Paradigms for Process Interaction

Andrews, Chapter 09

```
module Manager
  type pair = (int index, double value);
  op getTask(result int row, len; result pair [*]elems);
  op putResult(int row, len; pair [*]elems);
body Manager
  int lengthA[n], lengthC[n];
  pair *elementsA[n], *elementsC[n];
  # matrix A is assumed to be initialized
  int nextRow = 0, tasksDone = 0;

  process manager {
    while (nextRow < n or tasksDone < n) {
      # more tasks to do or more results needed
      in getTask(row, len, elems) ->
          row = nextRow;
          len = lengthA[i];
          copy pairs in  *elementsA[i] to elems;
          nextRow++;
      [] putResult(row, len, elems) ->
          lengthC[row] = len;
          copy pairs in elems to *elementsC[row];
          tasksDone++;
      ni
    }
  }
end Manager
```

Figure 9.1 (a) Sparse matrix multiplication: Manager process.

```
process worker[w = 1 to numWorkers] {
  int lengthB[n];
  pair *elementsB[n];  # assumed to be initialized
  int row, lengthA, lengthC;
  pair *elementsA, *elementsC;
  int r, c, na, nb;  # used in computing
  double sum;        #   inner products
  while (true) {
    # get a row of A, then compute a row of C
    call getTask(row, lengthA, elementsA);
    lengthC = 0;
    for [i = 0 to n-1]
      INNER_PRODUCT(i);  # see body of text
    send putResult(row, lengthC, elementsC);
  }
}
```

Figure 9.1 (b) Sparse matrix multiplication: Worker processes.

```
module Manager
  op getTask(result double left, right);
  op putResult(double area);
body Manager
  process manager {
    double a, b;         # interval to integrate
    int numIntervals;    # number of intervals to use
    double width = (b-a)/numIntervals;
    double x = a, totalArea = 0.0;
    int tasksDone = 0;
    while (tasksDone < numIntervals) {
      in getTask(left, right) st x < b ->
          left = x; x += width; right = x;
      [] putResult(area) ->
          totalArea += area;
          tasksDone++;
      ni
    }
    print the result totalArea;
  }
end Manager

double f() { ... }        # function to integrate
double quad(...) { ... }  # adaptive quad function

process worker[w = 1 to numWorkers] {
  double left, right, area = 0.0;
  double fleft, fright, lrarea;
  while (true) {
    call getTask(left, right);
    fleft = f(left); fright = f(right);
    lrarea = (fleft + fright) * (right - left) / 2;
    # calculate area recursively as shown in Section 1.5
    area = quad(left, right, fleft, fright, lrarea);
    send putResult(area);
  }
}
```

Figure 9.2 Adaptive quadrature using manager/workers paradigm.

```
chan first[1:P](int edge[n]);    # for exchanging edges
chan second[1:P](int edge[n]);
chan answer[1:P](bool);          # for termination check

process Worker[w = 1 to P] {
  int stripSize = m/W;
  int image[stripSize+2,n];    # local values plus edges
  int label[stripSize+2,n];    #   from neighbors
  int change = true;
  initialize image[1:stripSize,*] and label[1:stripSize,*];

  # exchange edges of image with neighbors
  if (w != 1)
    send first[w-1](image[1,*]);       # to worker above
  if (w != P)
    send second[w+1](image[stripSize,*]);   # to below
  if (w != P)
    receive first[w](image[stripSize+1,*]); # from below
  if (w != 1)
    receive second[w](image[0,*]);   # from worker above

  while (change) {
    exchange edges of label with neighbors, as above;
    update label[1:stripSize,*] and set change to true if
      the value of the label changes;
    send result(change);        # tell coordinator
    receive answer[w](change);  # and get back answer
  }
}
```

Figure 9.3 (a) Region labeling: Worker processes.

```
chan result(bool);  # for results from workers

process Coordinator {
  bool chg, change = true;
  while (change) {
    change = false;
    # see if there has been a change in any strip
    for [i = 1 to P] {
      receive result(chg);
      change = change or chg;
    }
    # broadcast answer to every worker
    for [i = 1 to P]
      send answer[i](change);
  }
}
```

Figure 9.3 (b) Region labeling: Coordinator process.

```
chan exchange[1:n,1:n](int row, column, state);

process cell[i = 1 to n, j = 1 to n] {
  int state;   # initialize to dead or alive
  declarations of other variables;
  for [k = 1 to numGenerations] {
    # exchange state with 8 neighbors
    for [p = i-1 to i+1, q = j-1 to j+1]
      if (p != q)
        send exchange[p,q](i, j, state);
    for [p = 1 to 8] {
      receive exchange[i,j](row, column, value);
      save value of neighbor's state;
    }
    update local state using rules in text;
  }
}
```

Figure 9.4 The Game of Life.

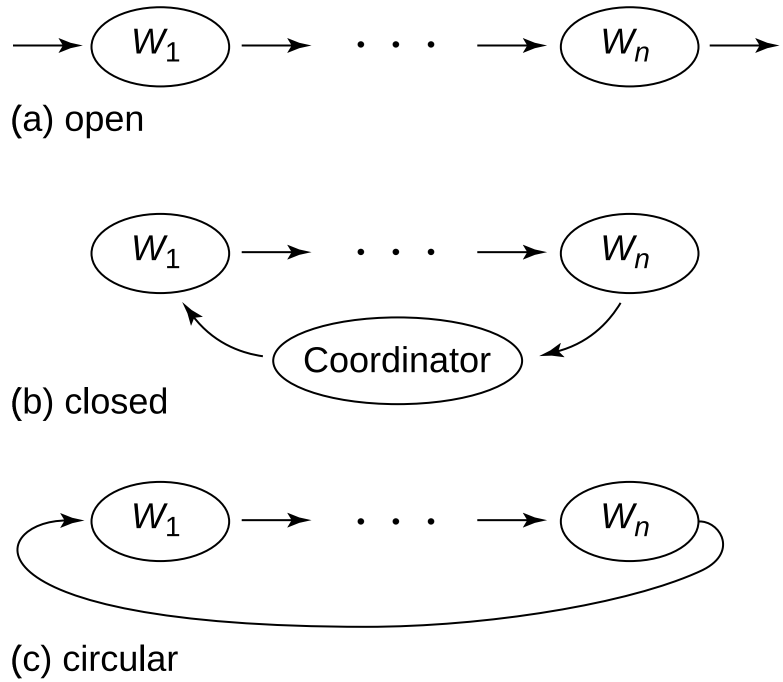


Figure 9.5 Pipeline structures for parallel computing.

```
chan vector[n](double v[n]);  # messages to workers
chan result(double v[n]);     # rows of c to coordinator

process Coordinator {
  double a[n,n], b[n,n], c[n,n];
  initialize a and b;
  for [i = 0 to n-1]         # send all rows of a
    send vector[0](a[i,*]);
  for [i = 0 to n-1]         # send all columns of b
    send vector[0](b[*,i]);
  for [i = n-1 to 0]         # receive rows of c
    receive result(c[i,*]);  # in reverse order
}
```

Figure 9.6 (a) Matrix multiplication pipeline: Coordinator process.

```
process Worker[w = 0 to n-1] {
  double a[n], b[n], c[n];  # my row or column of each
  double temp[n];           # used to pass vectors on
  double total;       # used to compute inner product

  # receive rows of a; keep first and pass others on
  receive vector[w](a);
  for [i = w+1 to n-1] {
    receive vector[w](temp); send vector[w+1](temp);
  }

  # get columns and compute inner products
  for [j = 0 to n-1] {
    receive vector[w](b);  # get a column of b
    if (w < n-1)    # if not last worker, pass it on
      send vector[w+1](b);
    total = 0.0;
    for [k = 0 to n-1]     # compute one inner product
      total += a[k] * b[k];
    c[j] = total;          # put total into c
  }

  # send my row of c to next worker or coordinator
  if (w < n-1)
    send vector[w+1](c);
  else
    send result(c);
  # receive and pass on earlier rows of c
  for [i = 0 to w-1] {
    receive vector[w](temp);
    if (w < n-1)
      send vector[w+1](temp);
    else
      send result(temp);
  }
}
```

Figure 9.6 (b) Matrix multiplication pipeline: Worker processes.

```
chan left[1:n,1:n](double);  # for circulating a left
chan up[1:n,1:n](double);    # for circulating b up

process Worker[i = 1 to n, j = 1 to n] {
  double aij, bij, cij;
  int LEFT1, UP1, LEFTI, UPJ;
  initialize above values;

  # shift values in aij circularly left i columns
  send left[i,LEFTI](aij); receive left[i,j](aij);
  # shift values in bij circularly up j rows
  send up[UPJ,j](bij); receive up[i,j](bij);
  cij = aij * bij;

  for [k = 1 to n-1] {
    # shift aij left 1, bij up 1, then multiply and add
    send left[i,LEFT1](aij); receive left[i,j](aij);
    send up[UP1,j](bij); receive up[i,j](bij);
    cij = cij + aij*bij;
  }
}
```

Figure 9.7 Matrix multiplication by blocks.

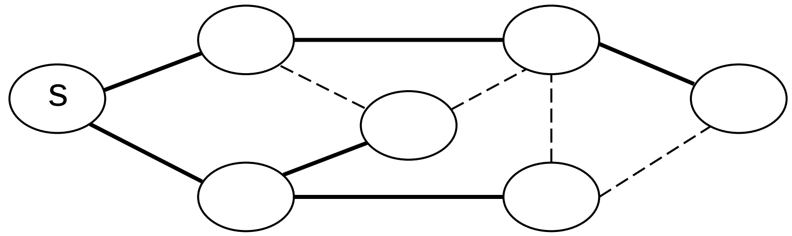


Figure 9.8 A spanning tree of a network of nodes.

```
type graph = bool [n,n];
chan probe[n](graph spanningTree; message m);

process Node[p = 0 to n-1] {
  graph t; message m;
  receive probe[p](t, m);
  for [q = 0 to n-1 st q is a child of p in t]
    send probe[q](t, m);
}

process Initiator {  # executed on source node S
  graph topology = network topology;
  graph t = spanning tree of topology;
  message m = message to broadcast;
  send probe[S](t, m);
}
```

Figure 9.9 Network broadcast using a spanning tree.

```
chan probe[n](message m);

process Node[p = 1 to n] {
  bool links[n] = neighbors of node p;
  int num = number of neighbors;
  message m;
  receive probe[p](m);
  # send m to all neighbors
  for [q = 0 to n-1 st links[q]]
    send probe[q](m);
  # receive num-1 redundant copies of m
  for [q = 1 to num-1]
    receive probe[p](m);
}

process Initiator {  # executed on source node S
  message m = message to broadcast;
  send probe[S](m);
}
```

Figure 9.10 Broadcast using neighbor sets.

```
type graph = bool [n,n];
chan probe[n](int sender);
chan echo[n](graph topology)    # parts of the topology
chan finalecho(graph topology)  # final topology

process Node[p = 0 to n-1] {
  bool links[n] = neighbors of node p;
  graph newtop, localtop = ([n*n] false);
  int parent;   # node from whom probe is received
  localtop[p,0:n-1] = links;   # initially my links

  receive probe[p](parent);
  # send probe to other neighbors, who are p's children
  for [q = 0 to n-1 st (links[q] and q != parent)]
    send probe[q](p);

  # receive echoes and union them into localtop
  for [q = 0 to n-1 st (links[q] and q != parent)] {
    receive echo[p](newtop);
    localtop = localtop or newtop;  # logical or
  }
  if (p == S)
    send finalecho(localtop);    # node S is root
  else
    send echo[parent](localtop);
}

process Initiator {
  graph topology;
  send probe[S](S)   # start probe at local node
  receive finalecho(topology);
}
```

Figure 9.11 Probe/echo algorithm for gathering the topology of a tree.

```
type graph = bool [n,n];
type kind = (PROBE, ECHO);
chan probe_echo[n](kind k; int sender; graph topology);
chan finalecho(graph topology);

process Node[p = 0 to n-1] {
  bool links[n] = neighbors of node p;
  graph newtop, localtop = ([n*n] false);
  int first, sender; kind k;
  int need_echo = number of neighbors - 1;
  localtop[p,0:n-1] = links;   # initially my links

  receive probe_echo[p](k, first, newtop);  # get probe
  # send probe on to to all other neighbors
  for [q = 0 to n-1 st (links[q] and q != first)]
    send probe_echo[q](PROBE, p, ∅);

  while (need_echo > 0) {
    # receive echoes or redundant probes from neighbors
    receive probe_echo[p](k, sender, newtop);
    if (k == PROBE)
      send probe_echo[sender](ECHO, p, ∅);
    else  # k == ECHO {
      localtop = localtop or newtop;  # logical or
      need_echo = need_echo-1;
    }
  }
  if (p == S)
    send finalecho(localtop);
  else
    send probe_echo[first](ECHO, p, localtop);
}

process Initiator {
  graph topology;   # network topology
  send probe_echo[source](PROBE, source, ∅);
  receive finalecho(topology);
}
```

Figure 9.12 Probe/echo algorithm for computing the topology of a graph.

```
type kind = enum(reqP, reqV, VOP, POP, ACK);
chan semop[n](int sender; kind k; int timestamp);
chan go[n](int timestamp);

process User[i = 0 to n-1] {
  int lc = 0, ts;
  ...
  # ask my helper to do V(s)
  send semop[i](i, reqV, lc); lc = lc+1;
  ...
  # ask my helper to do P(s), then wait for permission
  send semop[i](i, reqP, lc); lc = lc+1;
  receive go[i](ts); lc = max(lc, ts+1); lc = lc+1;
}

process Helper[i = 0 to n-1] {
  queue mq = new queue(int, kind, int);   # message queue
  int lc = 0, s = 0;         # logical clock and semaphore
  int sender, ts; kind k;    # values in received messages
  while (true) {   # loop invariant DSEM
    receive semop[i](sender, k, ts);
    lc = max(lc, ts+1); lc = lc+1;
    if (k == reqP)
      { broadcast semop(i, POP, lc); lc = lc+1; }
    else if (k == reqV)
      { broadcast semop(i, VOP, lc); lc = lc+1; }
    else if (k == POP or k == VOP) {
      insert (sender, k, ts) at appropriate place in mq;
      broadcast semop(i, ACK, lc); lc = lc+1;
    }
    else {  # k == ACK
      record that another ACK has been seen;
      for (all fully acknowledged VOP messages in mq)
        { remove the message from mq; s = s+1; }
      for (all fully acknowledged POP messages in mq st s > 0) {
        remove the message from mq; s = s-1;
        if (sender == i)    # my user's P request
          { send go[i](lc); lc = lc+1; }
      }
    }
  }
}
```

Figure 9.13 Distributed semaphores using a broadcast algorithm.

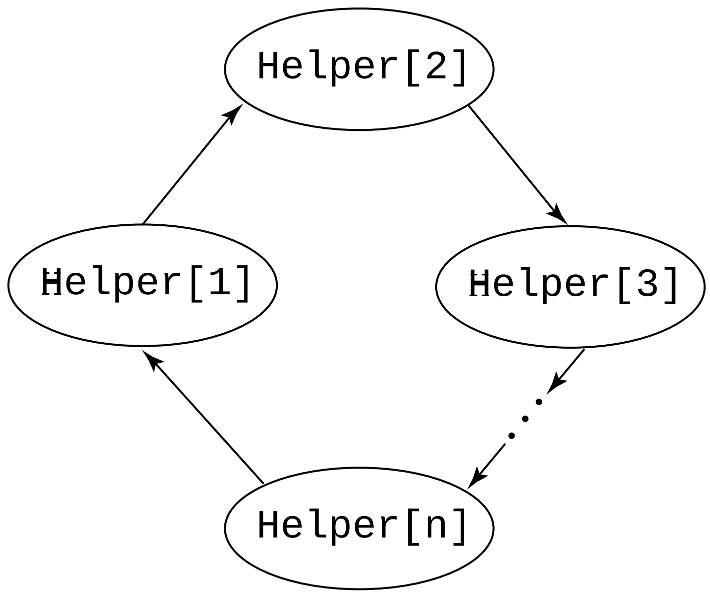


Figure 9.14 A token ring of helper processes.

```
chan token[1:n](), enter[1:n](), go[1:n](), exit[1:n]();

process Helper[i = 1 to n] {
  while (true) {   # loop invariant DMUTEX
    receive token[i]();          # wait for token
    if (not empty(enter[i])) {  # does user want in?
      receive enter[i]();          # accept enter msg
      send go[i]();                # give permission
      receive exit[i]();           # wait for exit
    }
    send token[i%n + 1]();       # pass token on
  }
}

process User[i = 1 to n] {
  while (true) {
    send enter[i]();           # entry protocol
    receive go[i]();
    critical section;
    send exit[i]();            # exit protocol
    non-critical section;
  }
}
```

Figure 9.15 Mutual exclusion with a token ring.

Global invariant *RING*:

    T[1] is blue ⇒ (T[1] ... T[token+1] are blue ∧
                     ch[2] ... ch[token%n + 1] are empty)

actions of **T[1]** when it first becomes idle:

```
    color[1] = blue; token = 0; send ch[2](token);
```

actions of **T[2], ..., T[n]** upon receiving a regular message:

```
    color[i] = red;
```

actions of **T[2], ..., T[n]** upon receiving the token:

```
    color[i] = blue; token++; send ch[i%n + 1](token);
```

actions of **T[1]** upon receiving the token:

```
    if (color[1] == blue)
      announce termination and halt;
    color[1] = blue; token = 0; send ch[2](token);
```

Figure 9.16 Termination detection in a ring.

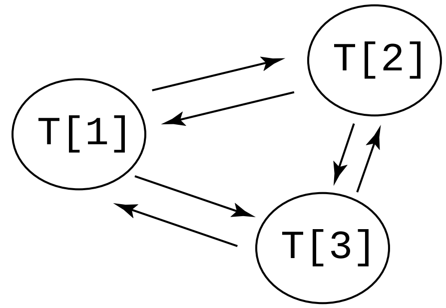


Figure 9.17 A complete communication graph.

Global invariant *GRAPH*:

token has value V $\Rightarrow$
  (the last V channels in cycle **C** were empty $\wedge$
  the last V processes to receive the token were **blue**)

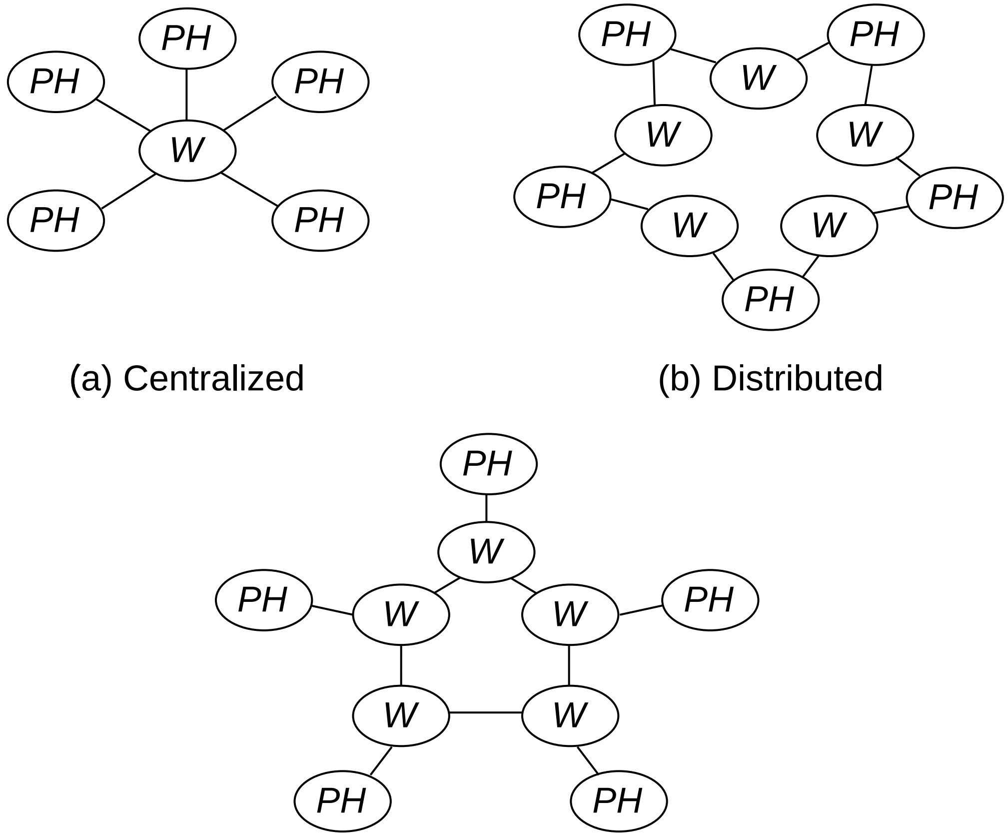
actions of **T[i]** upon receiving a regular message:

```
color[i] = red;
```

actions of **T[i]** upon receiving the token:

```
if (token == nc)
    announce termination and halt;
if (color[i] == red)
    { color[i] = blue; token = 0; }
else
    token++;
set j to index of channel for next edge in cycle C;
send ch[j](token);
```

Figure 9.18 Termination detection in a complete graph.



(a) Centralized

(b) Distributed

(c) Decentralized

Figure 9.19 Solution structures for the dining philosophers.

```
module Waiter[5]
  op getforks(), relforks();
body
  process the_waiter {
    while (true) {
      receive getforks();
      receive relforks();
    }
  }
end Waiter

process Philosopher[i = 0 to 4] {
  int first = i, second = i+1;
  if (i == 4) {
    first = 0; second = 4; }
  while (true) {
    call Waiter[first].getforks();
    call Waiter[second].getforks();
    eat;
    send Waiter[first].relforks();
    send Waiter[second].relforks();
    think;
  }
}
```

Figure 9.20 Distributed dining philosophers.

```
module Waiter[t = 0 to 4]
  op getforks(int), relforks(int);  # for philosophers
  op needL(), needR(),              # for waiters
     passL(), passR();
  op forks(bool,bool,bool,bool);  # for initialization
body
  op hungry(), eat();      # local operations
  bool haveL, dirtyL, haveR, dirtyR;  # status of forks
  int left = (t-1) % 5 ;       # left neighbor
  int right = (t+1) % 5;       # right neighbor

  proc getforks() {
    send hungry();  # tell waiter philosopher is hungry
    receive eat();  # wait for permission to eat
  }

  process the_waiter {
    receive forks(haveL, dirtyL, haveR, dirtyR);
    while (true) {
      in hungry() ->
          # ask for forks I don’t have
          if (!haveR) send Waiter[right].needL();
          if (!haveL) send Waiter[left].needR();
          # wait until I have both forks
          while (!haveL or !haveR)
            in passR() ->
                haveR = true; dirtyR = false;
            [] passL() ->
                haveL = true; dirtyL = false;
            [] needR() st dirtyR ->
                haveR = false; dirtyR = false;
                send Waiter[right].passL();
                send Waiter[right].needL()
            [] needL() st dirtyL ->
                haveL = false; dirtyL = false;
                send Waiter[left].passR();
                send Waiter[left].needR();
            ni
          # let philosopher eat, then wait for release
          send eat(); dirtyL = true; dirtyR = true;
          receive relforks();
      [] needR() ->
          # neighbor needs my right fork (its left)
          haveR = false; dirtyR = false;
          send Waiter[right].passL();
      [] needL() ->
          # neighbor needs my left fork (its right)
          haveL = false; dirtyL = false;
          send Waiter[left].passR();
      ni
    }
  }
end Waiter
```

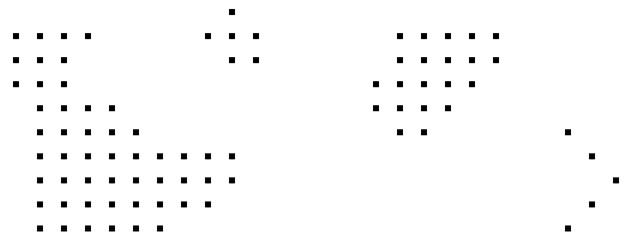
```
process Philosopher[i = 0 to 4] {
  while (true) {
    call Waiter[i].getforks();
    eat;
    call Waiter[i].relforks();
    think;
  }
}

process Main {  # initialize the forks held by waiters
  send Waiter[0].forks(true, true, true, true);
  send Waiter[1].forks(false, false, true, true);
  send Waiter[2].forks(false, false, true, true);
  send Waiter[3].forks(false, false, true, true);
  send Waiter[4].forks(false, false, false, false);
}
```

Figure 9.21 Decentralized dining philosophers.

```
process Worker[i = 1 to numWorkers] {
  declarations of local variables;
  initialize local variables;
  while (not done) {
    send values to neighbors;
    receive values from neighbors;
    update local values;
  }
}
```

Structure of heartbeat algorithms.



Sample image for the region-labeling problem.

```
sum = 0.0; na = 1; nb = 1;
c = elementsA[na]->index;     # column in row of A
r = elementsB[i][nb]->index;  # row in column of B
while (na <= lengthA and nb <= lengthB) {
  if (r == c) {
    sum += elementsA[na]->value *
             elementsB[i][nb]->value;
    na++; nb++;
    c = elementsA[na]->index;
    r = elementsB[i][nb]->index;
  } else if (r < c) {
    nb++; r = elementsB[i][nb]->index;
  } else { # r > c
    na++; c = elementsA[na]->index;
  }
}
if (sum != 0.0) {  # extend row of C
  elementsC[lengthC] = pair(i, sum);
  lengthC++;
}
```

Inner product code for Worker **`i`** in sparse matrix multiplication.

$a_{1,2}, b_{2,1}$	$a_{1,3}, b_{3,2}$	$a_{1,4}, b_{4,3}$	$a_{1,1}, b_{1,4}$
$a_{2,3}, b_{3,1}$	$a_{2,4}, b_{4,2}$	$a_{2,1}, b_{1,3}$	$a_{2,2}, b_{2,4}$
$a_{3,4}, b_{4,1}$	$a_{3,1}, b_{1,2}$	$a_{3,2}, b_{2,3}$	$a_{3,3}, b_{3,4}$
$a_{4,1}, b_{1,1}$	$a_{4,2}, b_{2,2}$	$a_{4,3}, b_{3,3}$	$a_{4,4}, b_{4,4}$

Initial arrangement for matrix multiplication by blocks.