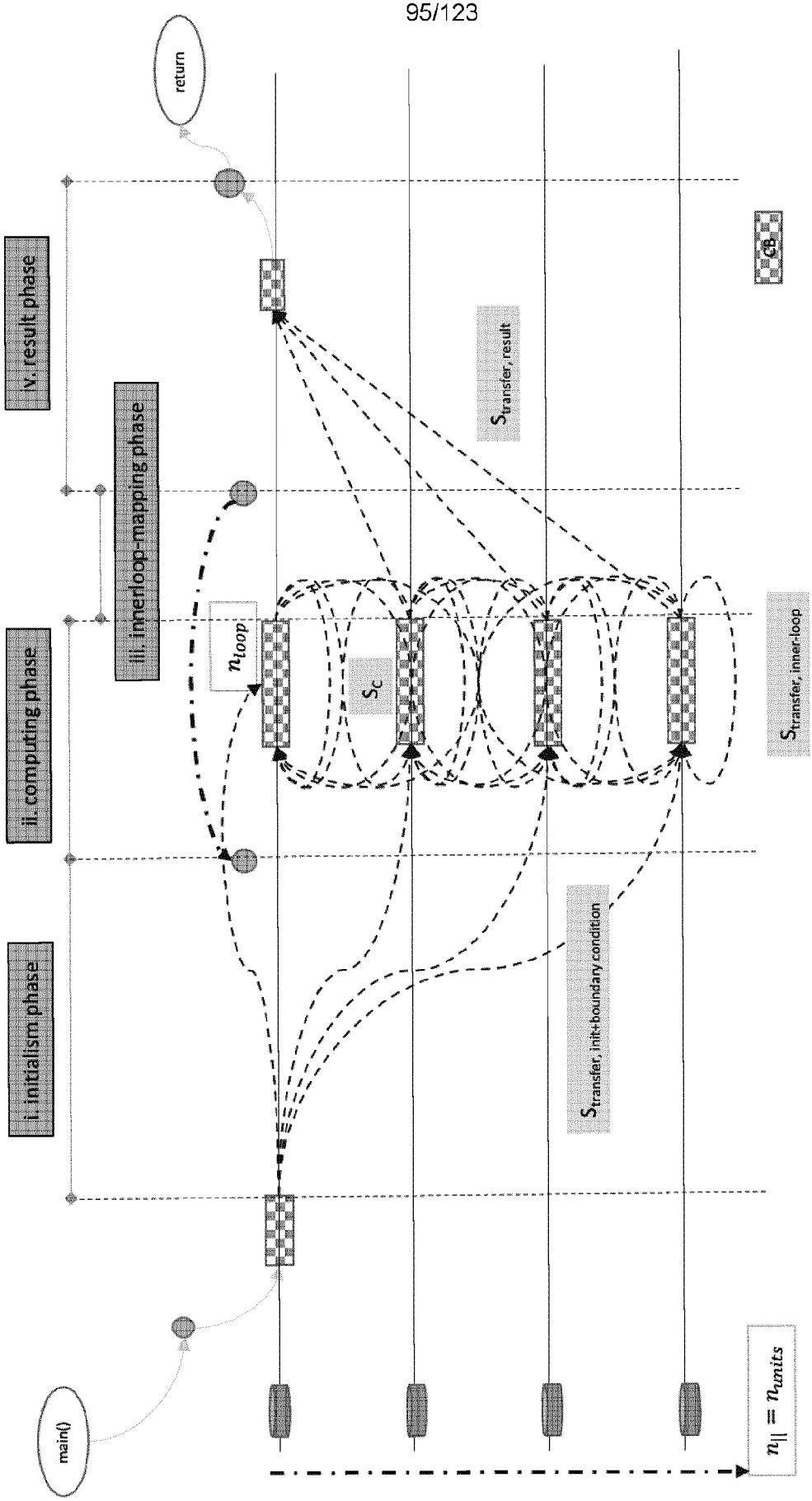

return
iv. result phase
iii. innerloop-mapping phase
ii. computing phase
i. initialism phase
main()
GB
$S_{transfer,\,result}$
$S_{transfer,\,inner\text{-}loop}$
$S_{transfer,\,init+boundary\,condition}$
S_C
n_{loop}
$n_{||} = n_{units}$

Fig. 95

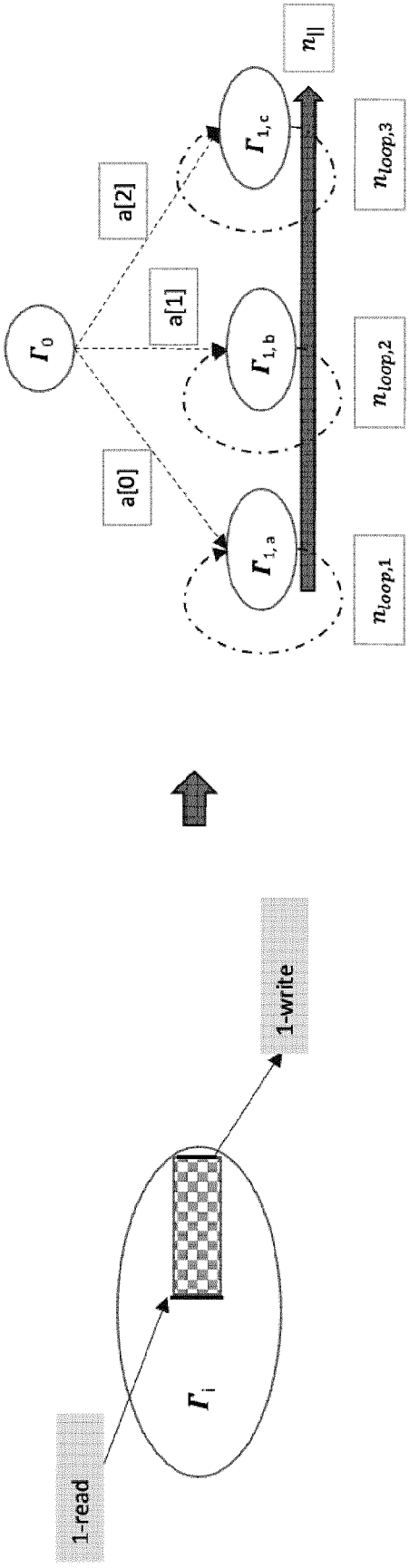

$n_{||}$
$\Gamma_{1,c}$
$n_{loop,3}$
a[2]
a[1]
Γ_0
$\Gamma_{1,b}$
$n_{loop,2}$
a[0]
$\Gamma_{1,a}$
$n_{loop,1}$
1-write
1-read
Γ_i

Fig. 96

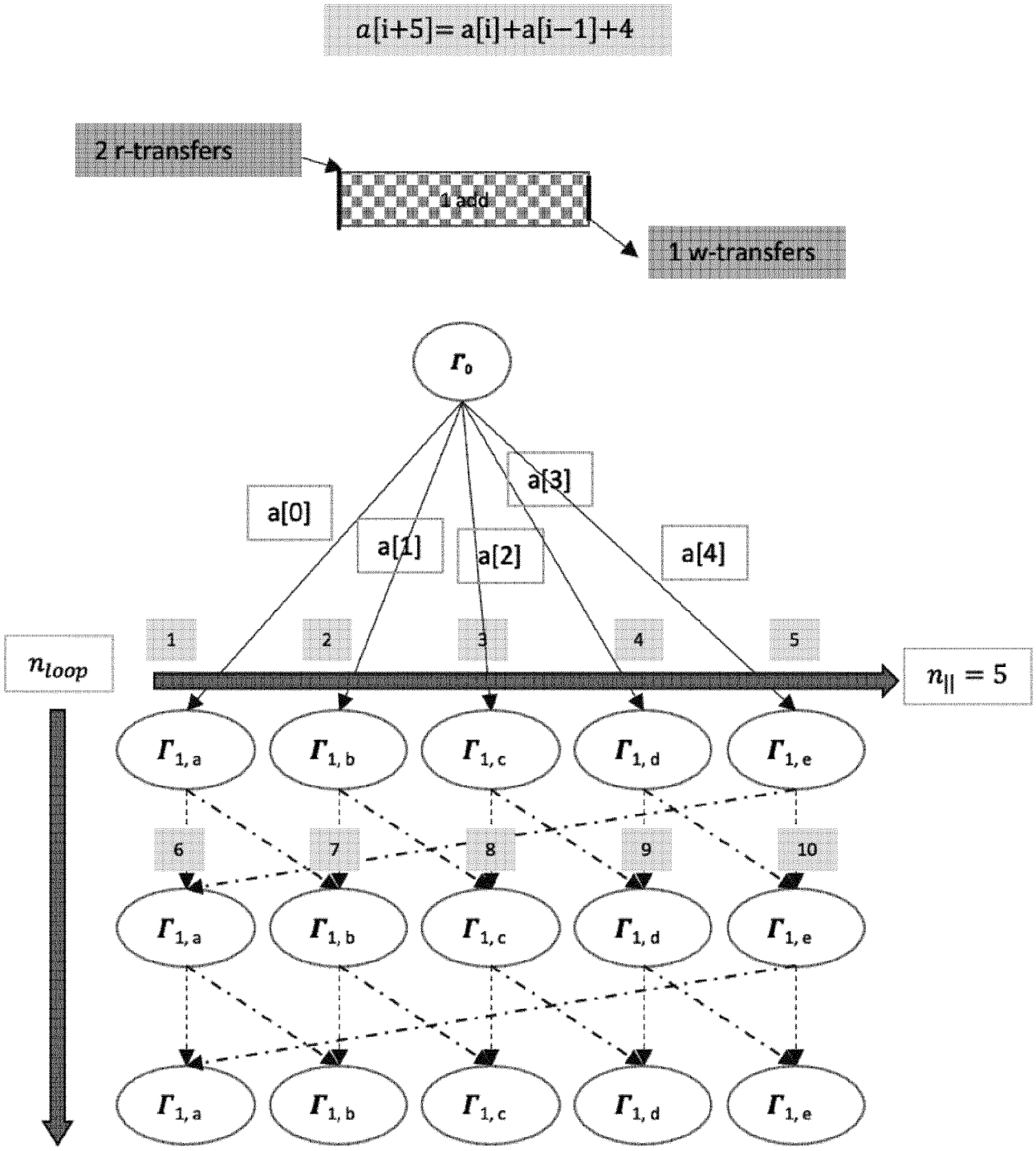

a[i+5]= a[i]+a[i−1]+4
2 r-transfers
1 add
1 w-transfers
Γ_0
a[0]
a[1]
a[2]
a[3]
a[4]
1
2
3
4
5
n_{loop}
$n_{||} = 5$
$\Gamma_{1,a}$
$\Gamma_{1,b}$
$\Gamma_{1,c}$
$\Gamma_{1,d}$
$\Gamma_{1,e}$
6
7
8
9
10

Fig. 97

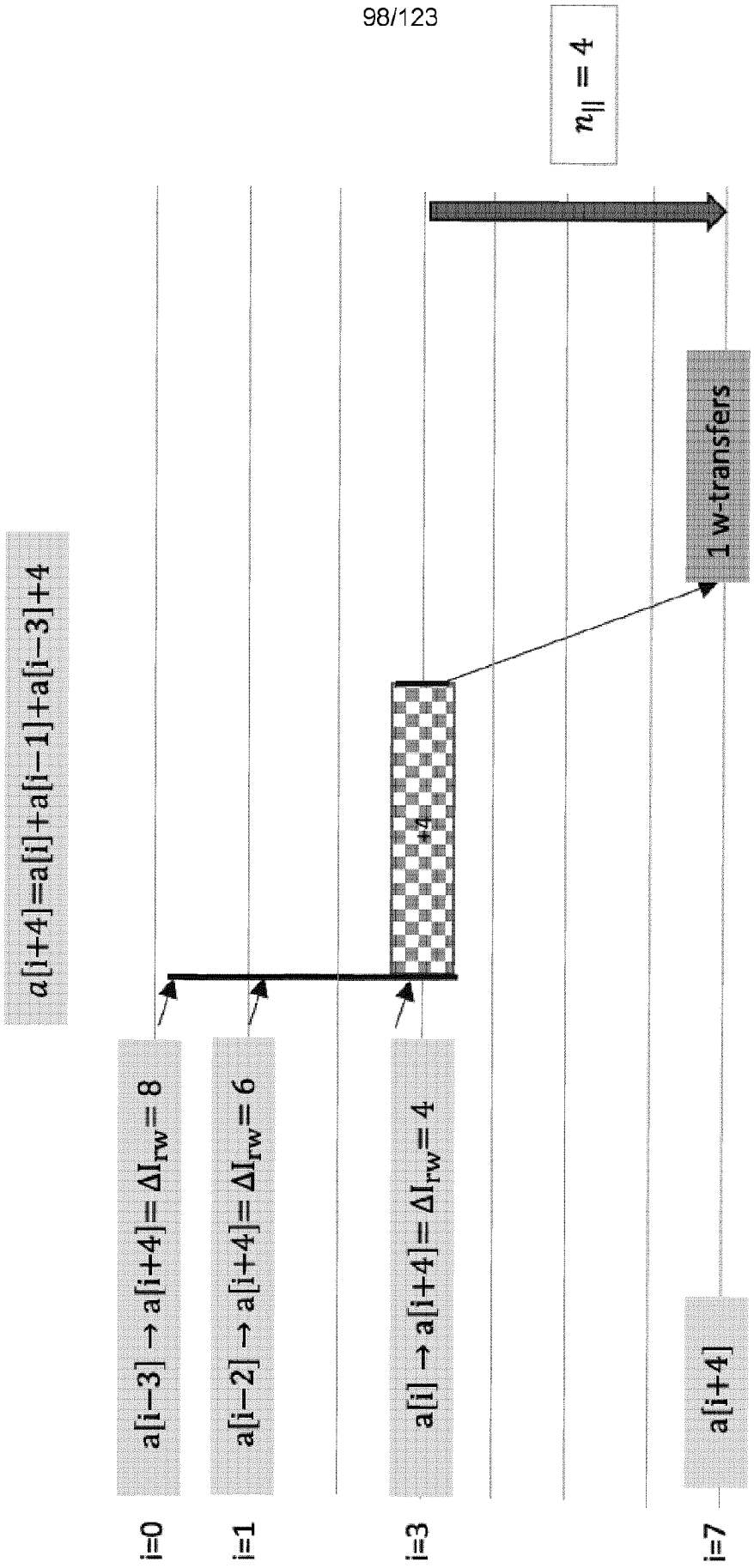

$n_{||} = 4$
1 w-transfers
a[i+4]=a[i]+a[i−1]+a[i−3]+4
a[i−3] → a[i+4]= ΔI_rw= 8
a[i−2] → a[i+4]= ΔI_rw= 6
a[i] → a[i+4]= ΔI_rw= 4
a[i+4]
i=0
i=1
i=3
i=7

Fig. 98

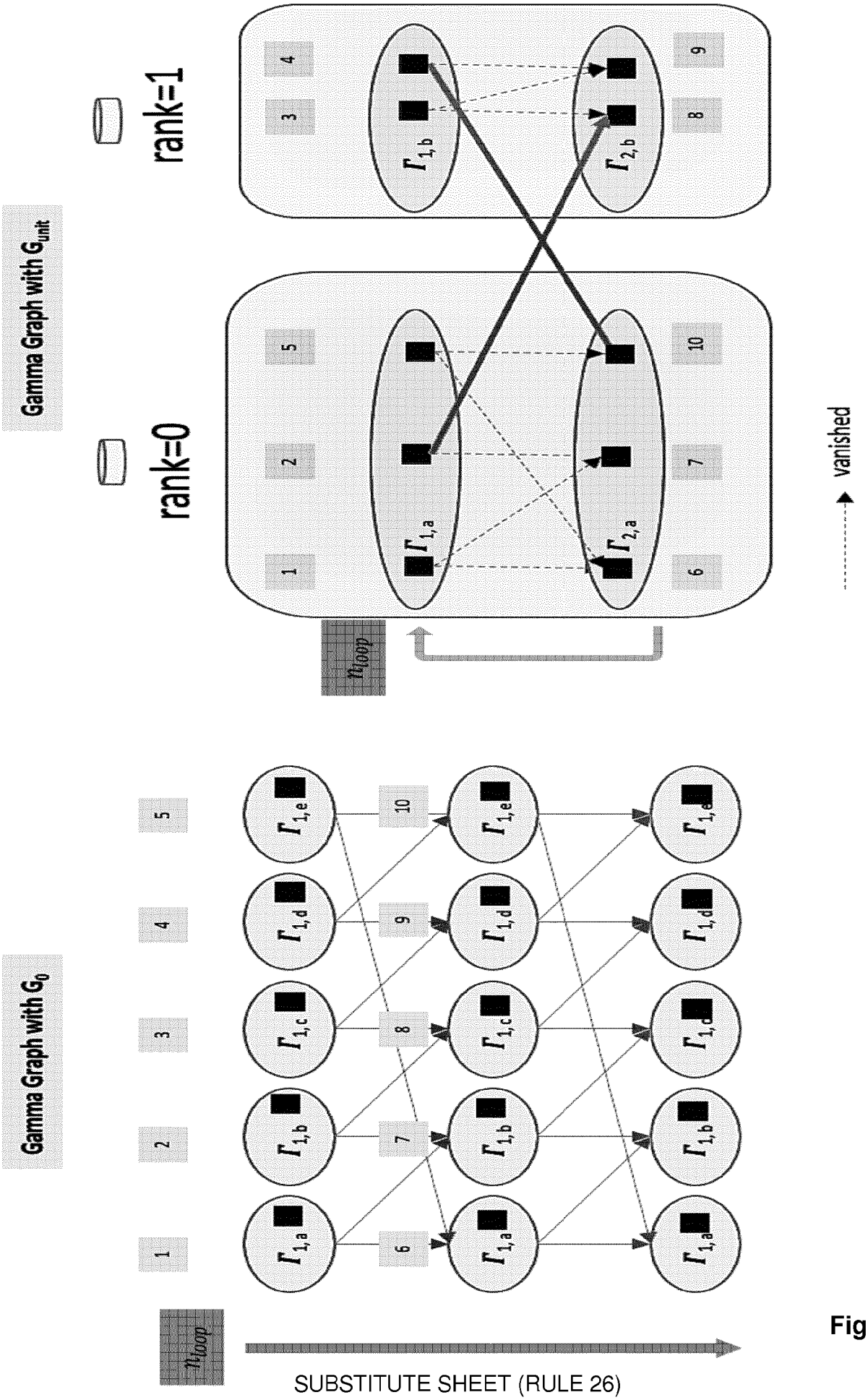

Gamma Graph with G_{unit}
rank=1
rank=0
$\Gamma_{1,a}$
$\Gamma_{1,b}$
$\Gamma_{2,a}$
$\Gamma_{2,b}$
1
2
3
4
5
6
7
8
9
10
n_{loop}
vanished
Gamma Graph with G_0
$\Gamma_{1,c}$
$\Gamma_{1,d}$
$\Gamma_{1,e}$

Fig. 99

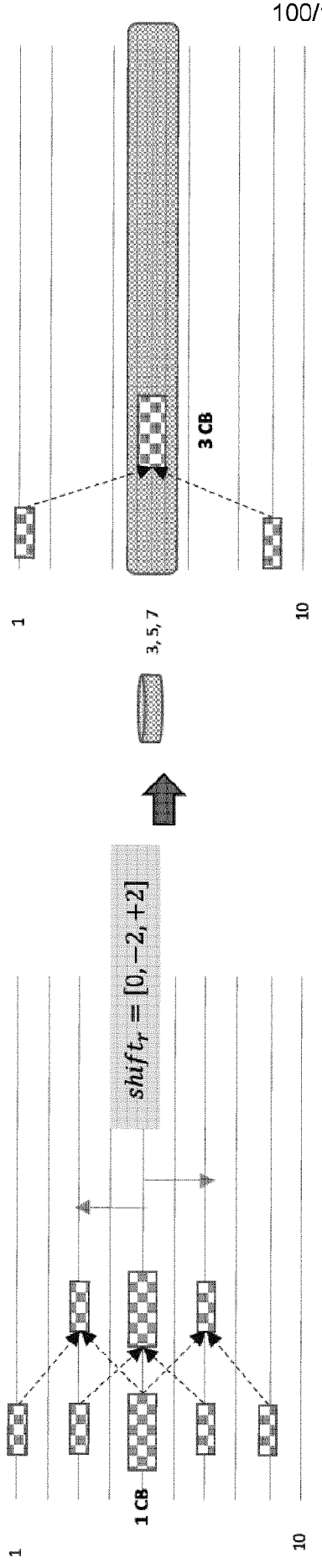


Fig. 100

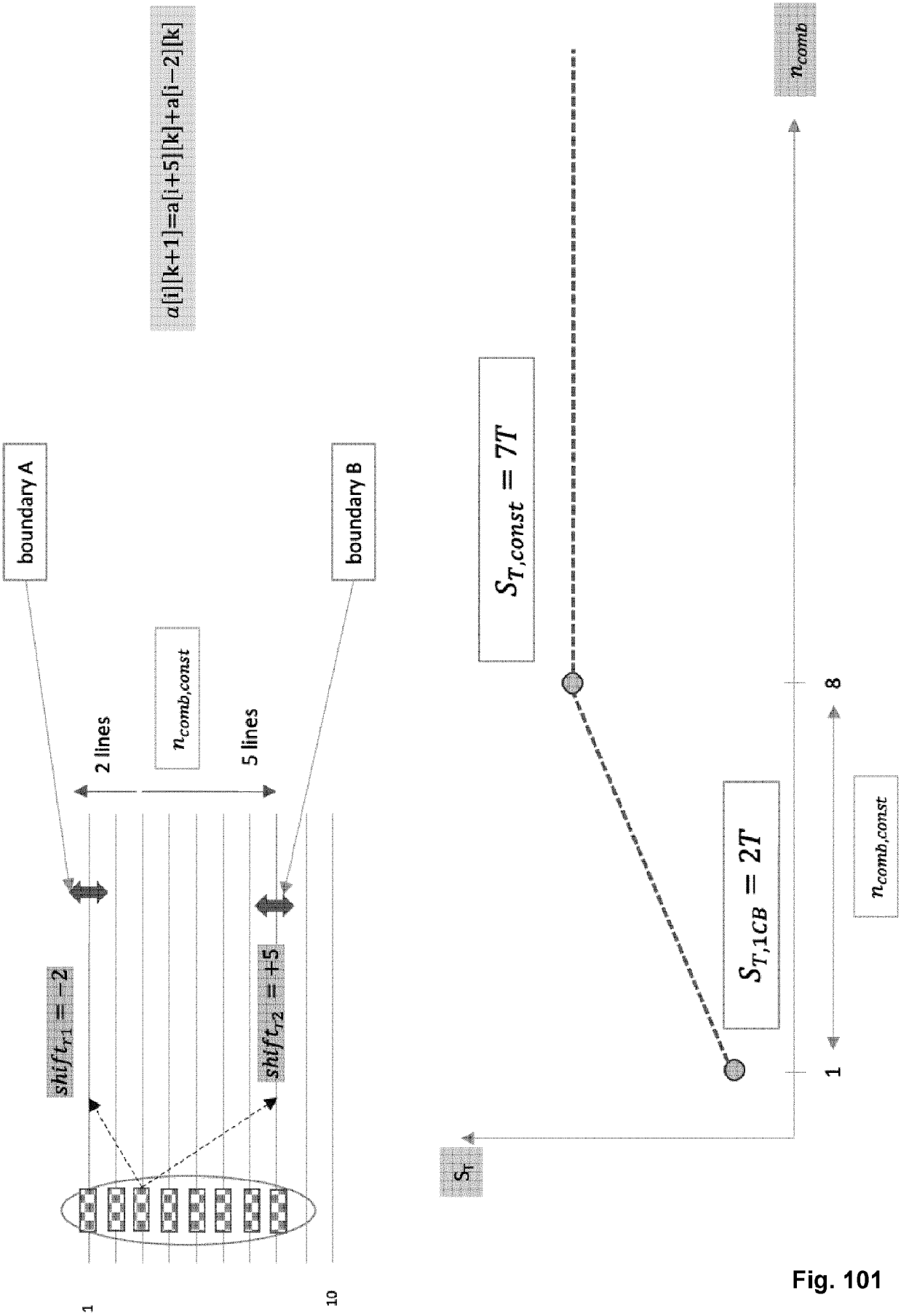

a[i][k+1]=a[i+5][k]+a[i−2][k]
boundary A
boundary B
$n_{comb,const}$
2 lines
5 lines
$shift_{r1} = -2$
$shift_{r2} = +5$
1
10
$S_{T,const} = 7T$
$S_{T,1CB} = 2T$
S_T
n_{comb}
8
1
$n_{comb,const}$

Fig. 101

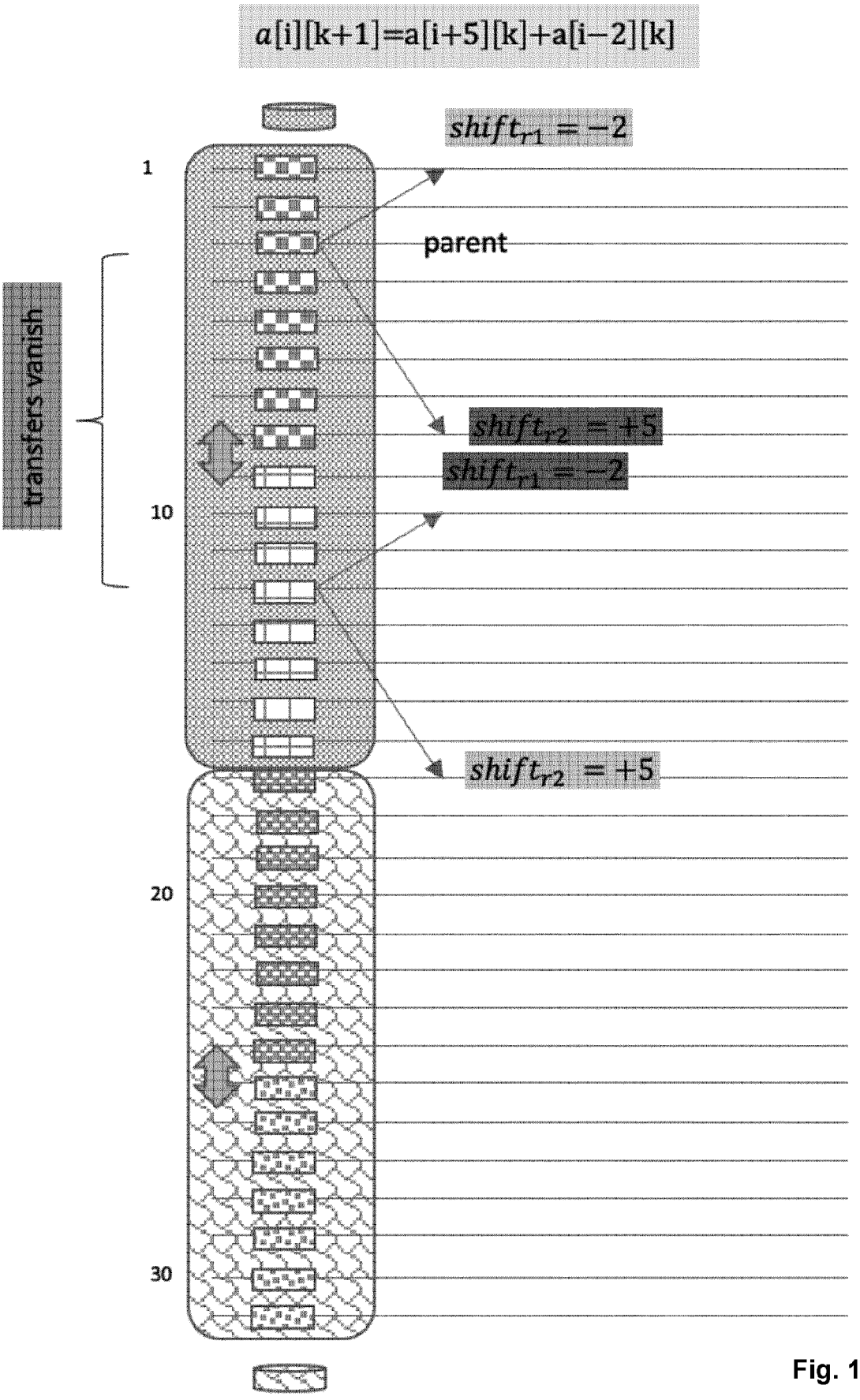

a[i][k+1]=a[i+5][k]+a[i−2][k]
$shift_{r1} = -2$
parent
transfers vanish
$shift_{r2} = +5$
$shift_{r1} = -2$
$shift_{r2} = +5$
1
10
20
30

Fig. 102

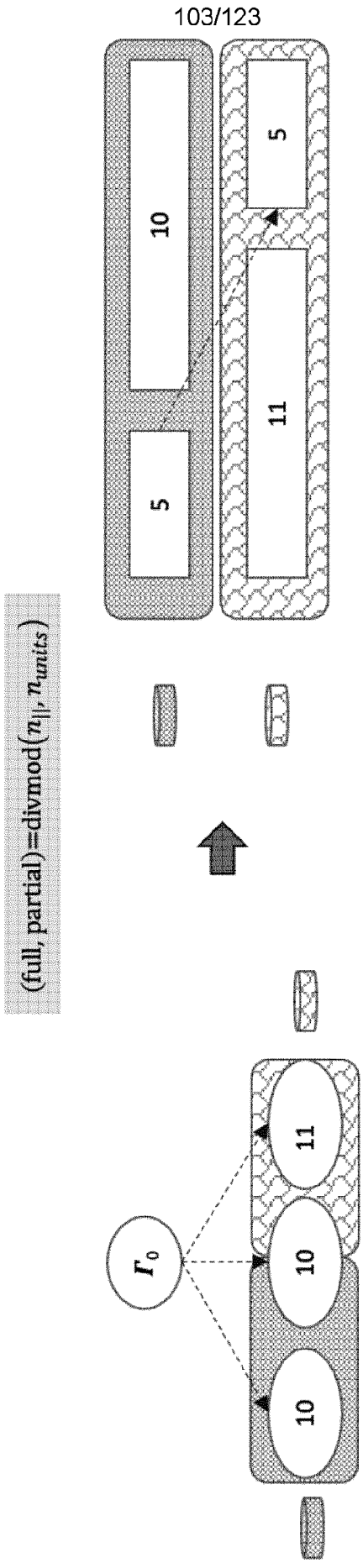


Fig. 103

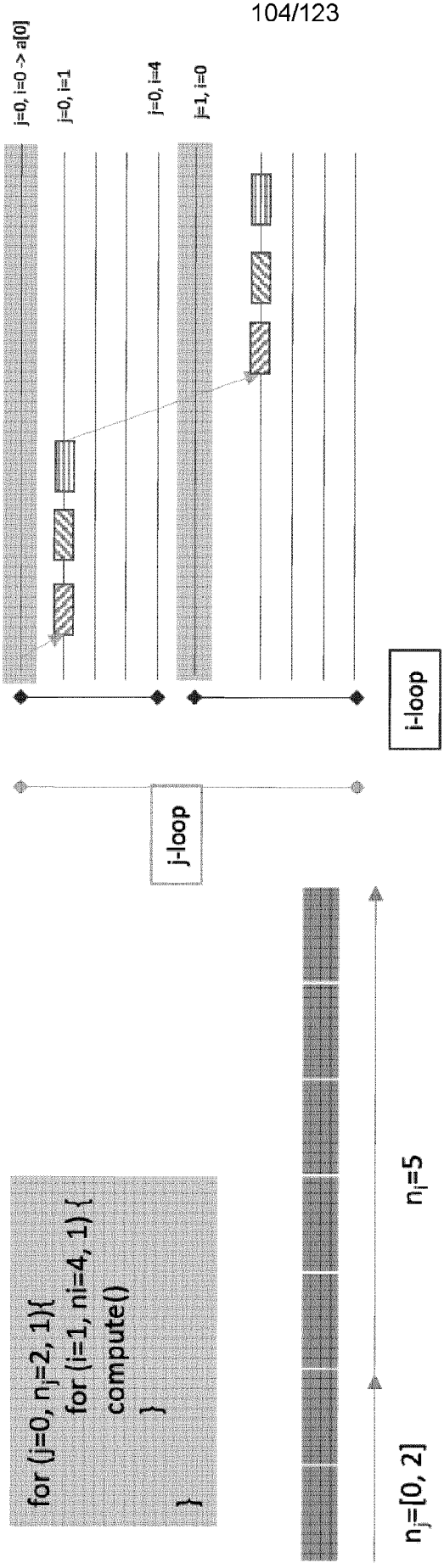

j=0, i=0 -> a[0]
j=0, i=1
j=0, i=4
j=1, i=0
i-loop
j-loop
for (j=0, nj=2, 1){
for (i=1, ni=4, 1) {
compute()
}
}
ni=5
nj=[0, 2]

Fig. 104

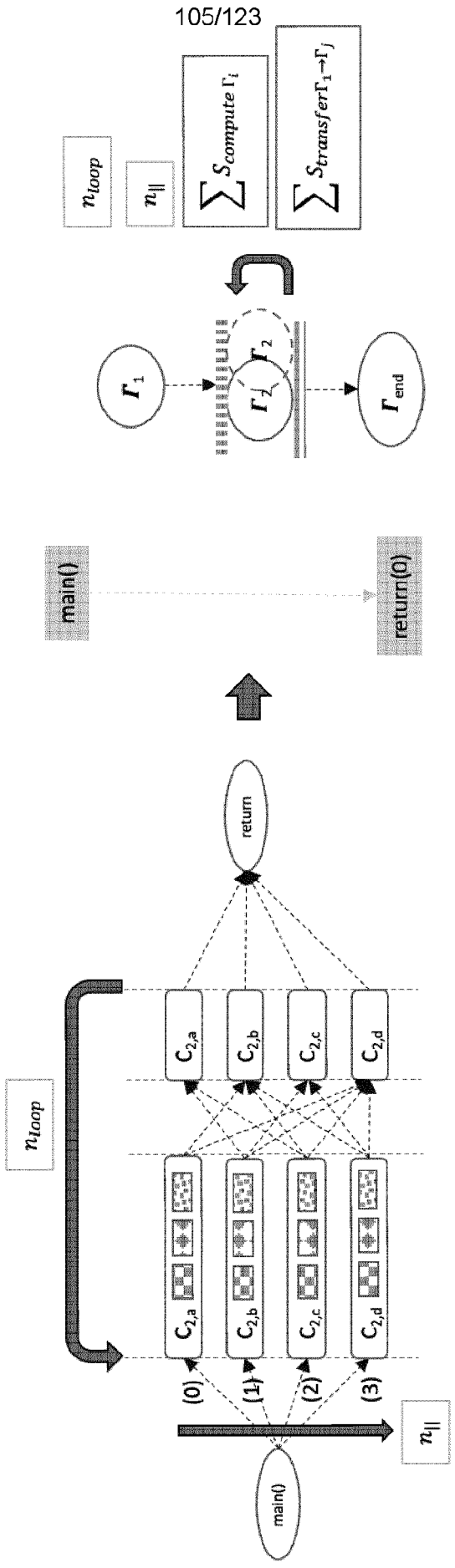

n_{loop}
$n_{||}$
$\sum S_{compute\,\Gamma_i}$
$\sum S_{transfer\Gamma_1 \to \Gamma_j}$
Γ_1
Γ_2
Γ_2
Γ_{end}
main()
return(0)
return
$C_{2,a}$
$C_{2,b}$
$C_{2,c}$
$C_{2,d}$
n_{loop}
$C_{2,a}$
$C_{2,b}$
$C_{2,c}$
$C_{2,d}$
(0)
(1)
(2)
(3)
$n_{||}$
main()

Fig. 105

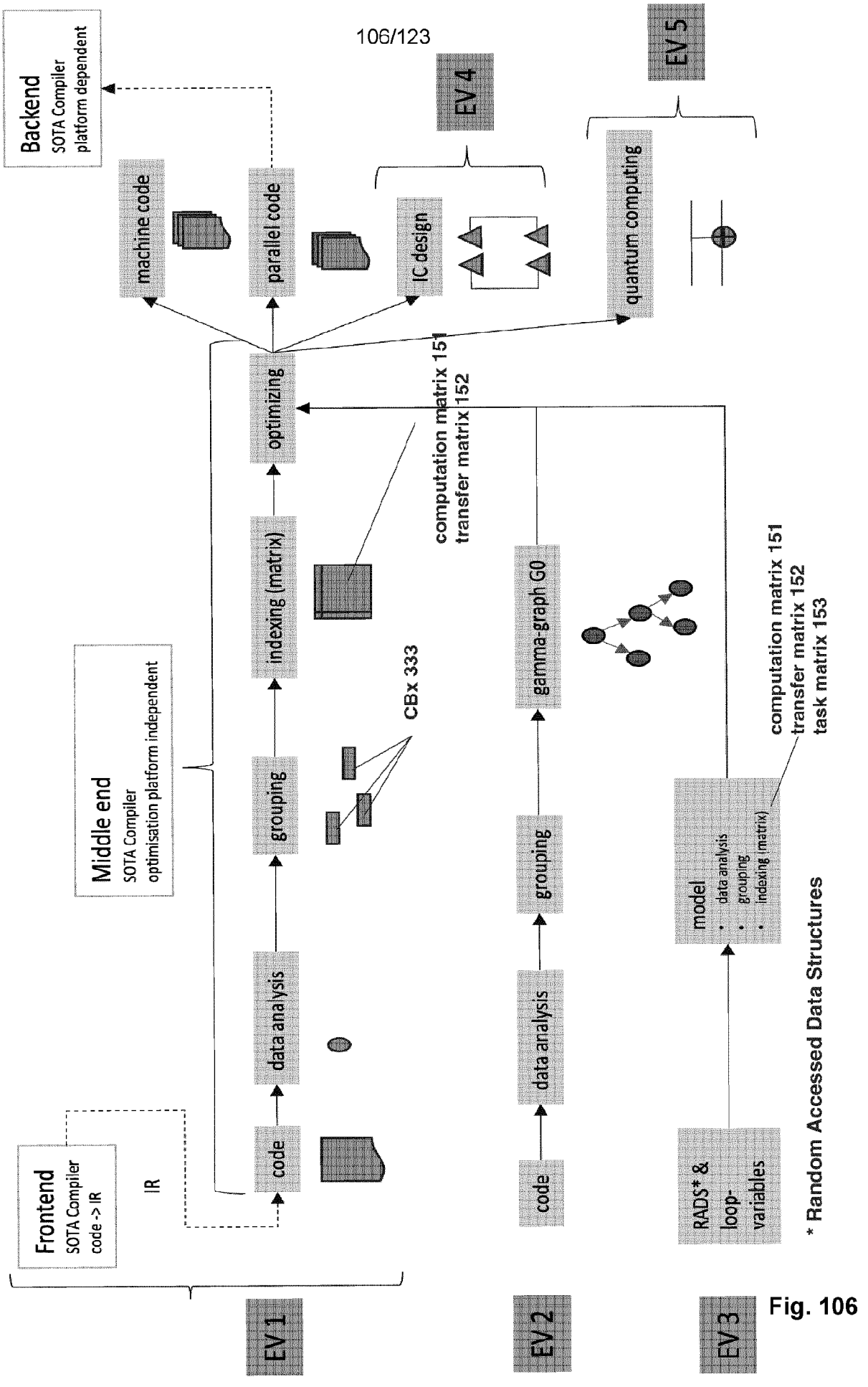

Backend
SOTA Compiler
platform dependent
machine code
parallel code
IC design
quantum computing
EV 4
EV 5
optimizing
computation matrix 151
transfer matrix 152
Middle end
SOTA Compiler
optimisation platform independent
indexing (matrix)
grouping
data analysis
code
CBx 333
gamma-graph G0
grouping
data analysis
code
model
• data analysis
• grouping
• indexing (matrix)
computation matrix 151
transfer matrix 152
task matrix 153
RADS* &
loop-
variables
Frontend
SOTA Compiler
code -> IR
IR
EV 1
EV 2
EV 3
* Random Accessed Data Structures

Fig. 106

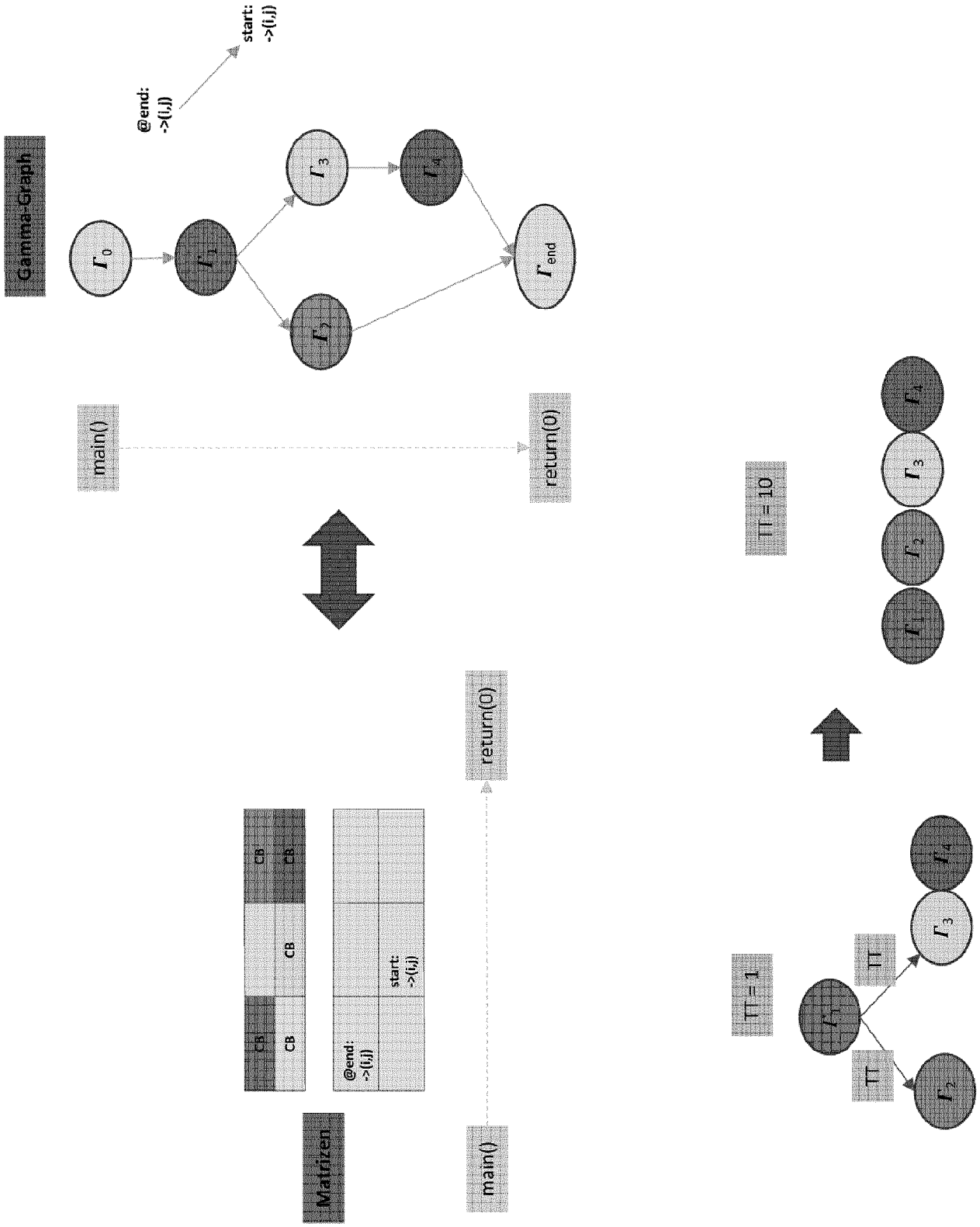
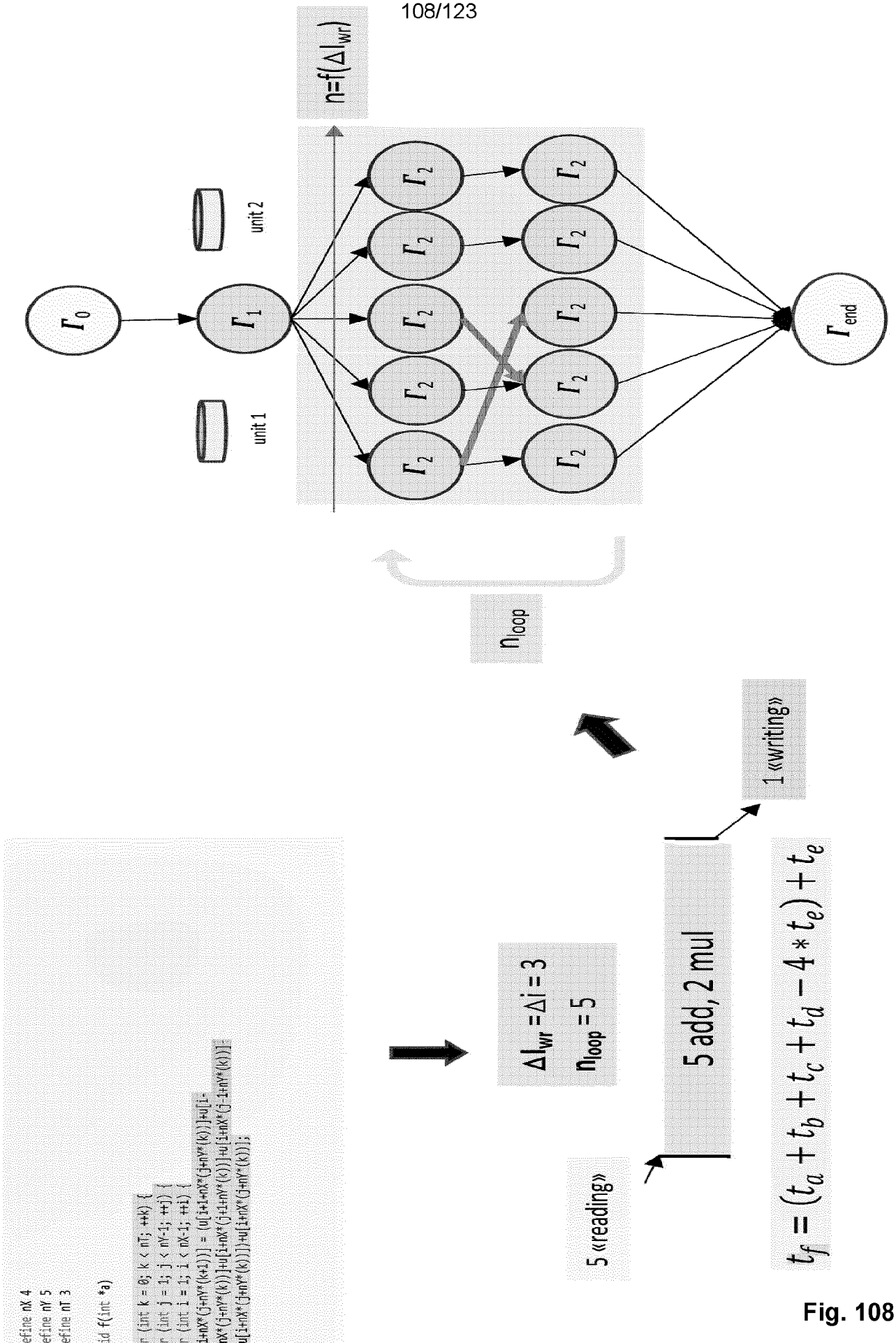

Gamma-Graph
@end:
->(i,j)
start:
->(i,j)
main()
return(0)
Matrizen
CB
@end:
->(i,j)
start:
->(i,j)
return(0)
main()
TT = 10
TT = 1
TT
TT

Fig. 107




```
#define nX 4
#define nY 5
#define nT 3
    void f(int *a)
    {
    for (int k = 0; k < nT; ++k) {
    for (int j = 1; j < nY-1; ++j) {
    for (int i = 1; i < nX-1; ++i) {
    u[i+nX*(j+nY*(k+1))] = (u[i+1+nX*(j+nY*(k))]+u[i-
    1+nX*(j+nY*(k))]+u[i+nX*(j+1+nY*(k))]+u[i+nX*(j-1+nY*(k))]+
    4*u[i+nX*(j+nY*(k))])+u[i+nX*(j+nY*(k))];
    }
    }
    }
    }
```

Fig. 108

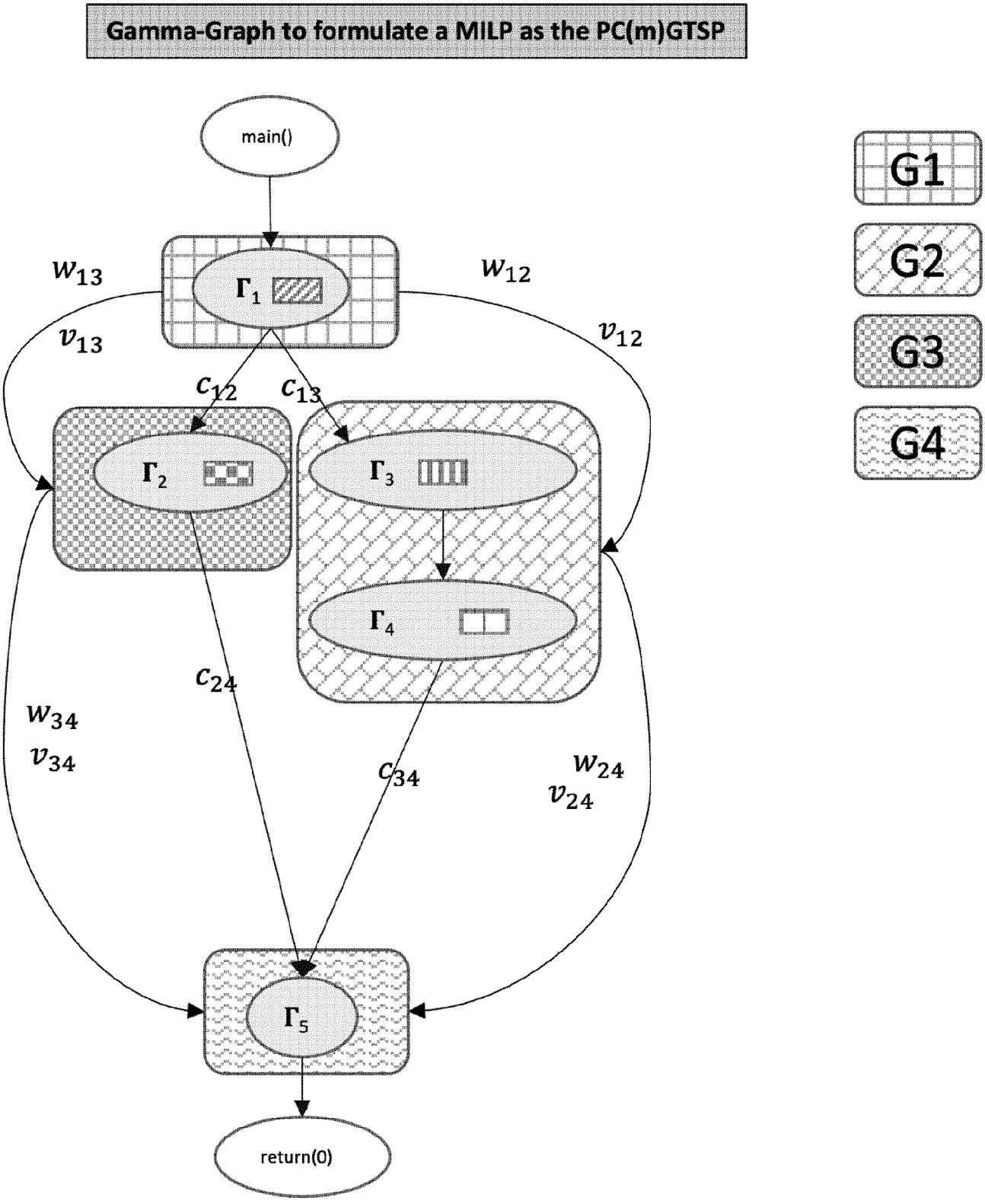

Gamma-Graph to formulate a MILP as the PC(m)GTSP
main()
Γ_1
w_{13}
v_{13}
w_{12}
v_{12}
c_{12}
c_{13}
Γ_2
Γ_3
Γ_4
c_{24}
w_{34}
v_{34}
c_{34}
w_{24}
v_{24}
Γ_5
return(0)
G1
G2
G3
G4

Fig. 109a

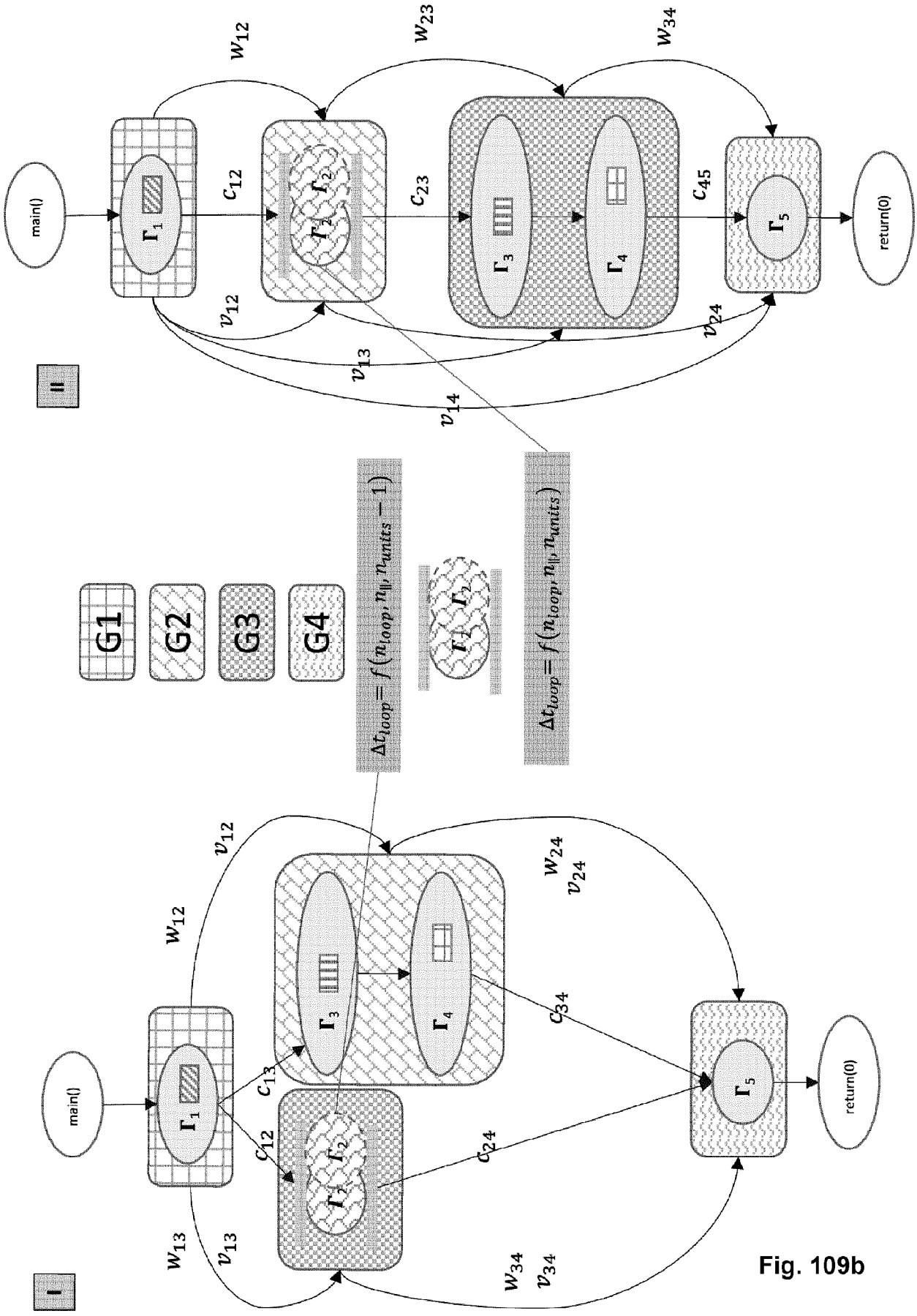

II
main()
return(0)
Γ_1
Γ_2
Γ_3
Γ_4
Γ_5
w_{12}
w_{23}
w_{34}
c_{12}
c_{23}
c_{45}
v_{12}
v_{13}
v_{14}
v_{24}
G1
G2
G3
G4
$\Delta t_{loop} = f(n_{loop}, n_{||}, n_{units} - 1)$
$\Delta t_{loop} = f(n_{loop}, n_{||}, n_{units})$
I
main()
return(0)
Γ_1
Γ_2
Γ_3
Γ_4
Γ_5
v_{12}
w_{12}
w_{24}
v_{24}
c_{34}
c_{13}
c_{12}
c_{24}
w_{13}
v_{13}
w_{34}
v_{34}

Fig. 109b

All transfer properties are known -> Size of variables
entity entity_name is
Port declaration;
end entity_name;
An entity declaration should start with 'entity' and end with 'end' keywords. The direction will be input, output or inout.
In Port can be read
Out Port can be written
Input Port can be read and written
Buffer Port can be read and written, it can have only one source.
Each gamma node comprises to a step in the program all serial instructions
Logic Operation – AND GATE
Symbol:
A
B
Q
x y z
0 0 0
0 1 0
1 0 0
1 1 1
VHDL Code:
Library ieee;
use ieee.std_logic_1164.all;
entity and1 is
port(x,y:in bit; z:out bit);
end and1;
architecture virat of and1 is
begin
z<= and y;
end virat;
Gamma-Graph
Γ0
Γ1
Γ2
Γ3
Γ4
Γend
main()
return(0)
#include<stdio.h>
int main ()
{
int n = 8, i;
int a[n]={0};
for (i=0; i<=n-3; i++) {
a[i+3]=a[i]+4;
}
return 0;
}

Fig. 110

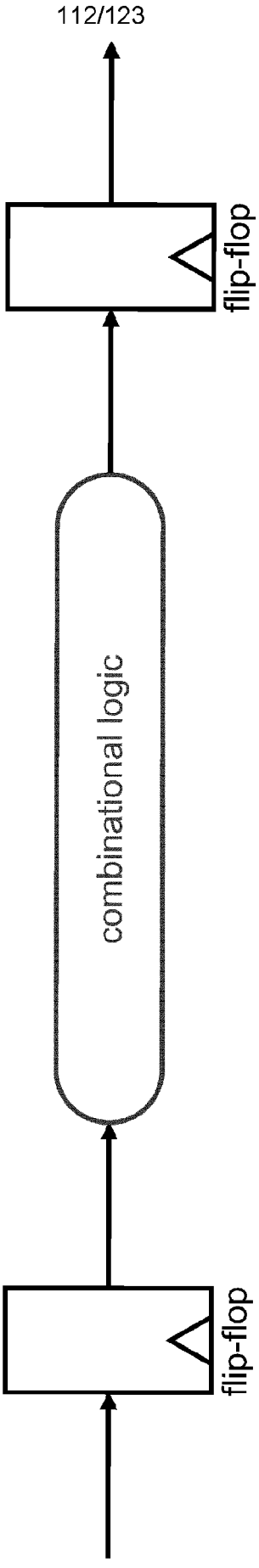


Fig. 111a

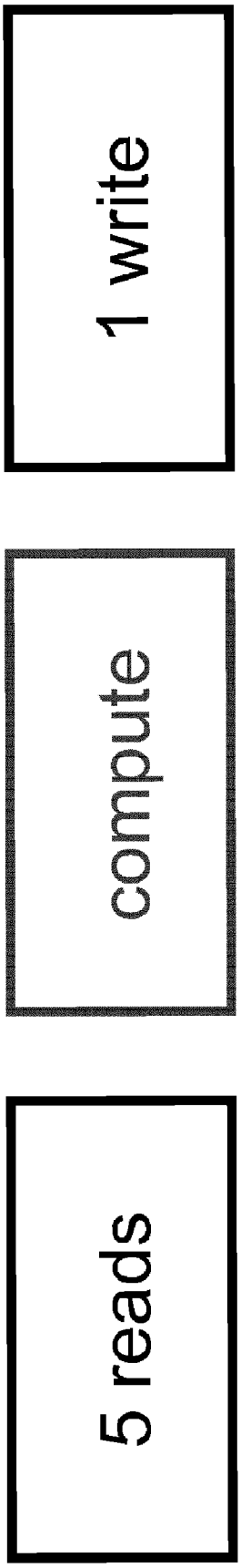


Fig. 111b

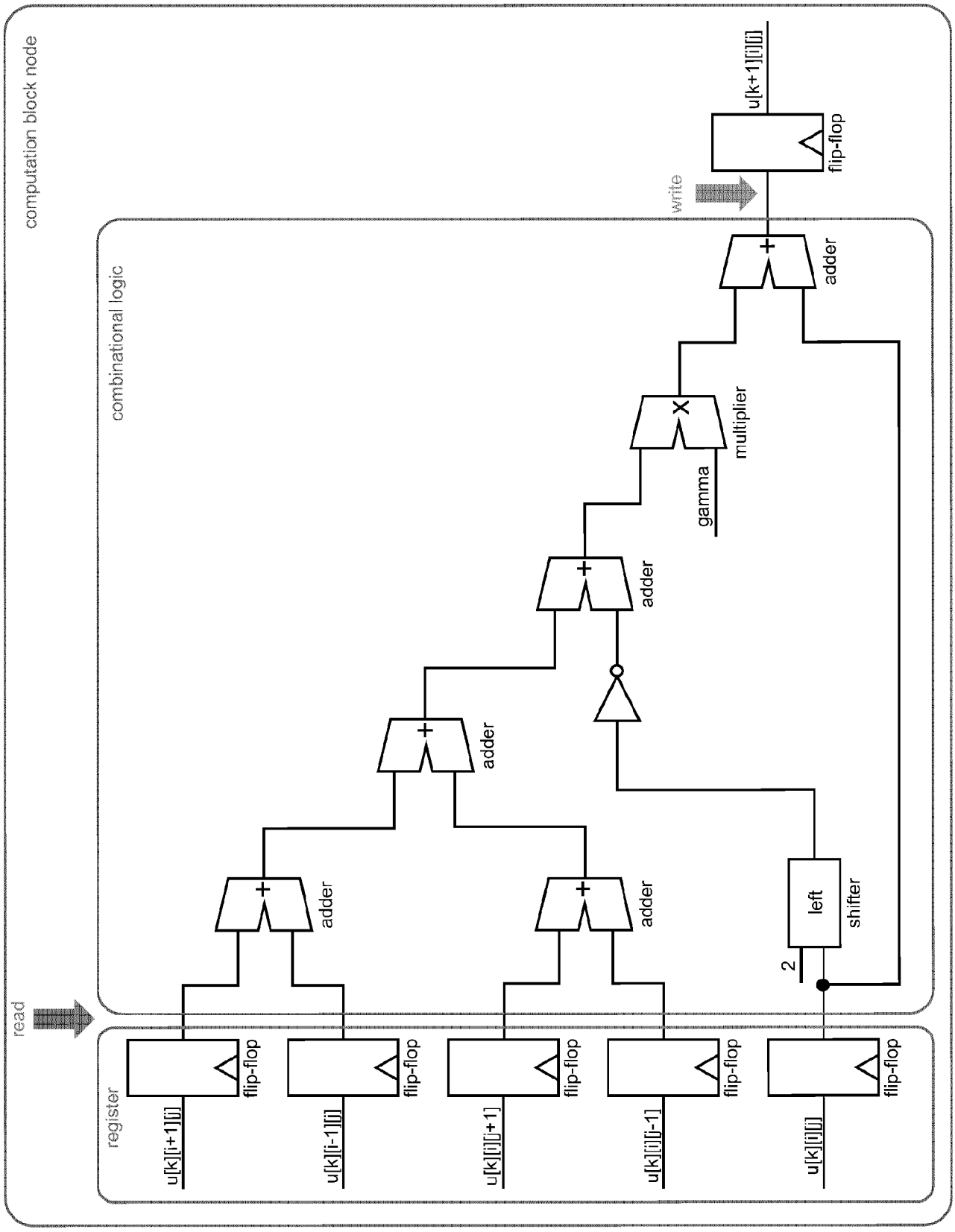

computation block node
read
register
flip-flop
u[k][i+1][j]
flip-flop
u[k][i-1][j]
flip-flop
u[k][i][j+1]
flip-flop
u[k][i][j-1]
flip-flop
u[k][i][j]
combinational logic
adder
adder
adder
adder
2
left
shifter
gamma
multiplier
adder
write
flip-flop
u[k+1][i][j]

Fig. 111c

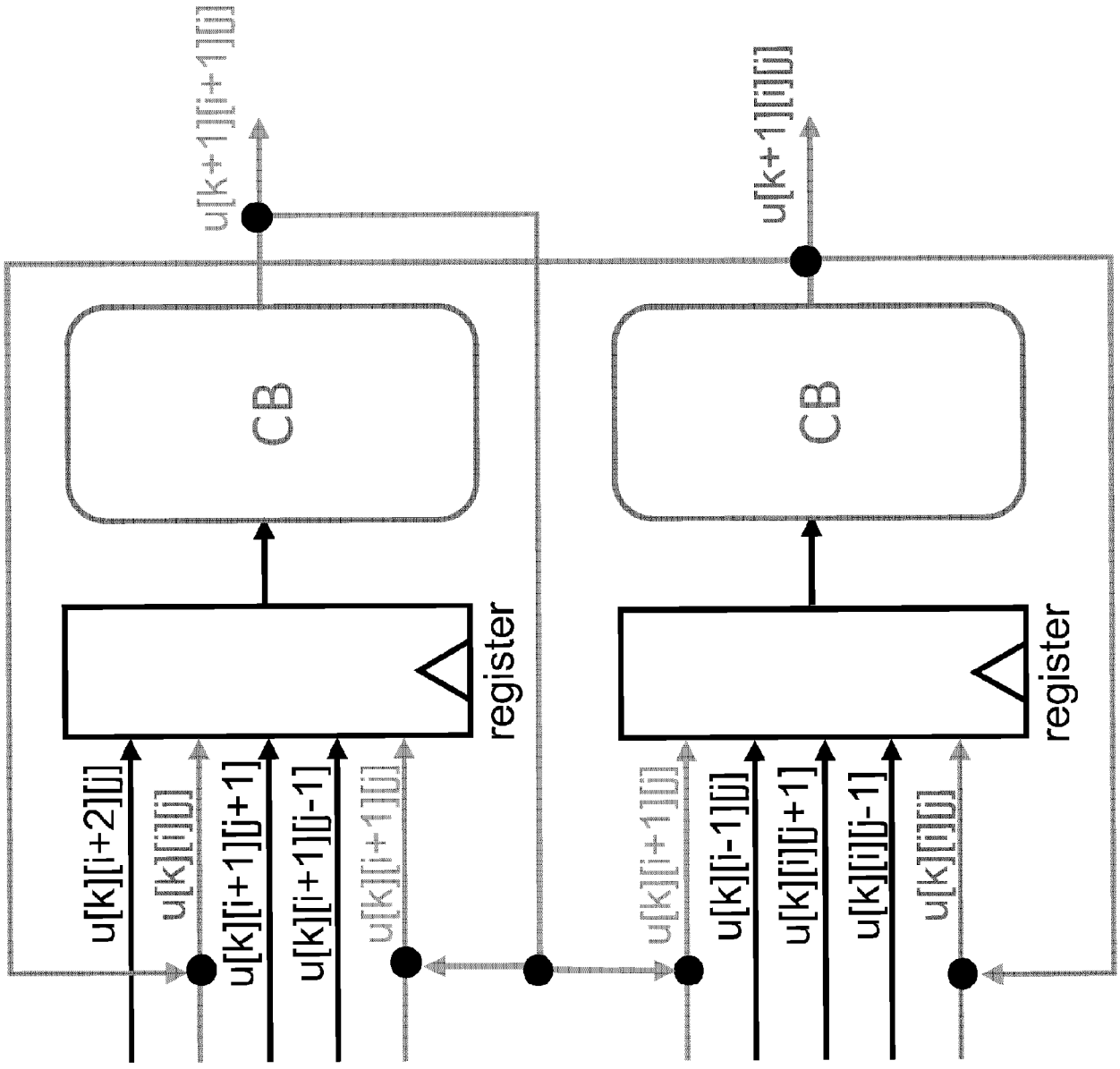

u[k+1][i+1][j]
u[k+1][i][j]
CB
CB
register
register
u[k][i+2][j]
u[k][i][j]
u[k][i+1][j+1]
u[k][i+1][j-1]
u[k][i+1][j]
u[k][i-1][j]
u[k][i][j+1]
u[k][i][j-1]
u[k][i][j]

Fig. 111d

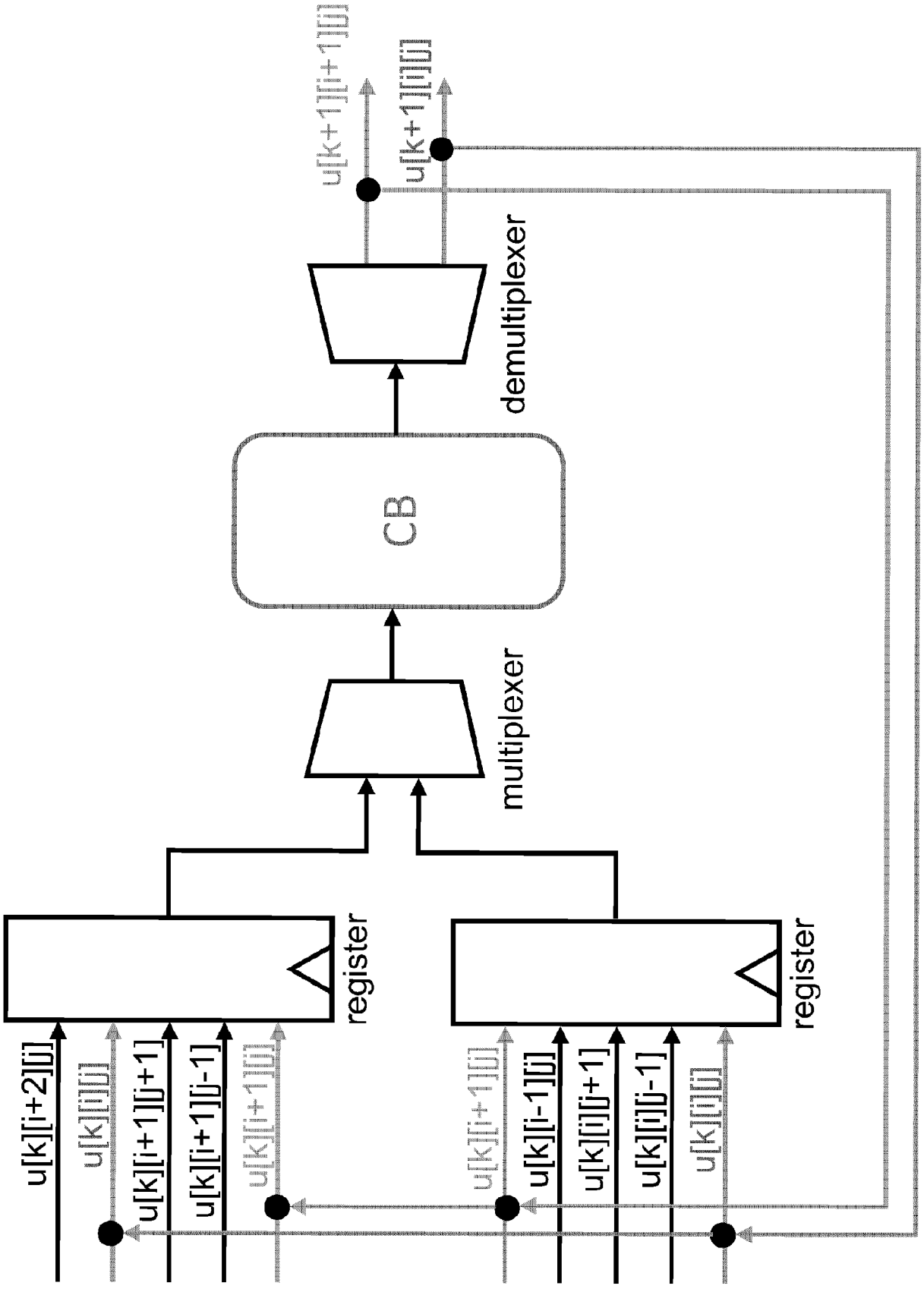

u[k+1][i+1][j]
u[k+1][i][j]
demultiplexer
CB
multiplexer
register
register
u[k][i+2][j]
u[k][i][j]
u[k][i+1][j+1]
u[k][i+1][j-1]
u[k][i-1][j]
u[k][i+1][j]
u[k][i-1][j]
u[k][i][j+1]
u[k][i][j-1]
u[k][i][j]

Fig. 111e

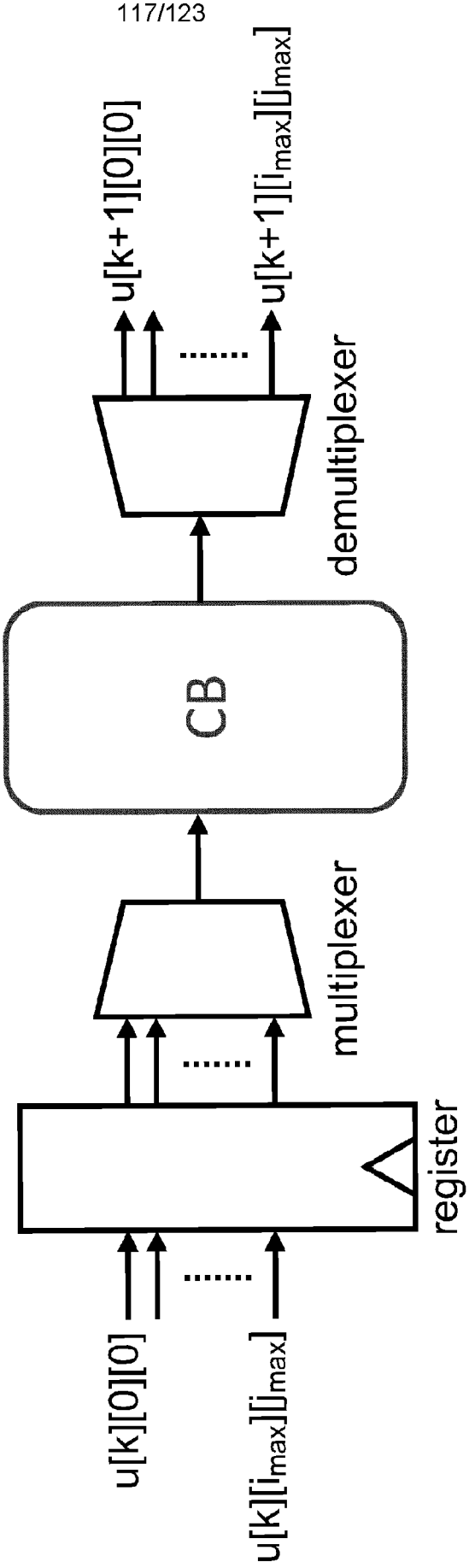

u[k][0][0]
u[k][imax][jmax]
register
multiplexer
CB
demultiplexer
u[k+1][0][0]
u[k+1][imax][jmax]

Fig. 111f

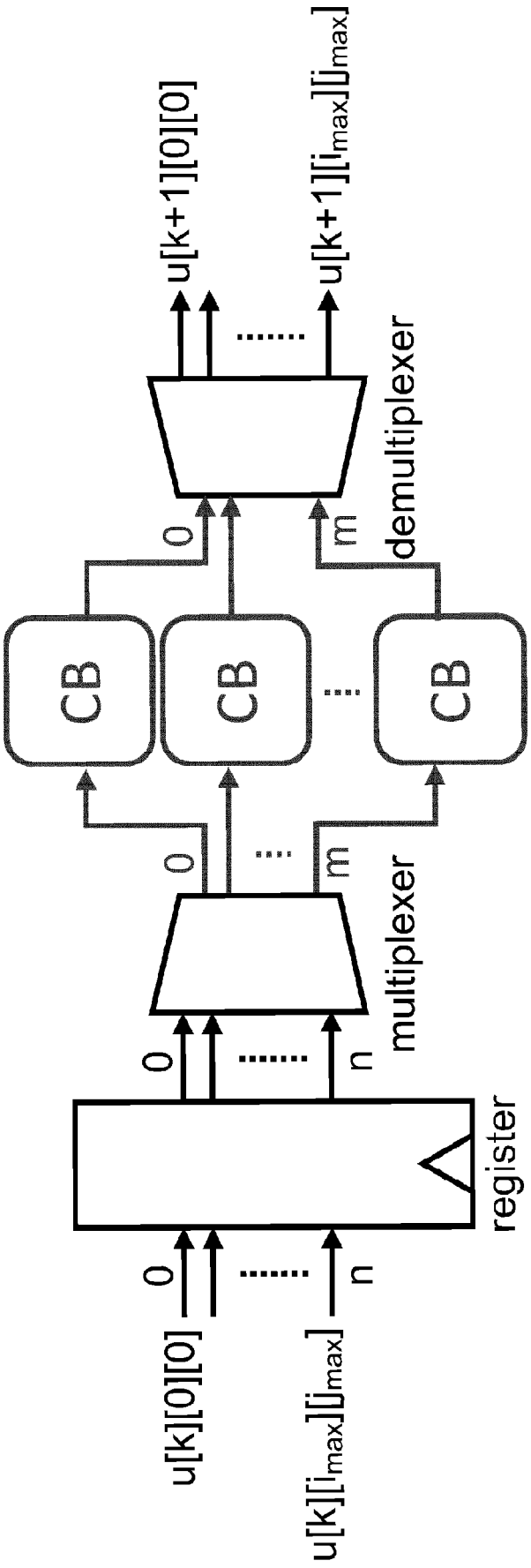

u[k][0][0]
u[k][i_max][j_max]
0
n
register
0
n
multiplexer
0
m
CB
CB
CB
demultiplexer
0
m
u[k+1][0][0]
u[k+1][i_max][j_max]

Fig. 111g

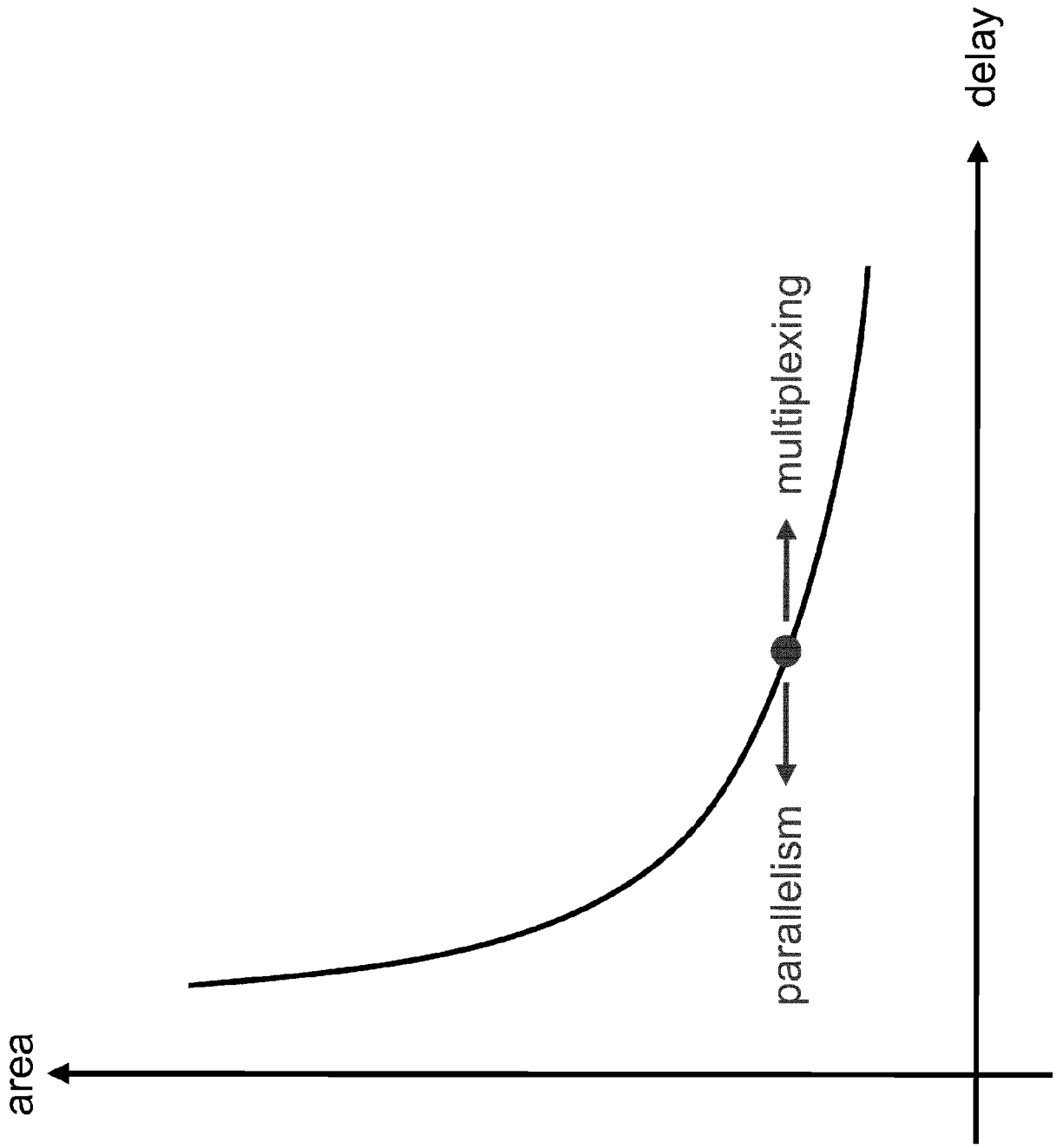


Fig. 111h

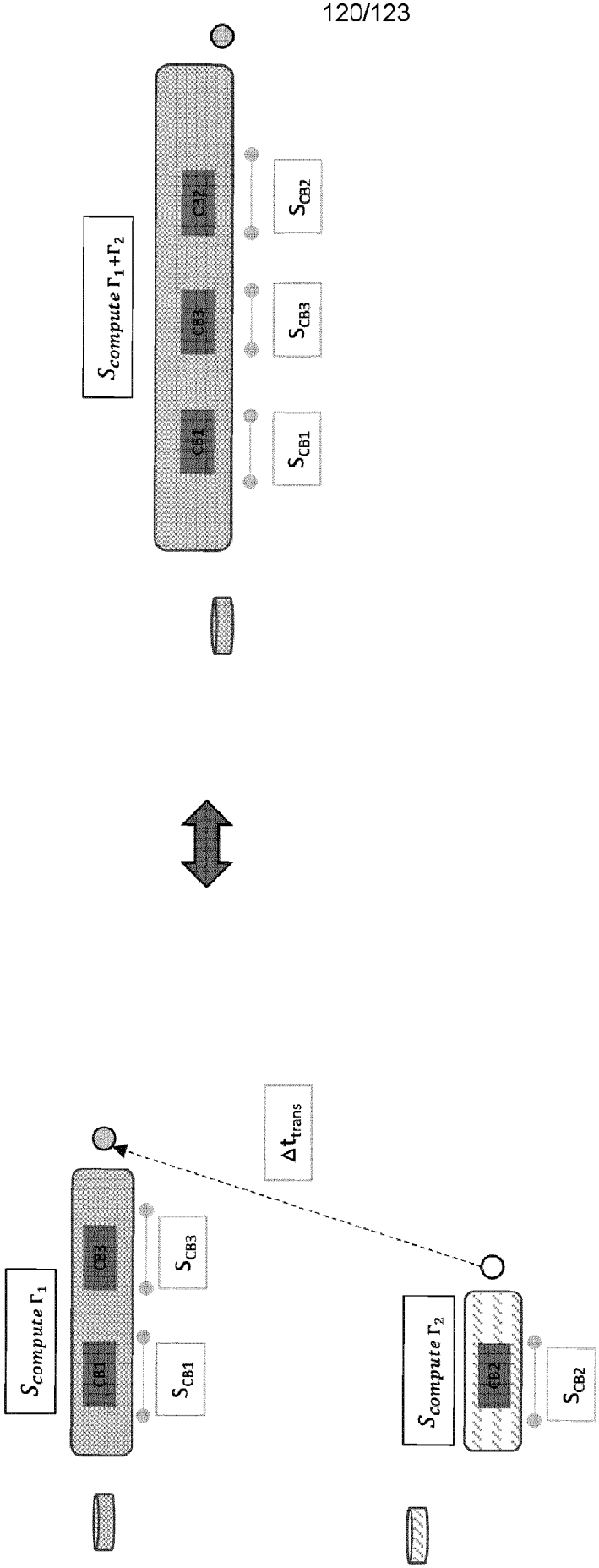

$S_{compute\ \Gamma_1+\Gamma_2}$
CB2
CB3
CB1
S_{CB2}
S_{CB3}
S_{CB1}
Δt_{trans}
$S_{compute\ \Gamma_1}$
CB3
CB1
S_{CB3}
S_{CB1}
$S_{compute\ \Gamma_2}$
CB2
S_{CB2}

Fig. 112

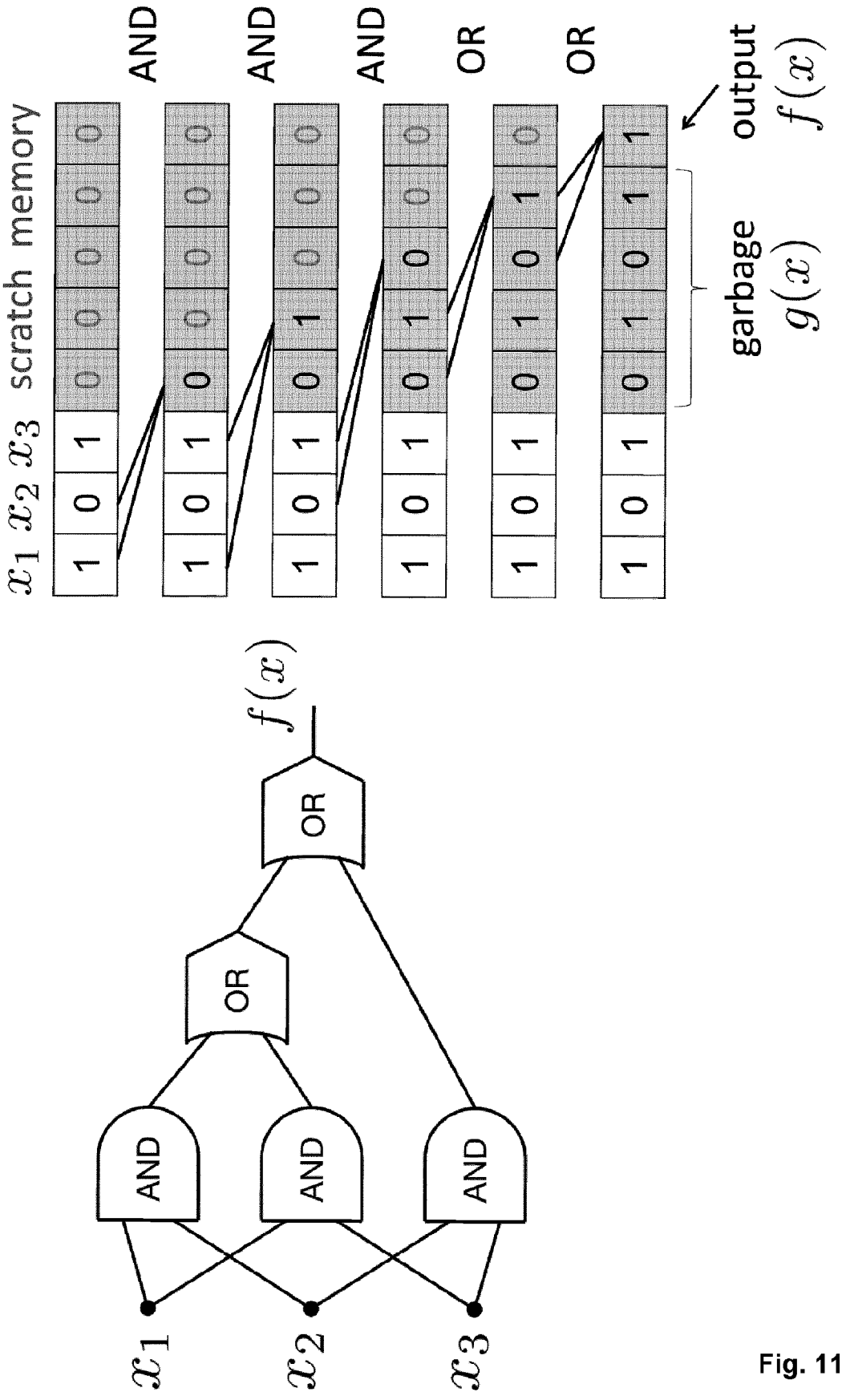

AND
AND
AND
OR
OR
output
$f(x)$
scratch memory
x_1 x_2 x_3
garbage
$g(x)$
$f(x)$
OR
OR
AND
AND
AND
x_1
x_2
x_3

Fig. 113

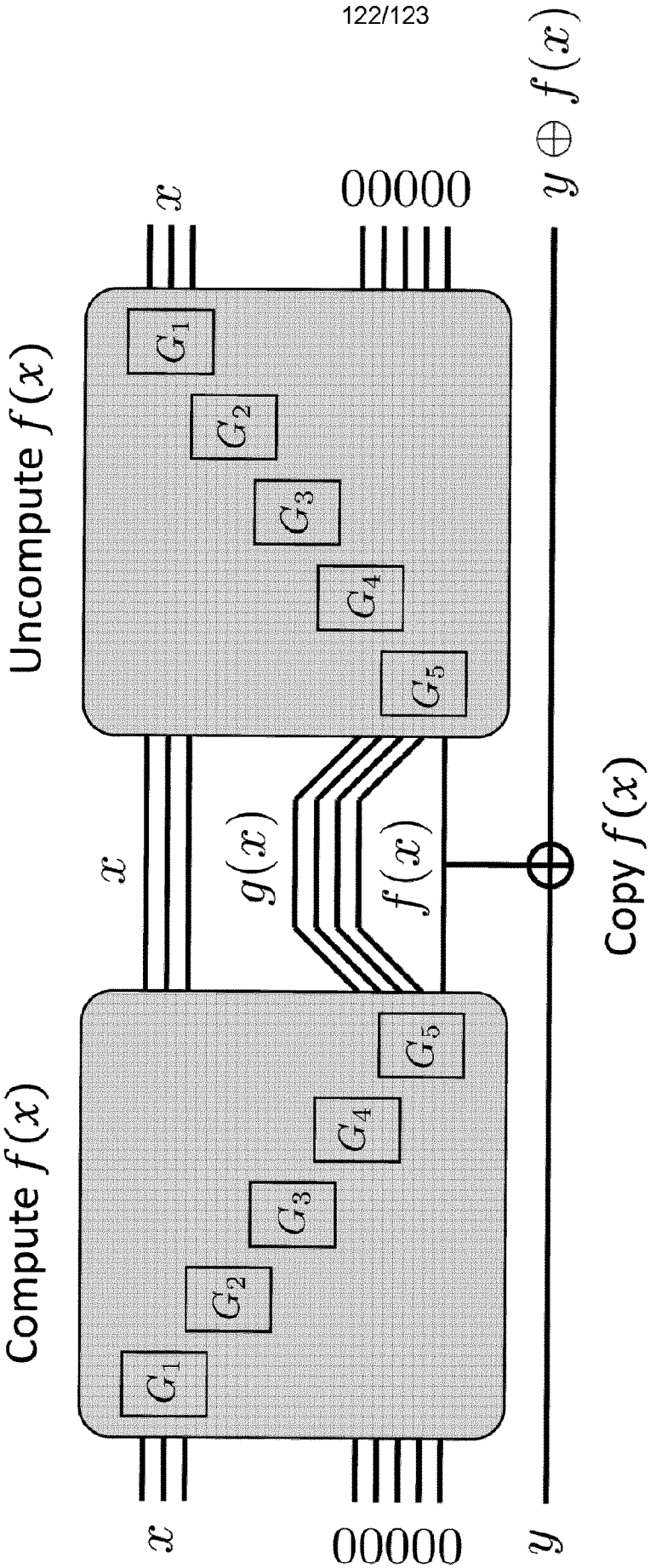

$y \oplus f(x)$
x
00000
Uncompute $f(x)$
G_1
G_2
G_3
G_4
G_5
x
$g(x)$
$f(x)$
Copy $f(x)$
Compute $f(x)$
G_5
G_4
G_3
G_2
G_1
x
00000
y

Fig. 114

Fig. 59: $t_w = \kappa * (t_a + t_b + t_c + t_d - 4 * t_e) + t_e$

Fig. 111c

combinational ic-circuit

Fig. 115