Parallelism Execution

Marc Gonzàlez Tallada

Dept. d’Arquitectura de Computadors

Universitat Politècnica de Catalunya

Most significant issues

- Loop scheduling and work balance
- Locality
- Some optimizations
- Runtime overhead
- Nested parallelism

Loop scheduling

- **Loop parameters**
  - N: number of iterations
  - T: number of threads
  - Assume normalized space iteration
    - ✓ Iterations are identified in range [0, N-1]
    - ✓ STEP = 1
  - Threads numbered from 0 to T-1
    - ✓ ID: thread identifier

STATIC scheduling

- **Number of iterations per thread**
  - NITER = N / T
- **First iteration:**
  - FIRST = ID * NITER
- **Last iteration**
  - LAST = FIRST + NITER - 1
- **Rest of iterations**
  - Distribute one more iteration per thread

STATIC scheduling

- **Example**
  - N = 100
  - T = 2,3,4,5,6,7,8,10,11

```
!$OMP PARALLEL DO
  DO I = 1, 100
    A(I) = 0
  END DO
```

Number of threads	2	3	4	5	6	7	8	10	11
Iterations per thread	50	33	25	20	16	14	12	10	9
Rest of iterations	0	1	0	0	4	2	4	0	1
Biggest piece of work	50	34	25	20	17	15	13	10	10

INTERLEAVE scheduling

- **Blocked distribution of iterations**
  - CHUNK
- **Number of iterations per thread**
  - Depends on the CHUNK value
  - NCHUNK = N / CHUNK
  - NITER = (NCHUNK / T) * CHUNK
- **First iteration:**
  - FIRST = ID*CHUNK
- **Last iteration**
  - LAST = FIRST + CHUNK * T * NCHUNK - 1
- **Rest of iterations**
  - Only one piece of work with less iterations than the value of CHUNK
  - Assigned to the corresponding thread

INTERLEAVE scheduling

- **Example**
  - N = 100
  - CHUNK = 5
  - NCHUNK = 20
  - T = 2,3,4,5,6,7,8,10,11

```
!$OMP PARALLEL DO
  DO I = 1, 100
    A(I) = 0
  END DO
```

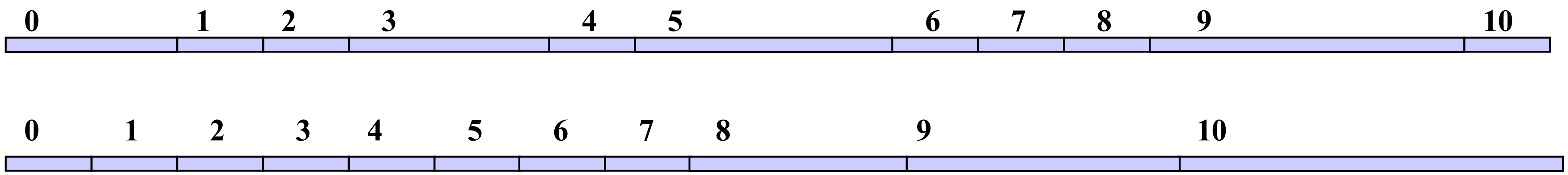
Number of threads	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**10**	**11**
Number of chunks per thread	**10**	**6**	**5**	**4**	**3**	**2**	**2**	**2**	**1**
Iterations per thread	**50**	**30**	**25**	**20**	**15**	**10**	**10**	**10**	**5**
Rest of chunks	**0**	**2**	**0**	**0**	**2**	**6**	**4**	**0**	**9**
Biggest piece of work	**50**	**35**	**25**	**20**	**20**	**15**	**15**	**10**	**10**

DYNAMIC scheduling

- **Blocked distribution of iterations**
  - CHUNK
- **Number of iterations per thread**
  - Depends on the CHUNK value
    - ✓ NCHUNK = N / CHUNK
  - Depends on runtime events
    - ✓ Threads dynamically pick up pieces of work of CHUNK iterations
- **First iteration:**
  - FIRST = *unknown*
- **Last iteration**
  - LAST = *unknown*
- **Rest of iterations**
  - Dynamically assigned

DYNAMIC scheduling

- **Requires thread synchronization for the work dispatch operations**
  - Mutual exclusion
  - The number of synchronizations depends on the CHUNK size
- **Used to balance the work between the executing threads**
- **The usefulness depends on the iteration space**
  - Iterations are not equally loaded
  - Most weighted iterations are uniformly distributed within the iteration space

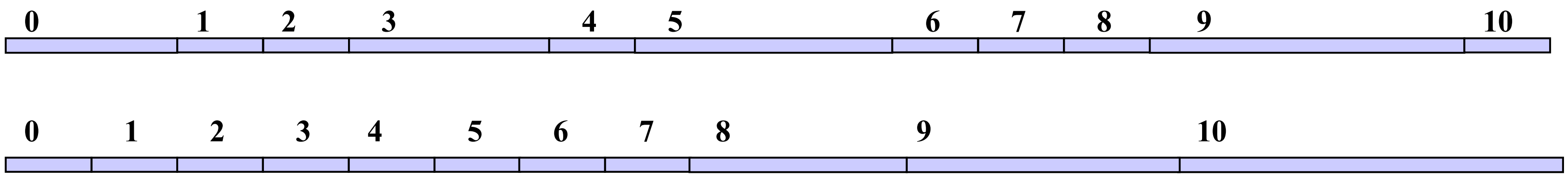


GUIDED scheduling

- **Incorporates some level of STATIC scheduling to guide the dynamic behavior**
- **Attempts to achieve two objectives**
  - Minimize the amount of synchronization overhead
  - Keep all threads busy at all times
- **Parallel iterations are scheduled**
  - Iterations are grouped and dispatched to threads
    - ✓ Reduces the thread synchronizations, likewise a DYNAMIC scheduling with constant value for CHUNK
  - The size of the CHUNK is no longer constant, determined to ensure all the threads will receive work to execute

GUIDED scheduling

- **Size of chunk of work**
  - N_t is the set of remaining iterations at a given time *t*
  - $CHUNK_t = N_t / T$
  - $N_{t+1} = N_t - CHUNK_t$
- **The usefulness depends on the iteration space**
  - Iterations are not equally loaded
  - Most weighted iterations are uniformly distributed within the iteration space



GUIDED scheduling

- **Example**
  - $N_0 = 100$
  - $T = 2$

```
!$OMP PARALLEL DO
   DO I = 1, 100
     A(I) = 0
   END DO
```

t	0	1	2	3	4	5	6	7	8
N_t	100	50	25	13	7	4	2	1	0
$CHUNK_t$	50	25	12	6	3	2	1	1	-
Thread 0	-	25	12	6	-	2	-	1	-
Thread 1	50	-	-	-	3	-	1	-	-

Locality (1)

- **When a loop is scheduled, data associated to iterations moves**
- **Data movement implies overhead**
  - Try to maintain the data distribution associated to the applied scheduling
- **Example**

```
  PROGRAM main
  ...
  DO timestep = 1, NSTEPS
    ...
    CALL compute()
    ...
  END DO
  ...
  END
```

```
  SUBROUTINE compute ()
  ...
!$OMP PARALLEL DO SCHEDULE(STATIC)
  DO I = 1, 1000
    A(I) = ...
  END DO
!OMP PARALLLE DO SCHEDULE(STATIC)
  DO I = 1, 1000
    B(I) = A(I) + ...
  END DO
  ...
  END
```

Locality (2)

- **Dependences might prevent keeping constant the data locality through work distribution**

```
PROGRAM main
...
DO timestep = 1, NSTEPS
  ...
  CALL compute()
  ...
END DO
...
END
```

```
  SUBROUTINE compute ()
  ...
!$OMP PARALLEL DO SCHEDULE(STATIC)
  DO J = 1, 1000
    DO I = 1, 1000
      A(I,J) = ...
    END DO
  END DO
!OMP PARALLLE DO SCHEDULE(STATIC)
  DO I = 1, 1000
    DO J = 1, 1000
      B(I,J) = B(I,J-1) + A(I,J) + ...
    END DO
  END DO
  ...
  END
```

Locality (3)

- **Depending on the costs associated to data movement, might be worth to execute sequentially, yet distributing work**

```
PROGRAM main
...
DO timestep = 1, NSTEPS
  ...
  CALL compute()
  ...
END DO
...
END
```

```
  SUBROUTINE compute ()
  ...
!$OMP PARALLEL DO SCHEDULE(STATIC)
  DO J = 1, 1000
    DO I = 1, 1000
      A(I,J) = ...
    END DO
  END DO
!OMP PARALLLE DO SCHEDULE(STATIC) ORDERED
  DO J = 1, 1000
!$OMP ORDERED
    DO I = 1, 1000
      B(I,J) = B(I,J-1) + A(I,J) + ...
    END DO
!$OMP END ORDERED
  END DO
  ...
  END
```

Some optimizations (1)

- **Loop fusion**
  - Loop fusion is the reverse of loop distribution
  - It reduces the loop fork/join overhead

```
!$OMP PARALLEL DO
  DO I = 1, N, 1
    A(I) = B(I)
  END DO
!$OMP PARALLEL DO
  DO I = 1, N, 1
    C(I) = A(I) + D(I)
  END DO
```

```
!$OMP PARALLEL DO
  DO I = 1, N, 1
    A(I) = B(I)
    C(I) = A(I) + D(I)
  END DO
```

Some optimizations (2)

- **Loop coalescing**
  - Can increase the number of iterations of a parallel loop
    - ✓ load balancing
  - Adds additional computation
    - ✓ overhead

```
!$OMP PARALLEL DO
  DO i=1,n
    DO j=1,m
      A(i,j) = B(i,j)
    ENDDO
  ENDDO
```

```
!$OMP PARALLEL DO PRIVATE(i,j)
  DO ij=1,n*m
    i = 1 + (ij-1) DIV m
    j = 1 + (ij-1) MOD m
    A(i,j) = B(i,j)
  ENDDO
```

Some optimizations (3)

- **Loop interchange**
  - Granularity of parallel computation (compare the number of parallel loops started)
  - Locality of reference (compare the cache-line reuse) these two effects may impact the performance in the same or in opposite directions.

```
  DO i=1,n
!$OMP PARALLEL DO
    DO j=1,m
      A(i,j) = A(i-1,j)
    ENDDO
  ENDDO
```

```
!$OMP PARALLEL DO
  DO j=1,m
    DO i=1,n
      A(i,j) = A(i-1,j)
    ENDDO
  ENDDO
```

Some optimizations (4)

- **Loop blocking**
  - Applied in the presence of dependences
    - ✓ Parallel execution requires some synchronizations to ensure dependences
  - Adds additional computation
    - ✓ overhead
  - Block size controls the work granularity between two synchronizations

Some optimizations (5)

- **Loop blocking**
  - Example

```
DO j=1,m
  DO i=1,n
    B(i,j)=A(i,j)+A(i-1,j-1)
  ENDDO
ENDDO
```

```
!OMP PARALLEL DO PRIVATE(sj,ej,si,ei)
  DO bj = 1, nbj
    DO bi = 1, nbi
      sj = MIN(nbj,1 + sizej*(bj-1))
      ej = sj + sizej
      si = MIN(nbi,1 + sizei*(bi-1))
      ei = si + sizei
      DO j= sj, ej
        DO i= si, ei
          B(i,j)=A(i,j)+A(i,j-1)
        ENDDO
      ENDDO
    END DO
  END DO
```

```
!OMP PARALLEL DO PRIVATE(sj,ej,si,ei)
  DO bj = 1, nbj
    DO bi = 1, nbi
      sj = MIN(nbj,1 + sizej*(bj-1))
      ej = sj + sizej
      si = MIN(nbi,1 + sizei*(bi-1))
      ei = si + sizei
      CALL wait(bj,bi)
      DO j= sj, ej
        DO i= si, ei
          B(i,j)=A(i,j)+A(i,j-1)
        ENDDO
      ENDDO
      CALL synch(bj,bi)
    END DO
  END DO
```

Some optimizations (6)

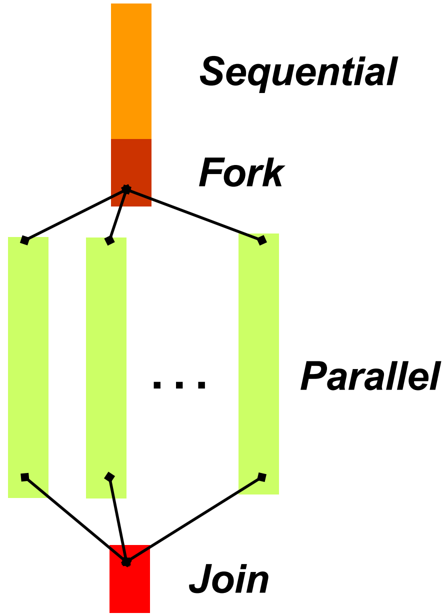
- **Relation between the memory model and the traversal of the iteration space**
  - Shared Memory Model
    - ✓ Cache line
  - Distributed Shared Memory Model
    - ✓ Page size
- **Parallelize loops where the applied work distribution does not imply to share data (cache line or memory page) among the executing threads**
  - Stressing the memory consistency protocol
    - ✓ Communication overhead

```
!$OMP PARALLEL DO
  DO I = 1, 1000
    DO J = 1, 1000
      A(I,J) = ...
    END DO
  END DO
```

```
!$OMP PARALLEL DO
  DO J = 1, 1000
    DO I = 1, 1000
      A(I,J) = ...
    END DO
  END DO
```

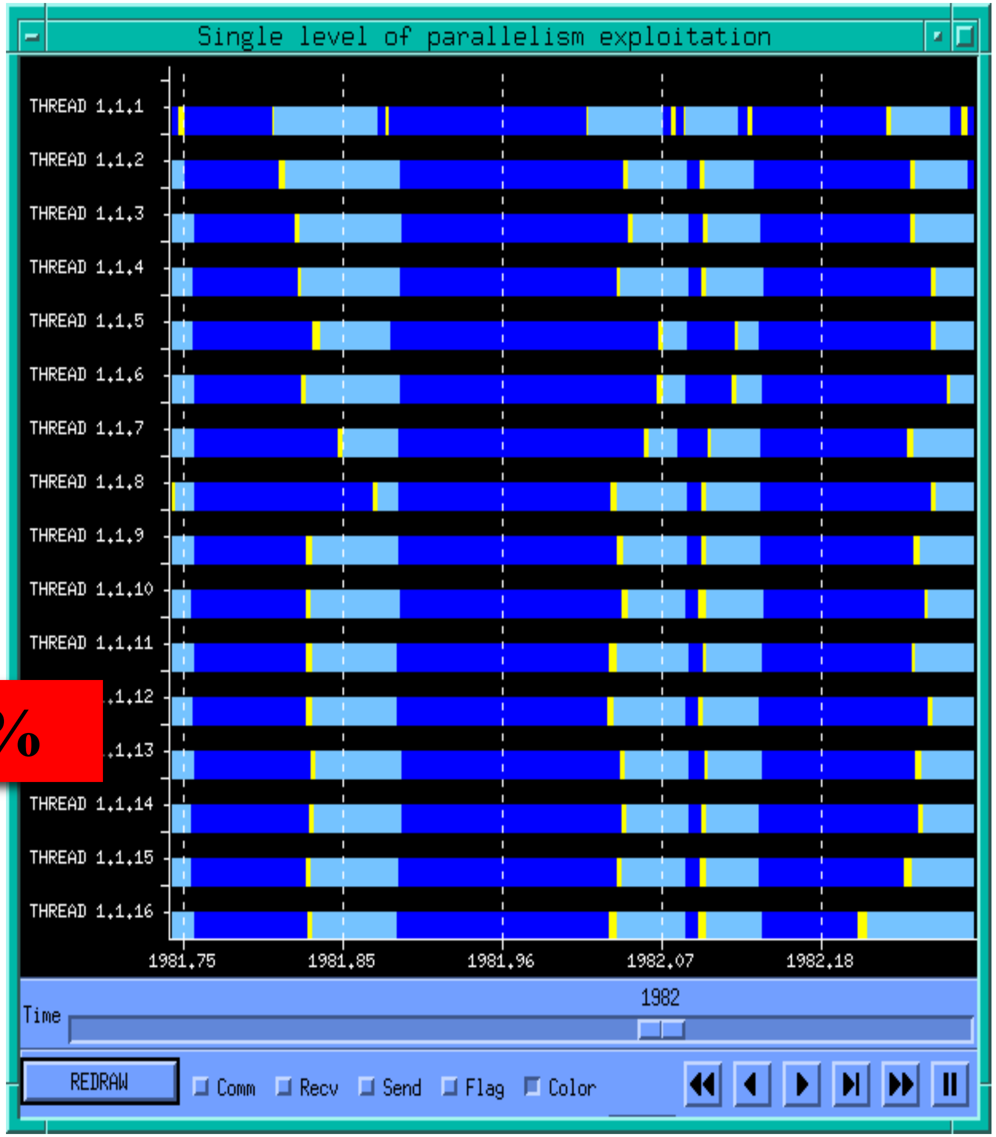
Runtime overhead

- **Assuming a fork/join model**
  - Overhead due to runtime execution
    - ✓ Fork
    - ✓ Join
  - The number of executing threads is critical
- **Within the parallel execution of a loop**
  - Relation between the parallel work and the runtime overheads
  - Load unbalance due to variances in execution times
    - ✓ Same number of iterations, different execution time
  - Increasing the number of threads, might incur in a loose of performance



Loosing performance (1)

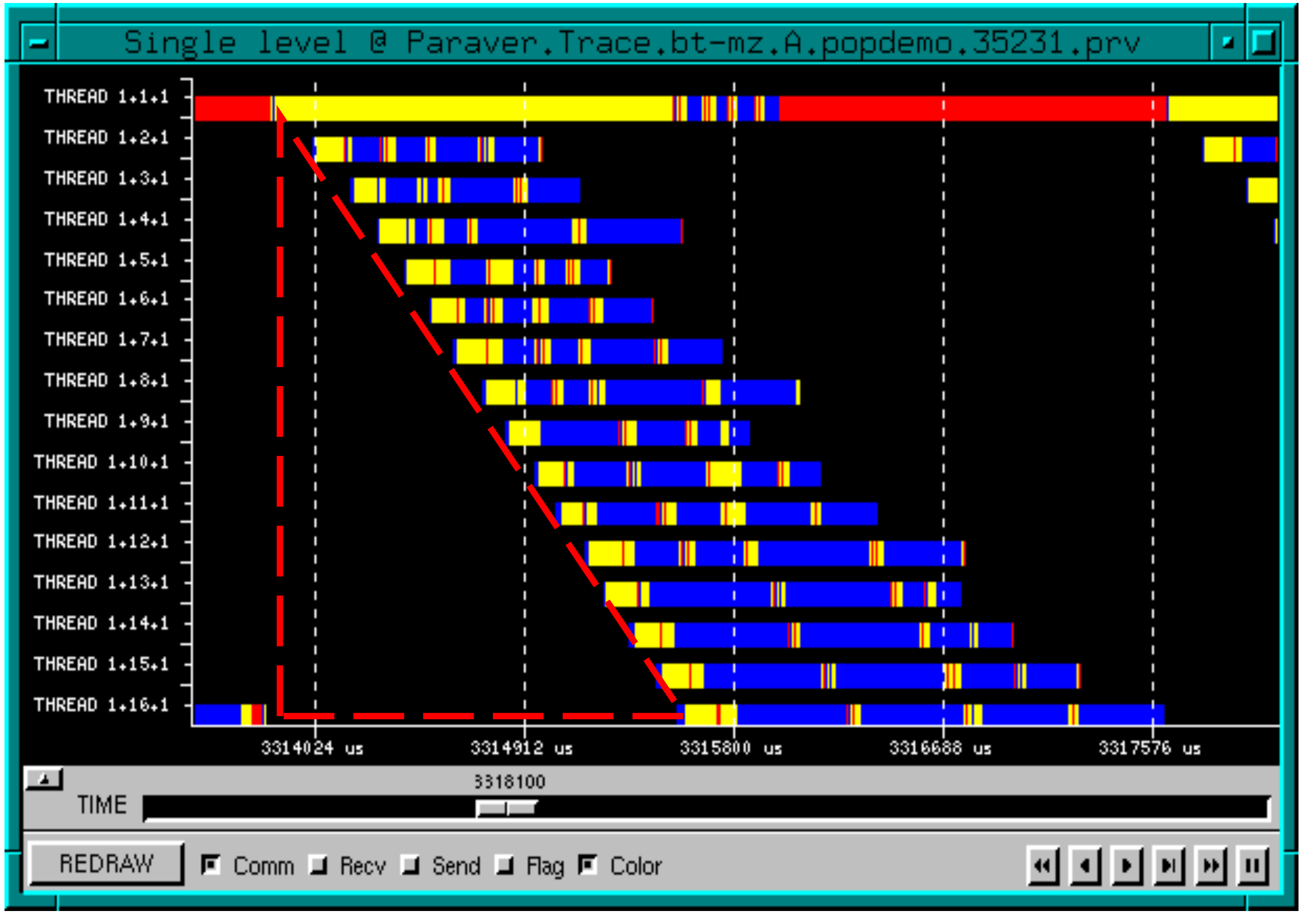
- **Single level of parallelism**
  - Load unbalancing
  - Fine grain
  - Fork/join cost



Utilization: 66%

Loosing performance (2)

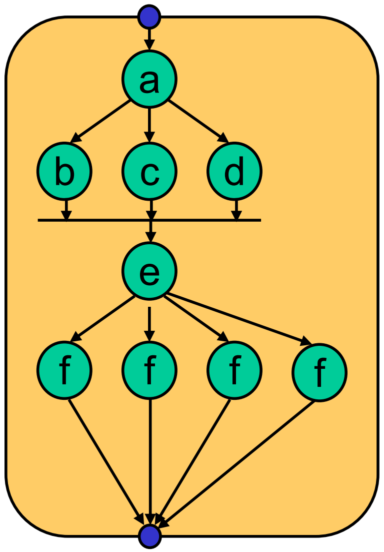
- **There is a point, where increasing the number of threads does not report any increase on performance**



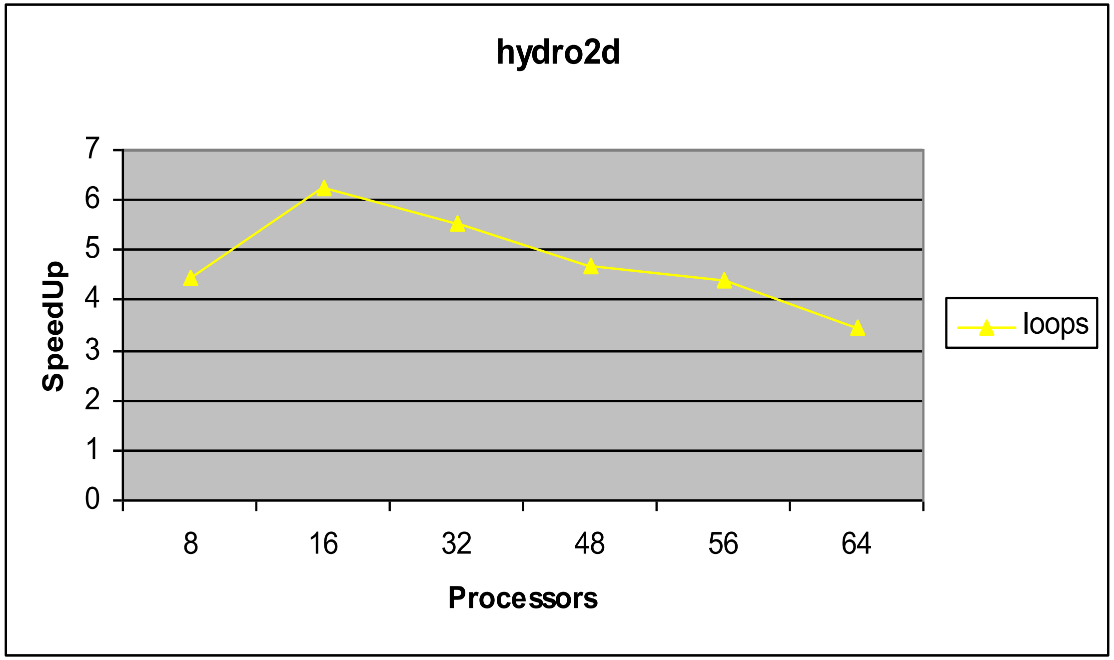
Runtime synchronization Runtime code Parallel work

Loosing performance (3)

- **Some experiments**
  - SPEC95 hydro2d



a, b, c, d, e, f function calls

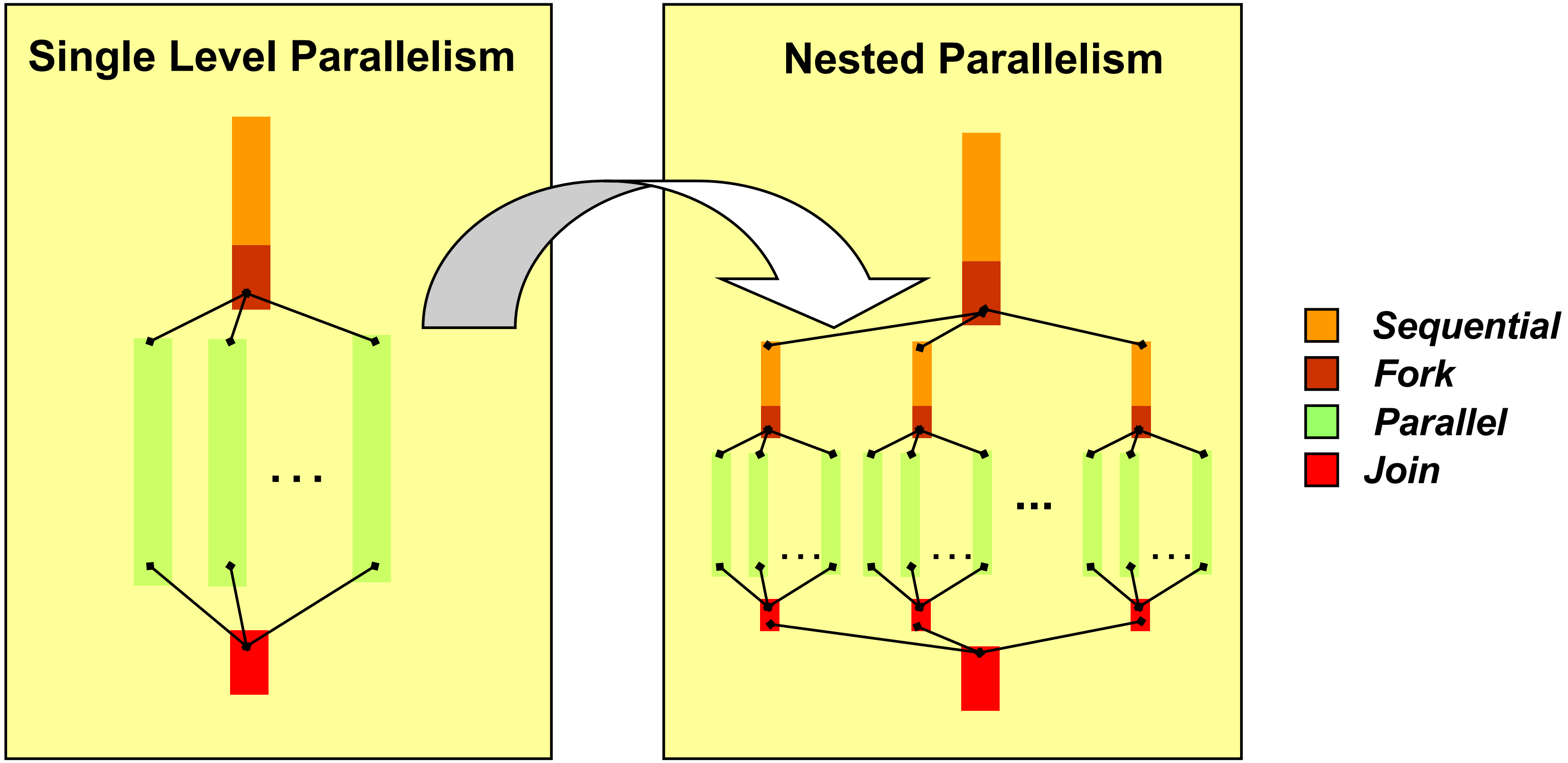


Performance degradation with more than 16 processors

Loosing performance (4)

- **Solution**
  - Increase the work granularity
    - ✓ Transform the loop
      - – Loop fusion, coalescing ...
  - Look for more parallelism
    - ✓ Nested parallelism

Nested parallelism



Each thread is able of generating more parallelism

Take OpenMP execution model

- **Parallelism definition**
  - PARALLEL, END PARALLEL

```
   program
   ...
!$OMP PARALLEL
   User Code
!$OMP END PARALLEL
   ...
   end
```

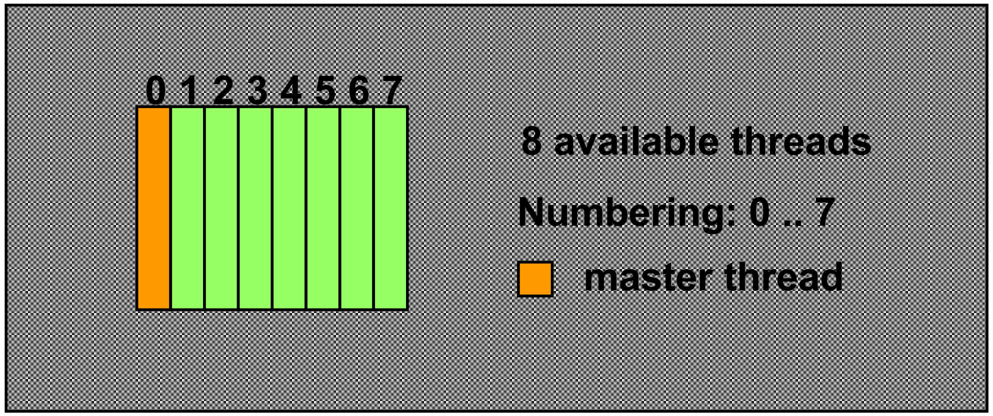
-A team with all available threads is defined

-All threads execute the enclosed code

-Threads are numbered from 0 to #nthreads-1

-Thread with id 0 is the master of the team

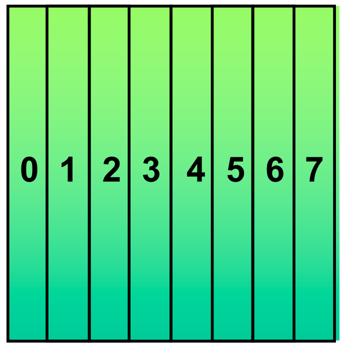
-Thread numbering is used for the work distribution



Threads and levels of parallelism

- **What threads to be used in the inner levels ?**
  - How many ?

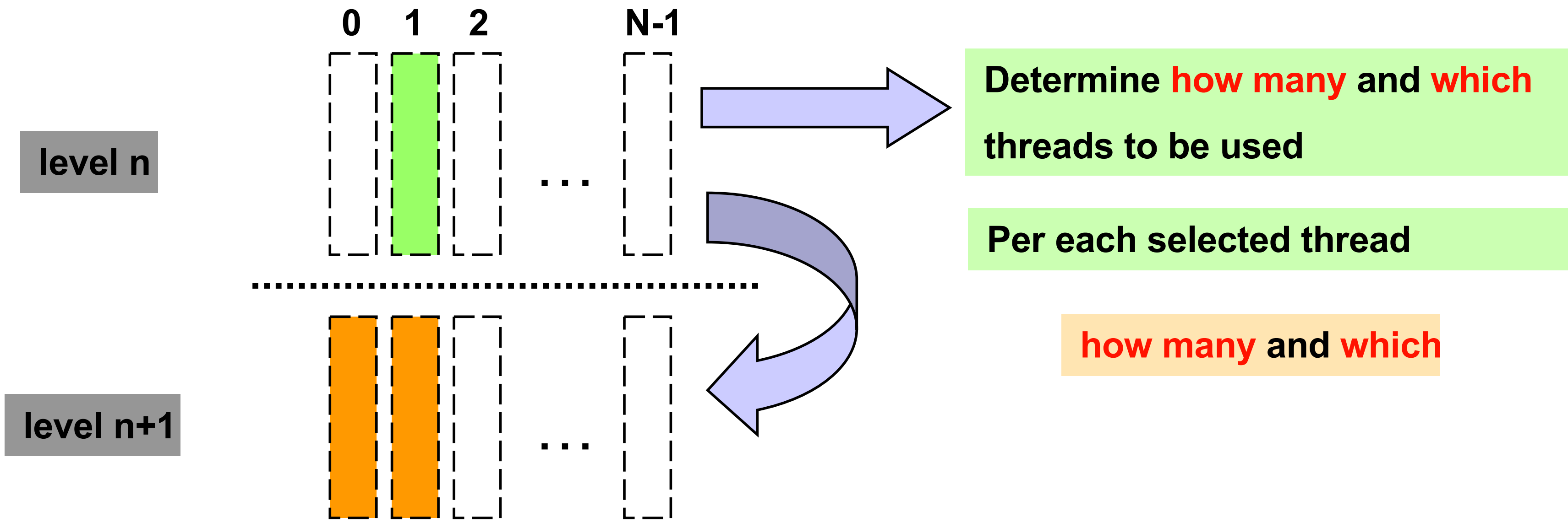
```
C$OMP PARALLEL SECTIONS
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1, N
         CALL SAME ( … )
      ENDDO
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,N
         CALL SAME ( … )
      ENDDO
C$OMP END PARALLEL SECTIONS
```



Threads and levels of parallelism

- **What is missing ?**

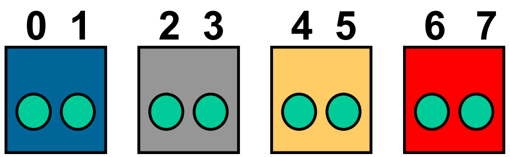
Allow the programmer to define the relation between two immediate nested levels according to the active numeration



Available threads: 0 ... N-1

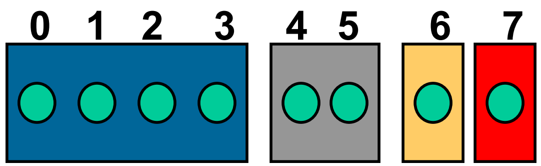
Thread groups

■ The group concept



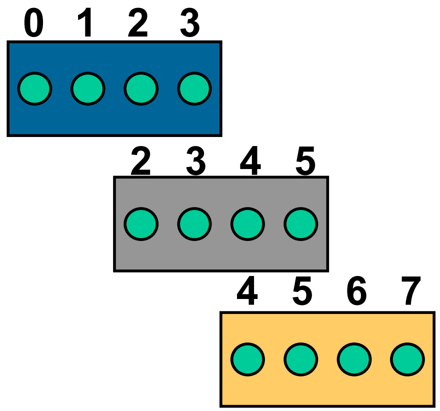
Uniform and Disjoint Groups

ngroups = 4
masters[4] = {0, 2, 4, 6}
howmany[4] = {2, 2, 2, 2}



Non Uniform and Disjoint Groups

ngroups = 4
masters[4] = {0, 4, 6, 7}
howmany[4] = {4, 2, 1, 1}



Uniform and Non Disjoint Groups

ngroups = 3
masters[3] = {0, 2, 4}
howmany[3] = {4, 4, 4}

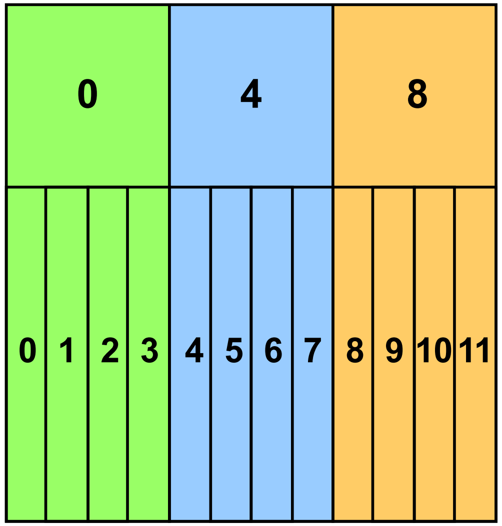
■ Definition

- N consecutive threads inside the current team
- A thread might belong to more than one group
- A group definition may only occur when a parallel region is started
  - ✓ the thread encountering the parallel region becomes its master
  - ✓ creates a team with as many threads as groups in the definition

Proposal of Thread Groups

Uniform and Disjoint

```
C$OMP PARALLEL SECTIONS GROUPS(3)
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1, N
         CALL SAME ( … )
      ENDDO
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,N
         CALL SAME ( … )
      ENDDO
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,N
         CALL SAME ( … )
      ENDDO
C$OMP END PARALLEL SECTIONS
```



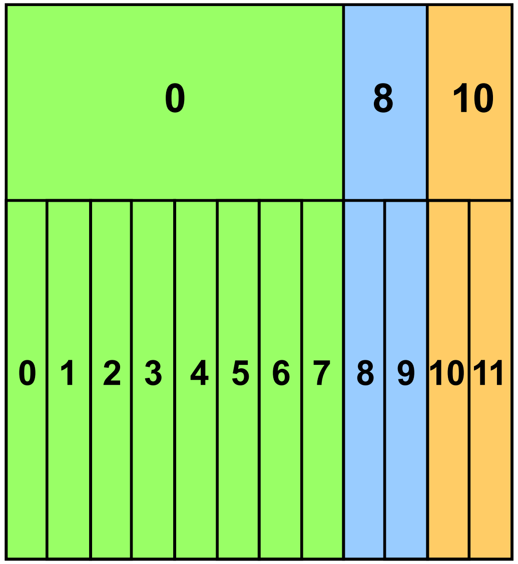
ngroups = 3
masters[3] = {0, 4, 8}
howmany[3] = {4, 4, 4}

Proposal of Thread Groups

- **Non Uniform and Disjoint (dynamic)**

```
C$OMP PARALLEL SECTIONS GROUPS(3, weight)
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1, N
         CALL EXPENSIVE ( … )
      ENDDO
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,N
         CALL CHEAP ( … )
      ENDDO
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,N
         CALL CHEAP ( … )
      ENDDO
C$OMP END PARALLEL SECTIONS
```

weight[3] = {4, 1, 1}



ngroups = 3
masters[3] = {0, 8, 10}
howmany[3] = {8, 2, 2}

Thread distribution

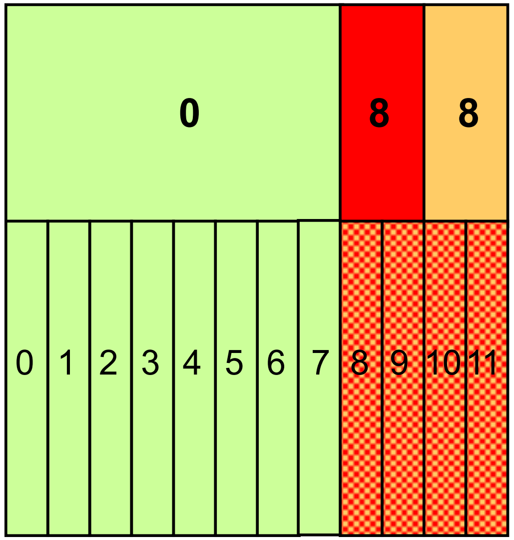
- **Input**
  - weight [ ]
- **Output**
  - master [ ]
  - howmany [ ]

```
howmany(1:ngroups) = 1
do while (sum(howmany(1:ngroups)) .lt. nthreads)
  pos = maxloc(weight(1:ngroups)/howmany(1:ngroups))
  howmany(pos) = howmany(pos) + 1
end do
masters(1) = 0
do i = 1, ngroups-1
  masters(i+1) = masters(i) + howmany(i)
end do
```

Proposal of Thread Groups

Non Uniform and non Disjoint (generic)

```
C$OMP PARALLEL SECTIONS GROUPS(3,masters,howmany)
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1, N
         CALL EXPENSIVE ( … )
      ENDDO
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,N
         CALL CHEAP ( … )
      ENDDO
C$OMP SECTION
C$OMP PARALLEL DO SCHEDULE(STATIC)
      DO  J=1,N
         CALL CHEAP ( … )
      ENDDO
C$OMP END PARALLEL SECTIONS
```



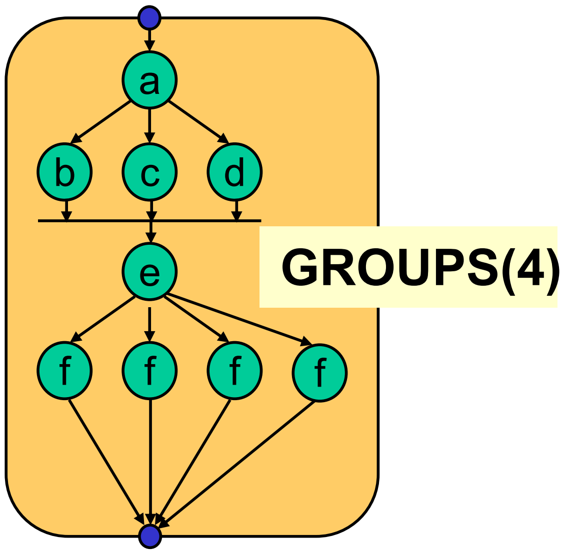
ngroups = 3

masters[3] = {0, 8, 8}

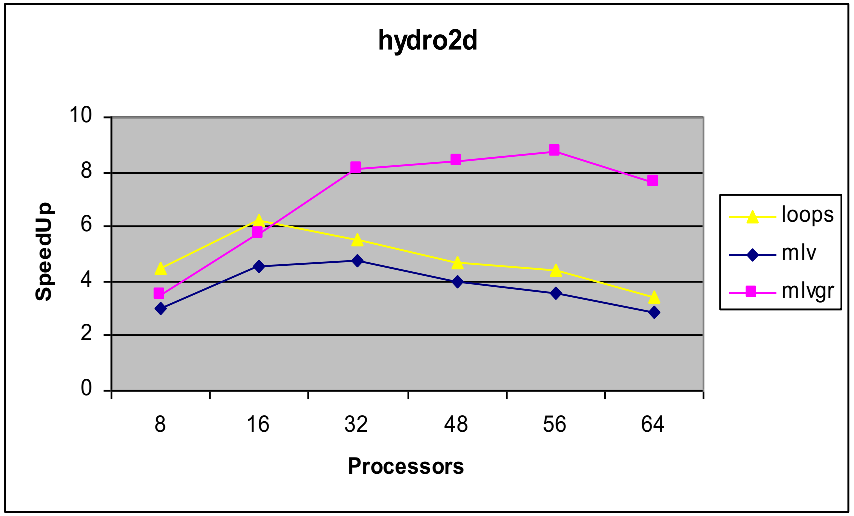
howmany[3] = {8, 4, 4}

Some experiments

- **SPEC95 hydro2d**



a, b, c, d, e, f function calls



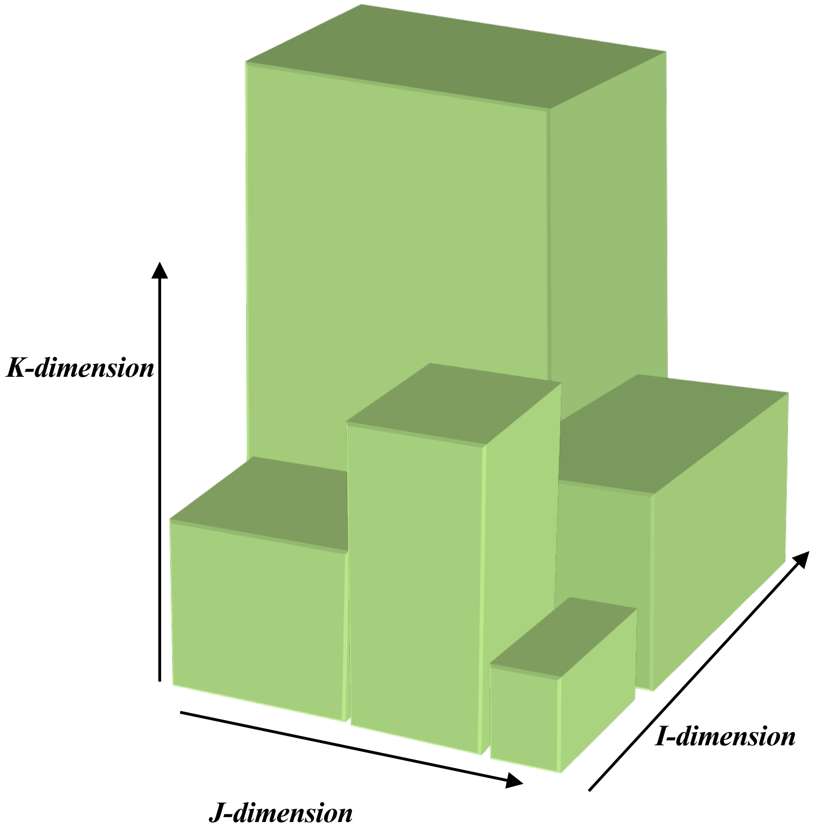
Loop parallelism: 6,26

Multilevel+Groups: 8,76

30%

Nested parallelism motivation: BT-MZ

- **NAS multi-zone benchmarks**
  - BT-MZ CLASS A
    - ✓ Different zone sizes



Block	I-dimension	J-dimension	K-dimension	Size	Proportions
1	13	13	16	2704	1
2	21	13	16	4368	1.61
3	36	13	16	7488	2.76
4	58	13	16	12064	4.46
5	13	21	16	4368	1.61
6	21	21	16	7056	2.61
7	36	21	16	12096	4.47
8	58	21	16	19488	7.20
9	13	36	16	7488	2.76
10	21	36	16	12096	4.47
11	36	36	16	20736	7.66
12	58	36	16	33408	12.35
13	13	58	16	12064	4.46
14	21	58	16	19488	7.20
15	36	58	16	33408	12.35
16	58	58	16	53824	19.9

Nested parallelism motivation: BT-MZ

- **Code structure**

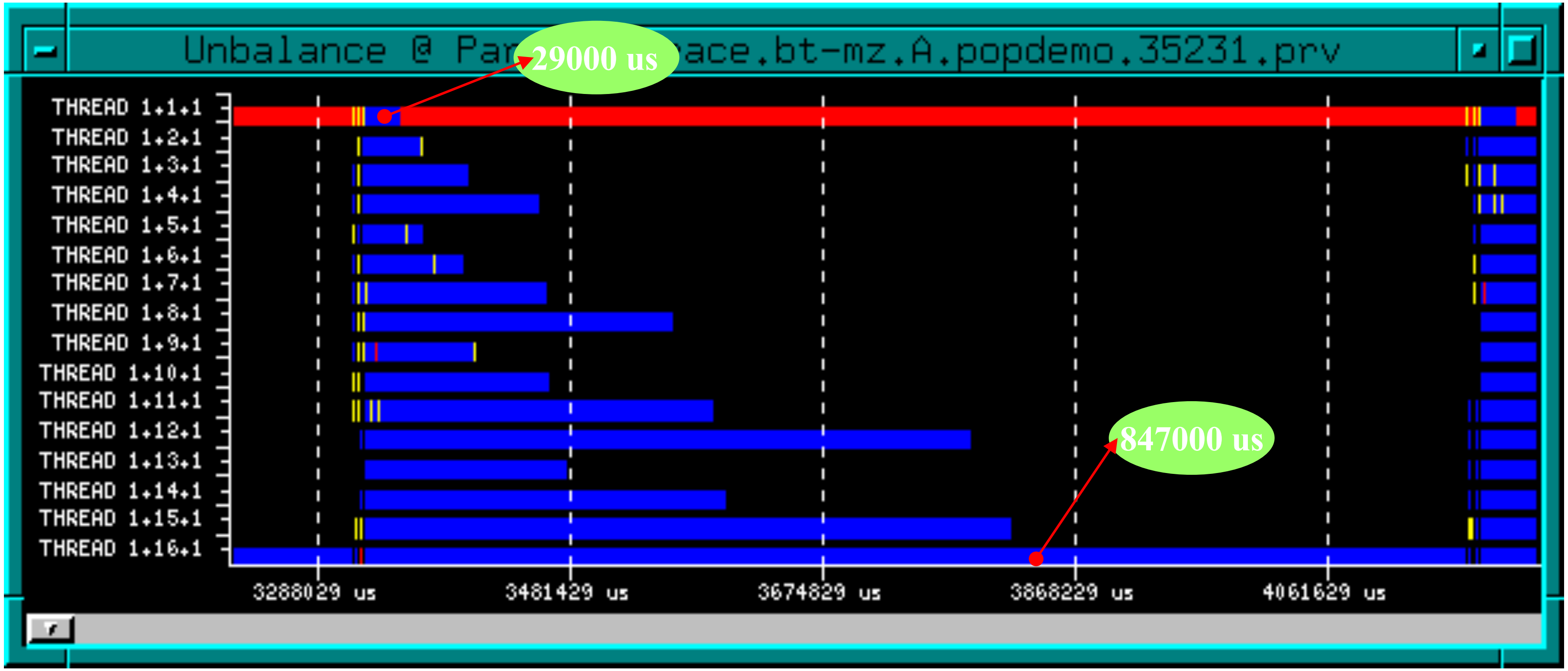
```
PROGRAM main
  ...
  DO step = 1, niter
    call exch_qbc(u, qbc, nx, nxmax, ny, nz, start5, qstart_west,
                  qstart_east, qstart_south, qstart_north)

C$OMP PARALLEL DO PRIVATE(zone)
    DO zone = 1, num_zones
      call adi(rho_i(start1(zone)), us(start1(zone)),
               vs(start1(zone)), ws(start1(zone)),
               qs(start1(zone)), square(start1(zone)),
               rhs(start5(zone)), forcing(start5(zone))
               u(start5(zone)),
               nx(zone), nxmax(zone), ny(zone), nz(zone
    END DO
C$OMP END PARALLEL DO
  END DO
  ...
END
```

```
SUBROUTINE adi (..., nx, ny, nz)
  ...
C$OMP PARALLEL DO PRIVATE (i,j,k)
  DO 200 j = 1, nx
  DO 200 k = 1, ny
  DO 200 I = 1, nz
    ...
  200 CONTINUE
  ...
  RETURN
END
```

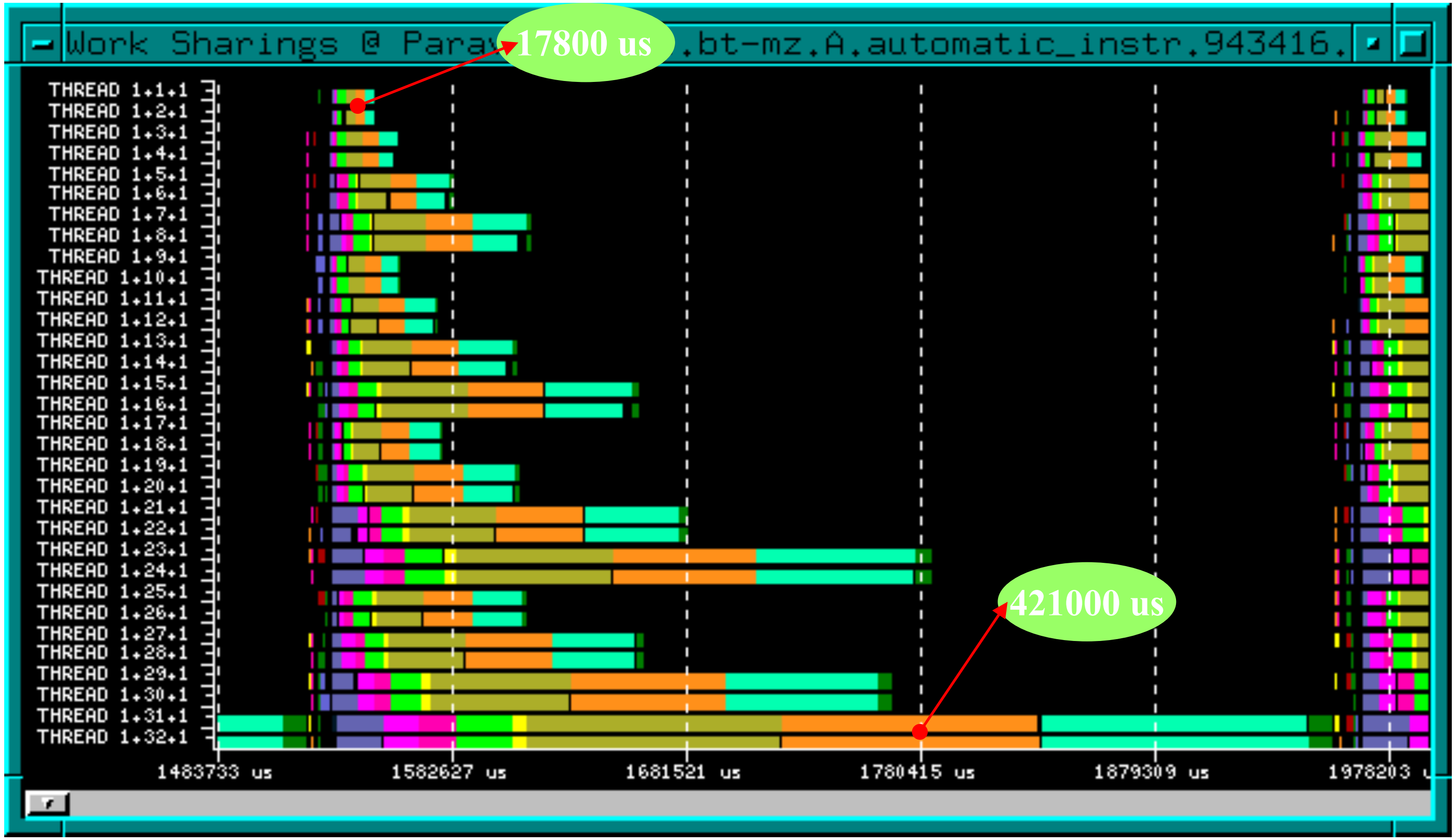
Nested parallelism motivation: BT-MZ

- **Unbalance in the outer level of parallelism**



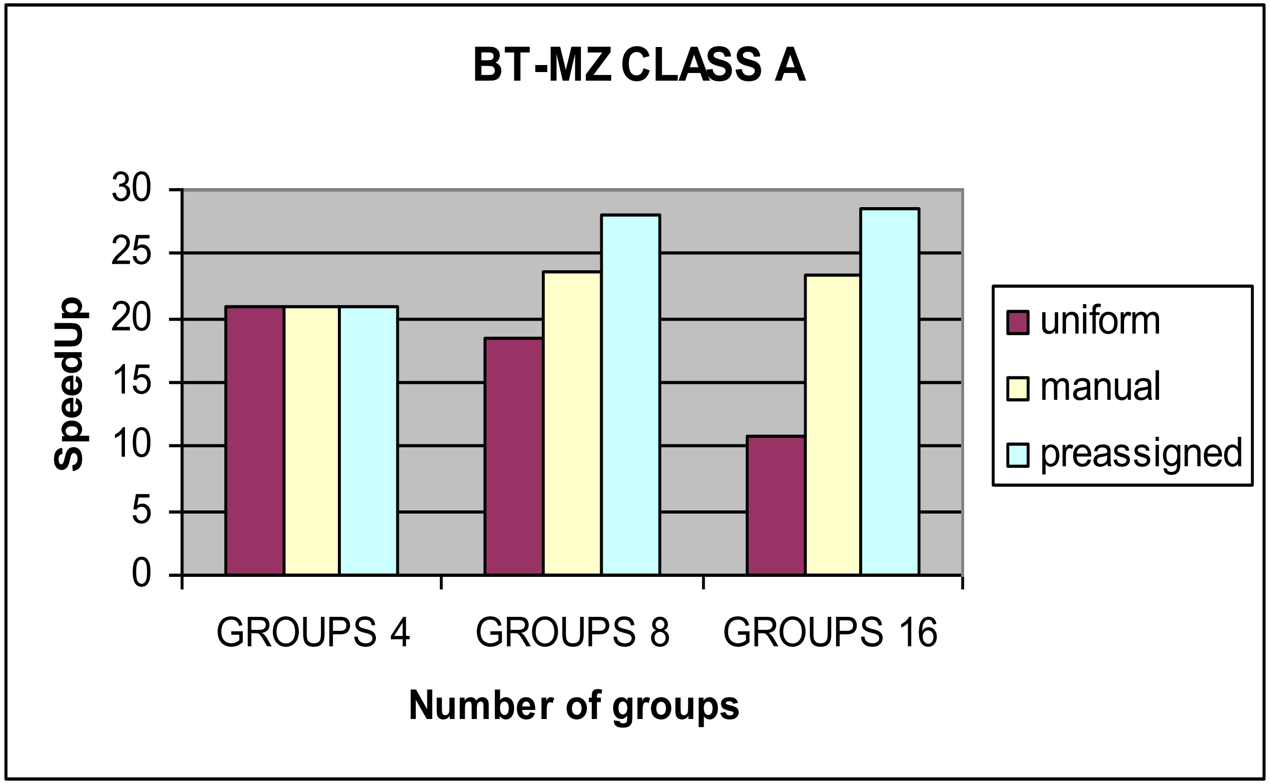
Nested parallelism motivation: BT-MZ

- **BT-MZ 16 zones, uniform distribution**
  - NP = 16 (outermost level)
  - NT = 2 (innermost level)



Some evaluation

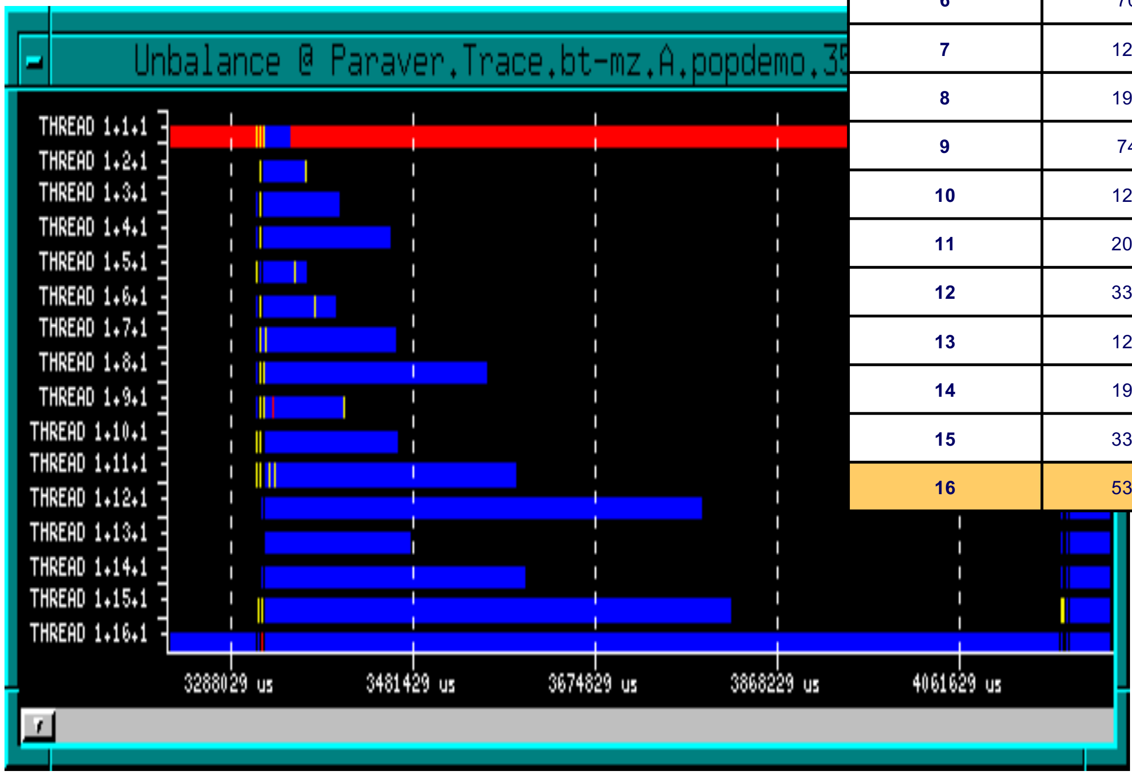
- **BT-MZ 16 zones, non uniform distribution**
  - 32 threads
  - 4, 8, 16 threads outermost level



Some evaluation

- **BT-MZ 16 zones, 16 threads**
  - Different unbalance ratios
    - ✓ Unbalance in zone size
    - ✓ Unbalance in execution time

Block	Size	Proportions Sizes	Execution Time	Proportions Execution Time
1	2704	1	29000	1
2	4368	1.61	46400	1,61
3	7488	2.76	80000	2,76
4	12064	4.46	135300	4,65
5	4368	1.61	47600	1,64
6	7056	2.61	78000	2,69
7	12096	4.47	141000	4,86
8	19488	7.20	240000	8,27
9	7488	2.76	87300	3,01
10	12096	4.47	144200	4,97
11	20736	7.66	270000	9,31
12	33408	12.35	467000	16,10
13	12064	4.46	157400	5,41
14	19488	7.20	280000	9,65
15	33408	12.35	500000	17,24
16	53824	19.9	847000	29,20



Proposal

- **Can we compute the best thread distribution ?**
  - Compile time
    - ✓ Not enough information
  - Run time
    - ✓ What metric ?
      - – Execution time
- **Parallelism is executed several times**
  - Allows for "observing"
  - Adapt the thread distribution at runtime
- **Instrument the parallel code**
  - Take samples of execution time
  - Derive load and possible unbalance
  - Distribute threads according to runtime measurements

Basic strategy

- **Starting thread distribution**
  - Uniform
- **Execute the parallelism and make runtime measurements**
  - Where to place the probes
    - ✓ Distortion caused by synchronization constructs
  - Accuracy
    - ✓ Distortion caused by the memory hierarchy
- **Redistribute the threads according to the measurements**
  - Thread distribution algorithm
  - Prediction of new execution times after new thread distribution
  - Validation of the thread distribution

Place the probes (1)

- **Immediate level of parallelism after a GROUPS definition**
  - Level n = “GROUPS definition”
  - Level n+1
    - ✓ Informs upper level about execution times

```
  ...
!$OMP PARALLEL GROUPS(...)
!$OMP DO
  DO I = 1, N
     CALL inner_parallelism ( )
  END DO
!$OMP END DO
!$OMP END PARALLEL
  ...
```

```
  SUBROUTINE inner_parallelism ( )
!$OMP PARALLEL
  ...
!$OMP END PARALLEL
  END
```

Place the probes (2)

- **At beginning/end of the parallel regions**
  - Synchronization constructs
    - ✓ Distortion in the samples
    - ✓ Contention can be understood as a demand of more threads

```
  ...
!$OMP PARALLEL GROUPS( N )
!$OMP DO
  DO I = 1, N
     CALL inner_parallelism (
  END DO
!$OMP END DO
!$OMP END PARALLEL
  ...
```

```
  SUBROUTINE inner_parallelism ( )
!$OMP PARALLEL
                                         PROBE
  ...
!$OMP CRITICAL
  DO j = 1, 5
    C ( j ) = …
  END DO
!$OMP END CRITICAL
                                         PROBE
!$OMP END PARALLEL
  END
```

Place the probes (3)

- **At beginning/end of work-sharing constructs**
  - Samples only include real computation
  - Mean of all work sharing constructs

```
  ...
!$OMP PARALLEL GROUPS( N )
!$OMP DO
  DO I = 1, N
     CALL inner_parallelism ( )
  END DO
!$OMP END DO
!$OMP END PARALLEL
  ...
```

```
  SUBROUTINE inner_parallelism ( )
!$OMP PARALLEL
  ...
                                   PROBE
!$OMP DO
  DO J = 1, YDIM
     ...
  END DO
!$OMP END DO
                                   PROBE
  ...
!$OMP END PARALLEL
  END
```

Thread distribution

- **Interpret samples as computational weights**

```
pos=minloc(samples(1:ngroups))
weight(1:ngroups)=samples(1:ngroups)/samples(pos)
howmany(1:ngroups) = 1
do while (sum(howmany(1:ngroups)) .lt. num_threads)
  pos =
maxloc(weight(1:ngroups)/howmany(1:ngroups))
  howmany(pos) = howmany(pos) + 1
end do
masters(1) = 0
do i = 1, ngroups-1
  masters(i+1) = masters(i) + howmany(i)
end do
```

Prediction of new critical path

- **Critical path validation algorithm**

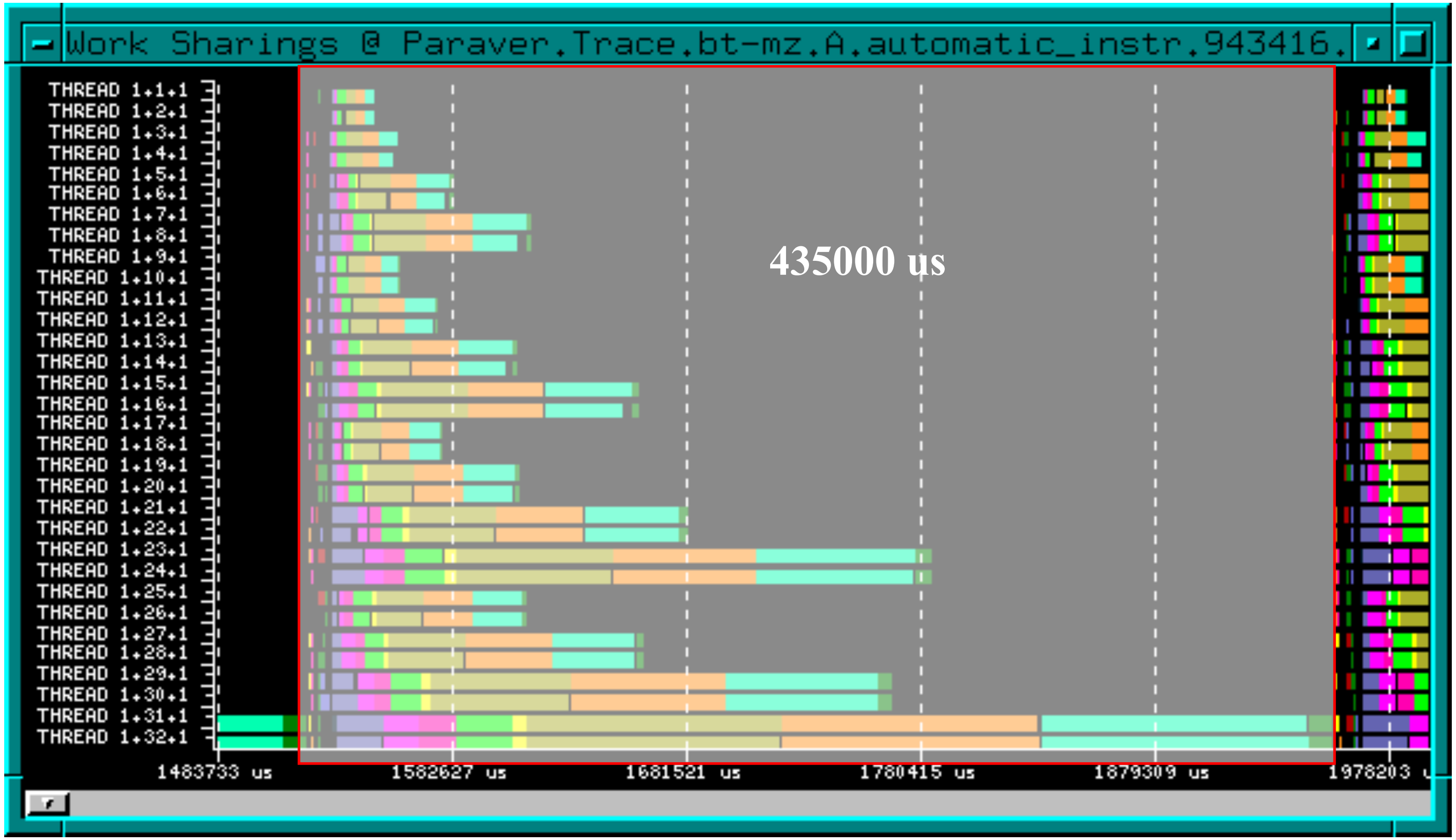
```
threshold = 0.1 ( number between 0 and 1 )
pos=maxloc(samples(1:ngroups)/new_howmany(1:ngroups))
new_critical_path=samples(pos)/new_howmany(pos)
if ( new_critical_path .lt prev_critical_path ) then
  difference = new_critical_path - prev_critical_path
  if ( difference .gt. (threshold * prev_critical_path) )
then
    return true
  else
    return false
  end if
else
  return false
endif
```

Distortion caused by cache effects

- **Warm the memory hierarchy**
  - Invoke the thread distribution after several executions
    - ✓ Every *n* executions
    - ✓ Always take samples of execution time
      - – Make the arithmetic mean of the last *n-1*
- **Main parameters**
  - Initial distribution: GROUPS (n)
  - Threshold = 0.1
  - Predictions
    - ✓ Divide the sampled execution time by the number of assigned threads
  - Sampling period: every *n = 3* executions

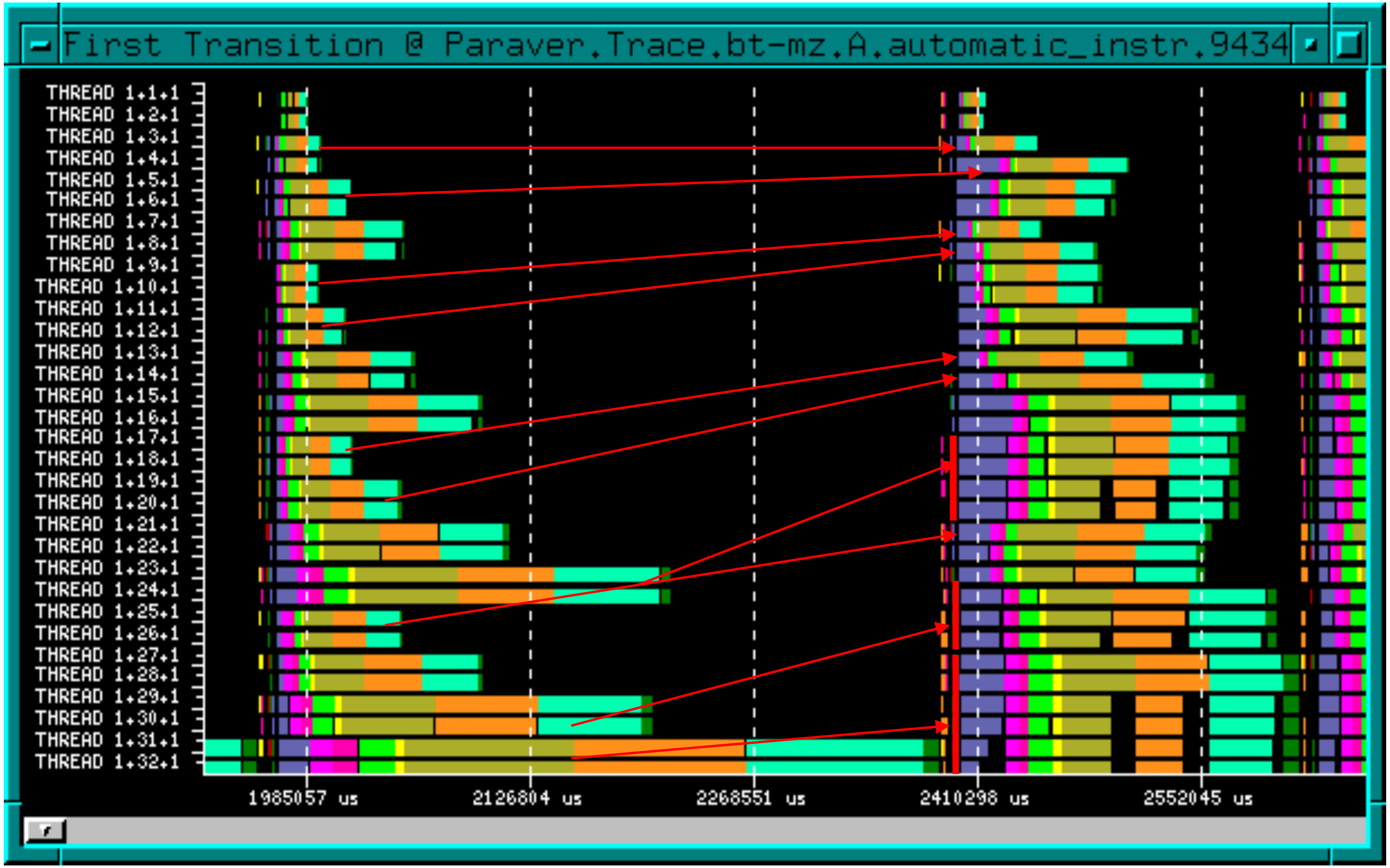
Initial distribution

- **BT-MZ 16 zones, uniform distribution**
  - 16 threads outermost level
  - 2 threads innermost level



First transition

- **Changes in zones**
  - 2, 3, 5, 6, 9, 10, 12, 13, 15, 16



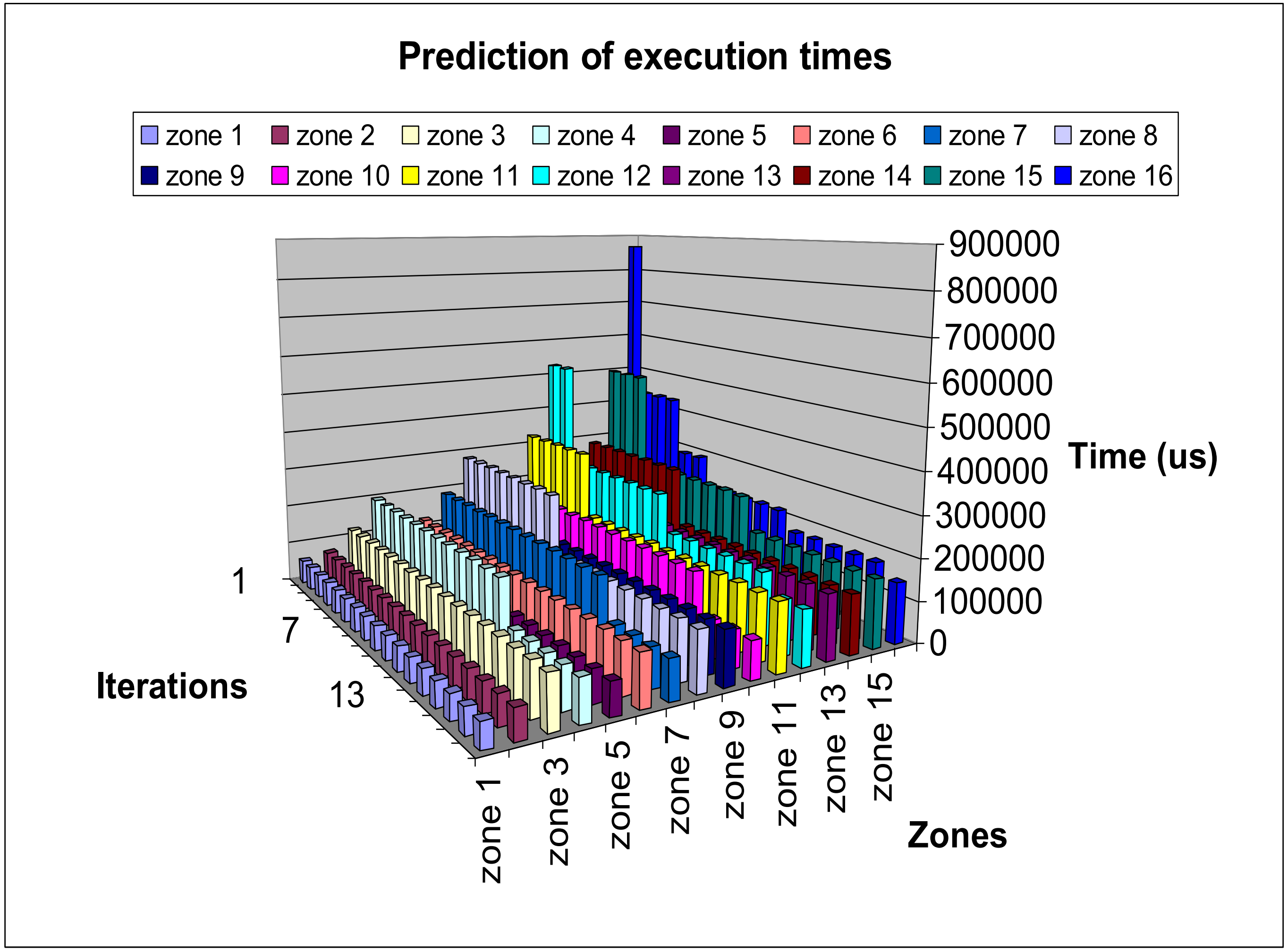
Block	Current	Next
1	2	2
2	2	1
3	2	1
4	2	2
5	2	1
6	2	1
7	2	2
8	2	2
9	2	1
10	2	1
11	2	2
12	2	4
13	2	1
14	2	2
15	2	3
16	2	6

Evolution of the distribution algorithm

Iteration		0		1		2		3		4		5		6		7	
zone 1	58000	1	58000	1	58000	1	58000	1	58000	1	58000	1	58000	1	58000	1	58000
zone 2	69000	1	69000	1	69000	1	69000	1	69000	1	69000	1	69000	1	69000	1	69000
zone 3	122200	1	122200	1	122200	1	122200	1	122200	1	122200	1	122200	1	122200	1	122200
zone 4	196000	1	196000	1	196000	1	196000	1	196000	1	196000	1	196000	1	196000	1	196000
zone 5	74400	1	74400	1	74400	1	74400	1	74400	1	74400	1	74400	1	74400	1	74400
zone 6	120000	1	120000	1	120000	1	120000	1	120000	1	120000	1	120000	1	120000	1	120000
zone 7	186000	1	186000	1	186000	1	186000	1	186000	1	186000	1	186000	1	186000	1	186000
zone 8	282000	1	282000	1	282000	1	282000	1	282000	1	282000	1	282000	1	282000	1	141000
zone 9	128000	1	128000	1	128000	1	128000	1	128000	1	128000	1	128000	1	128000	1	128000
zone 10	180000	1	180000	1	180000	1	180000	1	180000	1	180000	1	180000	1	180000	1	180000
zone 11	324000	1	324000	1	324000	1	324000	1	324000	1	162000	2	162000	2	162000	2	162000
zone 12	528000	1	528000	1	264000	2	264000	2	264000	2	264000	2	264000	2	264000	2	264000
zone 13	156000	1	156000	1	156000	1	156000	1	156000	1	156000	1	156000	1	156000	1	156000
zone 14	284000	1	284000	1	284000	1	284000	1	284000	1	284000	1	284000	1	142000	2	142000
zone 15	496000	1	496000	1	496000	1	248000	2	248000	2	248000	2	248000	2	248000	2	248000
zone 16	868000	1	434000	2	434000	2	434000	2	289333,3	3	289333,3	3	217000	4	217000	4	217000

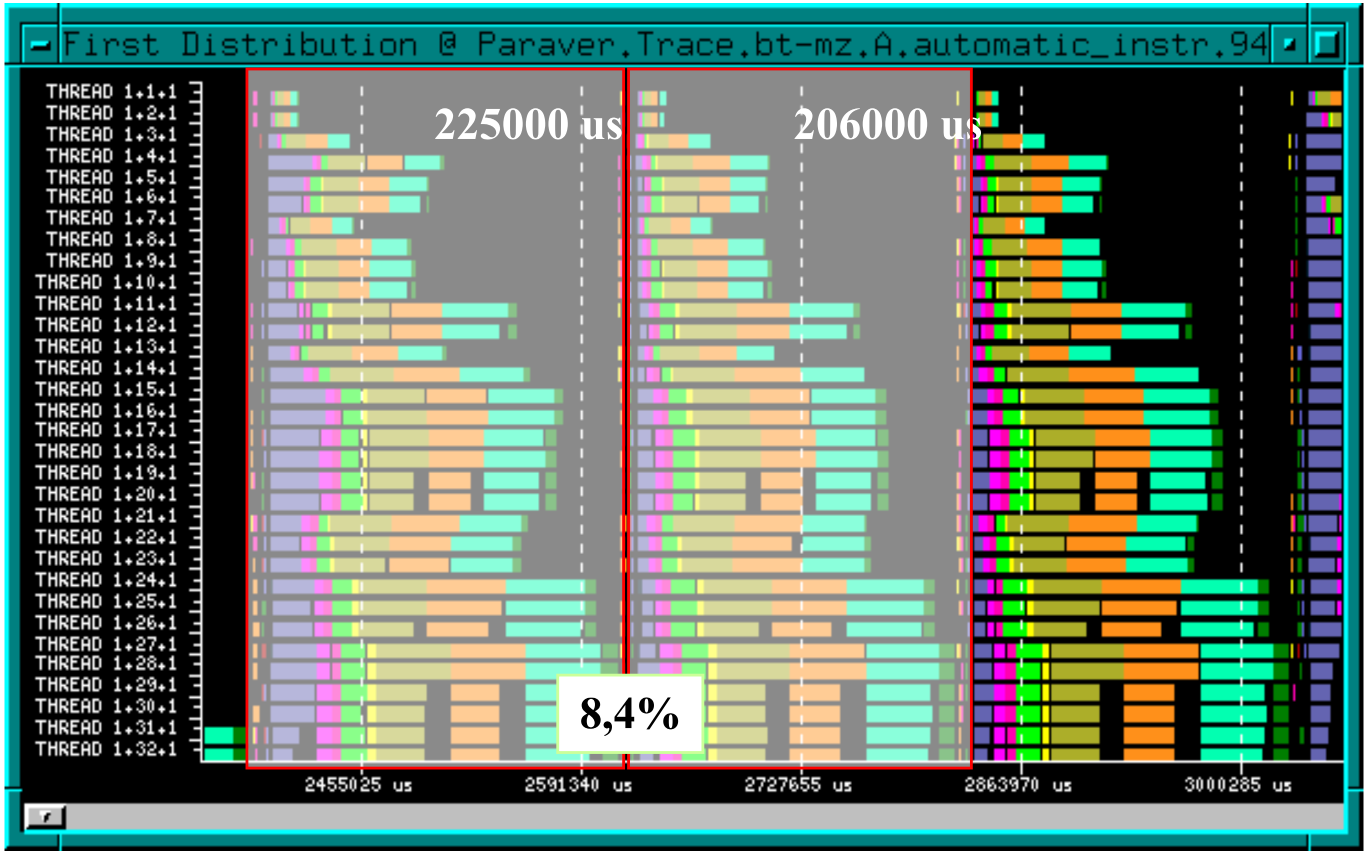
																	16
																58000	1
																69000	1
																122200	1
196000	1	196000	1	196000	1	196000	1	98000	2	98000	2	98000	2	98000	2	98000	2
74400	1	74400	1	74400	1	74400	1	74400	1	74400	1	74400	1	74400	1	74400	1
120000	1	120000	1	120000	1	120000	1	120000	1	120000	1	120000	1	120000	1	120000	1
186000	1	186000	1	186000	1	186000	1	186000	1	93000	2	93000	2	93000	2	93000	2
141000	2	141000	2	141000	2	141000	2	141000	2	141000	2	141000	2	141000	2	141000	2
128000	1	128000	1	128000	1	128000	1	128000	1	128000	1	128000	1	128000	1	128000	1
180000	1	180000	1	180000	1	180000	1	180000	1	180000	1	90000	2	90000	2	90000	2
162000	2	162000	2	162000	2	162000	2	162000	2	162000	2	162000	2	162000	2	162000	2
264000	2	176000	3	176000	3	176000	3	176000	3	176000	3	176000	3	132000	4	132000	4
156000	1	156000	1	156000	1	156000	1	156000	1	156000	1	156000	1	156000	1	156000	1
142000	2	142000	2	142000	2	142000	2	142000	2	142000	2	142000	2	142000	2	142000	2
248000	2	248000	2	165333,3	3	165333,3	3	165333,3	3	165333,3	3	165333,3	3	165333,3	3	165333,3	3
217000	4	217000	4	217000	4	173600	5	173600	5	173600	5	173600	5	173600	5	144666,7	6

Evolution of the distribution algorithm



Cache effects in first transition

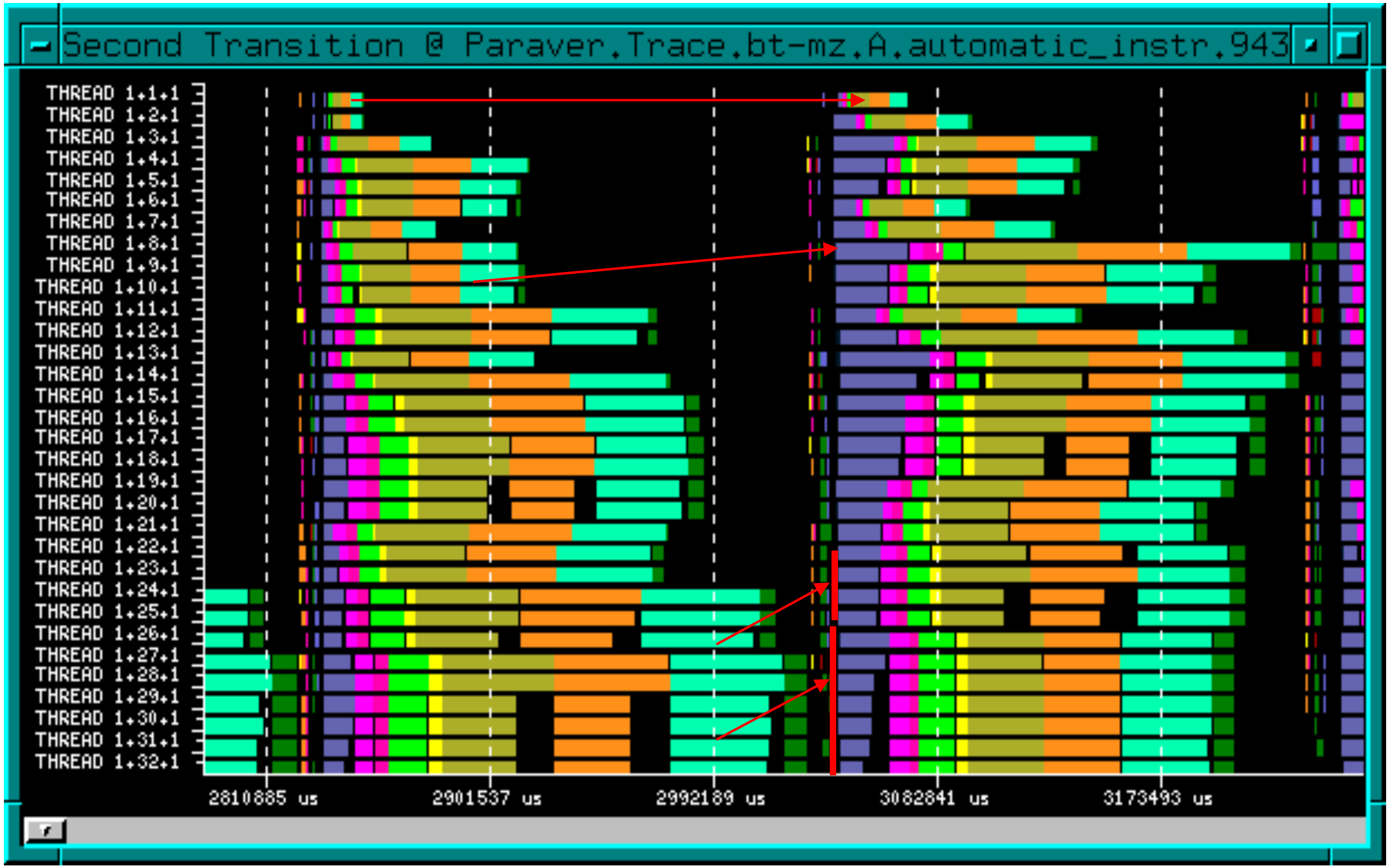
- **Changes in zones**
  - 2, 3, 5, 6, 9, 10, 12, 13, 15, 16



Block	who	howmany	New who	New howmany
1	0	2	0	2
2	2	2	2	1
3	4	2	3	1
4	6	2	4	2
5	8	2	6	1
6	10	2	7	1
7	12	2	8	2
8	14	2	10	2
9	16	2	12	1
10	18	2	13	1
11	20	2	14	2
12	22	2	16	4
13	24	2	20	1
14	26	2	21	2
15	28	2	23	3
16	30	2	26	6

Second transition

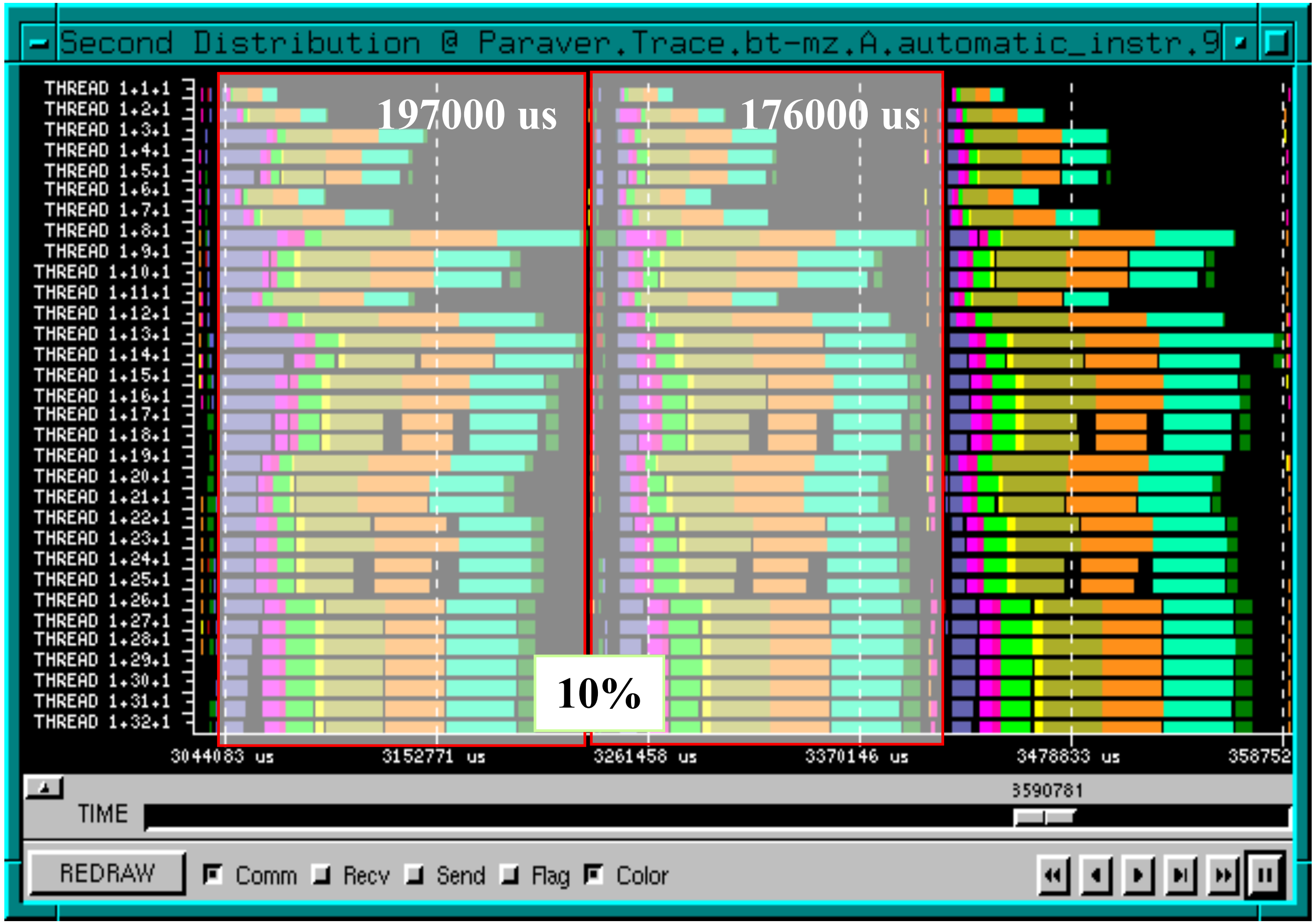
- **Changes in zones**
  - 1, 7, 15, 16



Block	Current	Next
1	2	1
2	1	1
3	1	1
4	2	2
5	1	1
6	1	1
7	2	1
8	2	2
9	1	1
10	1	1
11	2	2
12	4	4
13	1	1
14	2	2
15	3	4
16	6	7

Cache effects in second transition

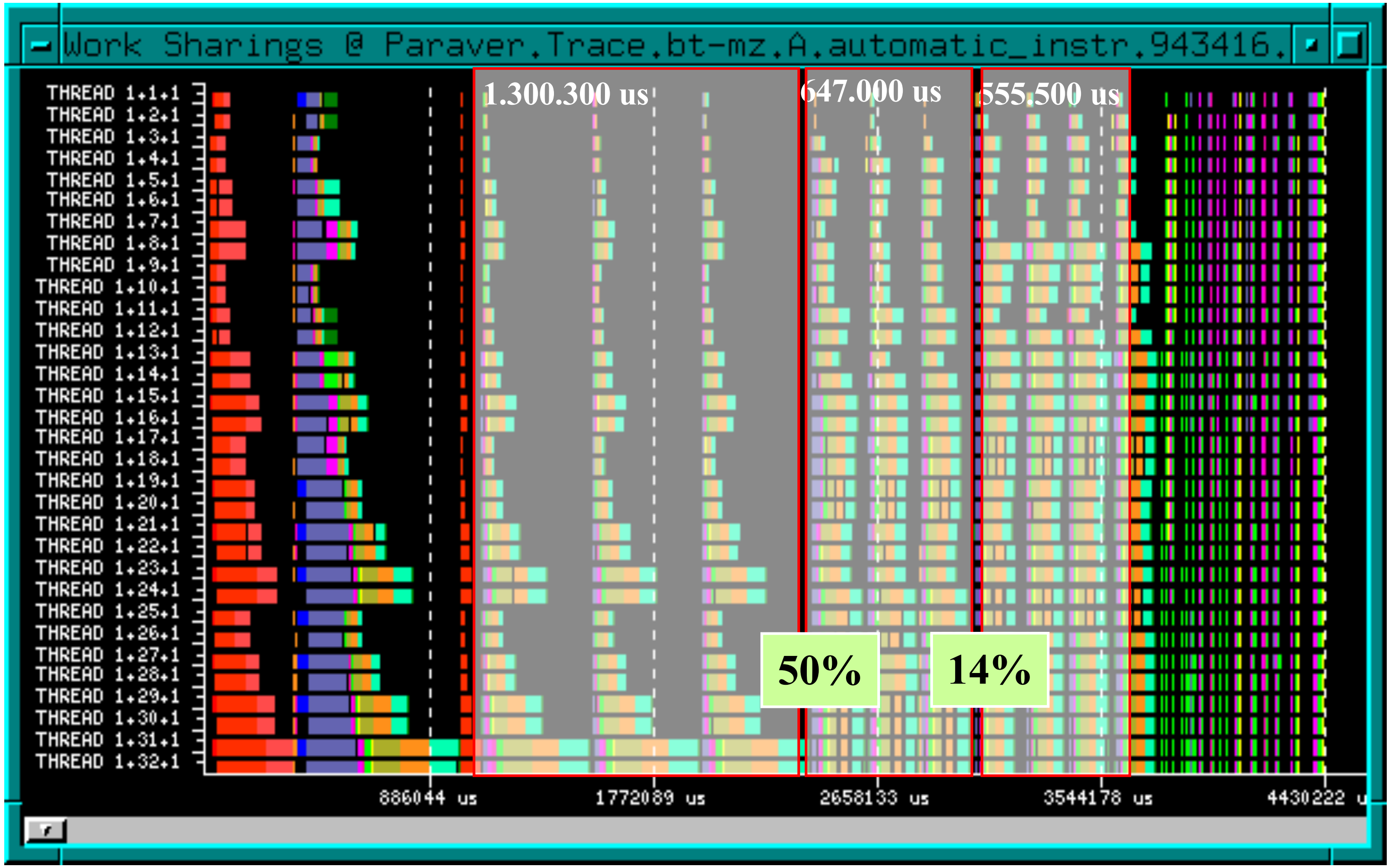
- **Changes in zones**
  - 1, 7, 15, 16



Block	who	howmany	New who	New howmany
1	0	2	0	1
2	2	1	1	1
3	3	1	2	1
4	4	2	3	2
5	6	1	5	1
6	7	1	6	1
7	8	2	7	1
8	10	2	8	2
9	12	1	10	1
10	13	1	11	1
11	14	2	12	2
12	16	4	14	4
13	20	1	18	1
14	21	2	19	2
15	23	3	21	4
16	26	6	25	7

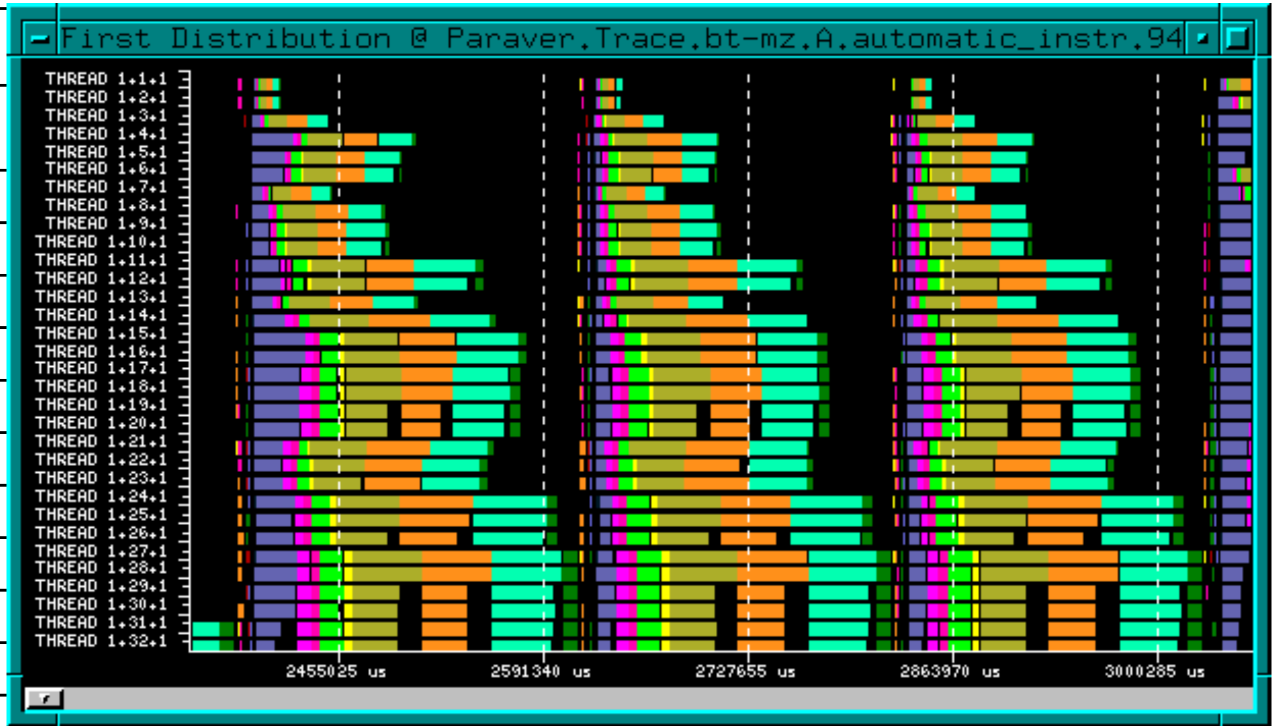
Evolution of the thread distribution

- **BT-MZ 16 zones**
  - 16 threads outermost level



Prediction of gains

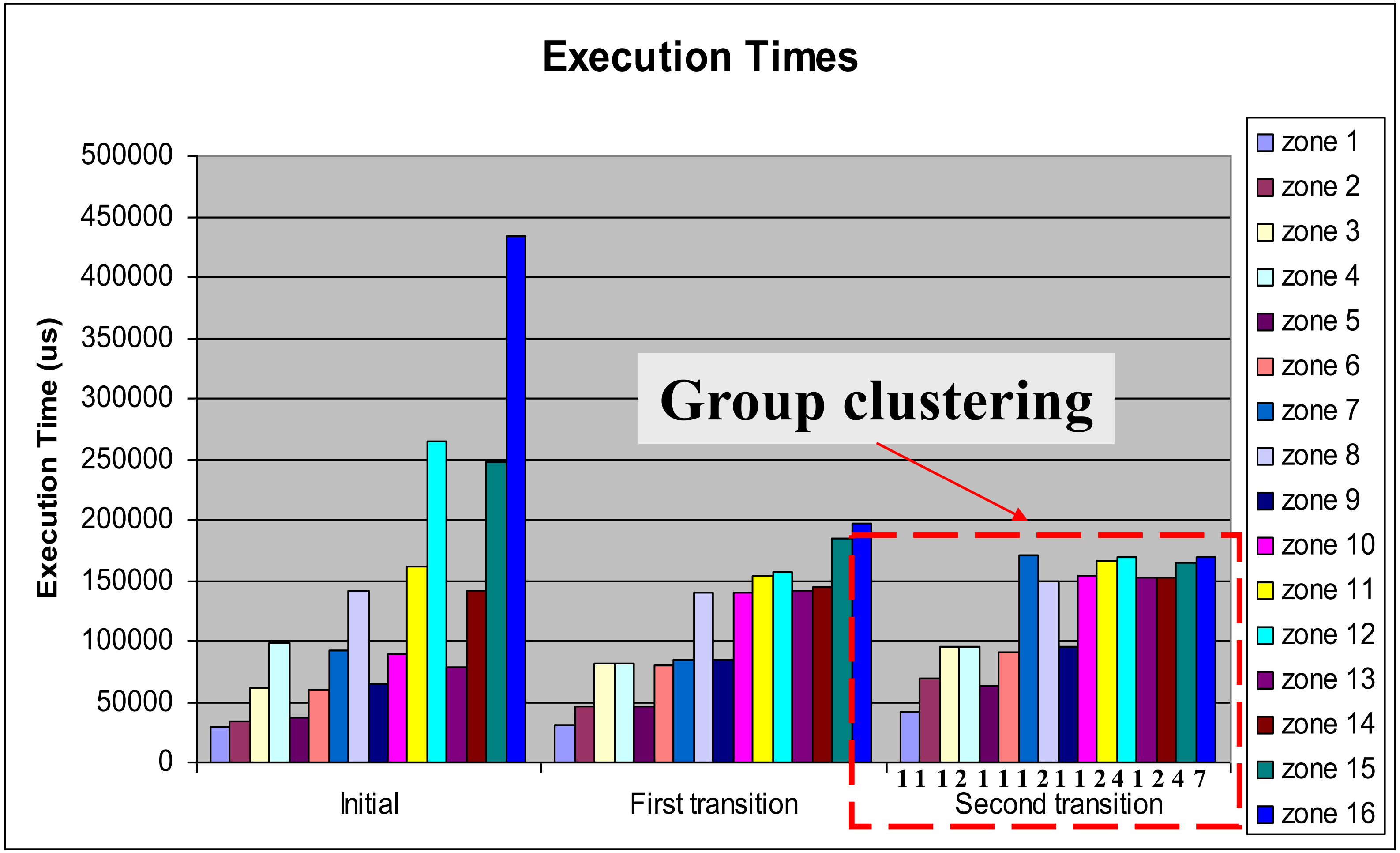
- **Error in prediction: 35% - 40%**
  - Unbalance in inner level



	INITIAL							
Block	Thread Distribution	Measured Execution Times						
1	2	29000						
2	2	34500						
3	2	61100						
4	2	98000						
5	2	37200						
6	2	60000						
7	2	93000						
8	2	141000						
9	2	64000						
10	2	90000						
11	2	162000						
12	2	264000	4	132000	157000	4	157000	170000
13	2	78000	1	160000	142000	1	142000	153000
14	2	142000	2	142000	145000	2	145000	153000
15	2	248000	3	133000	185000	4	138000	164000
16	2	434000	6	144000	197000	7	123000	170000

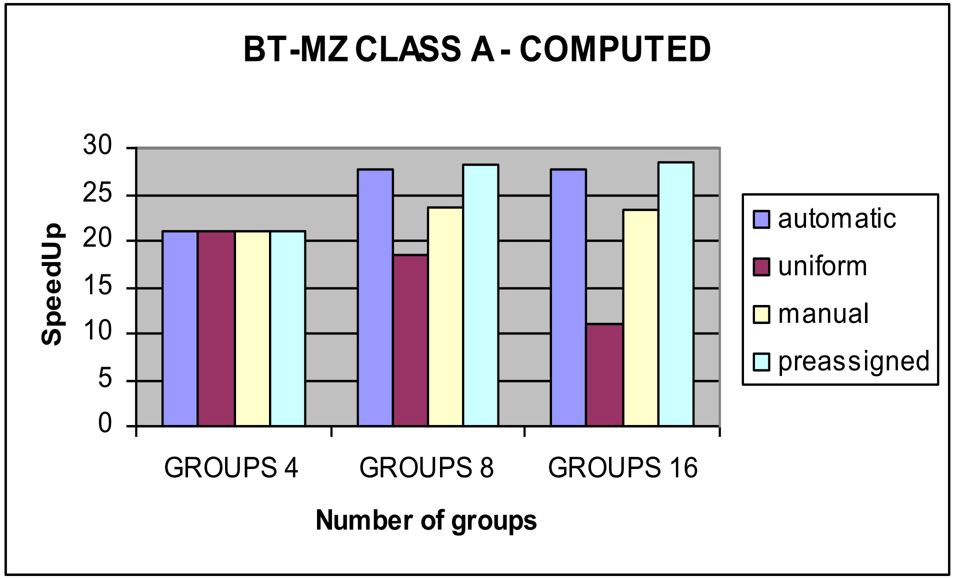
Evaluation

- **Evolution of zone execution times**



Evaluation

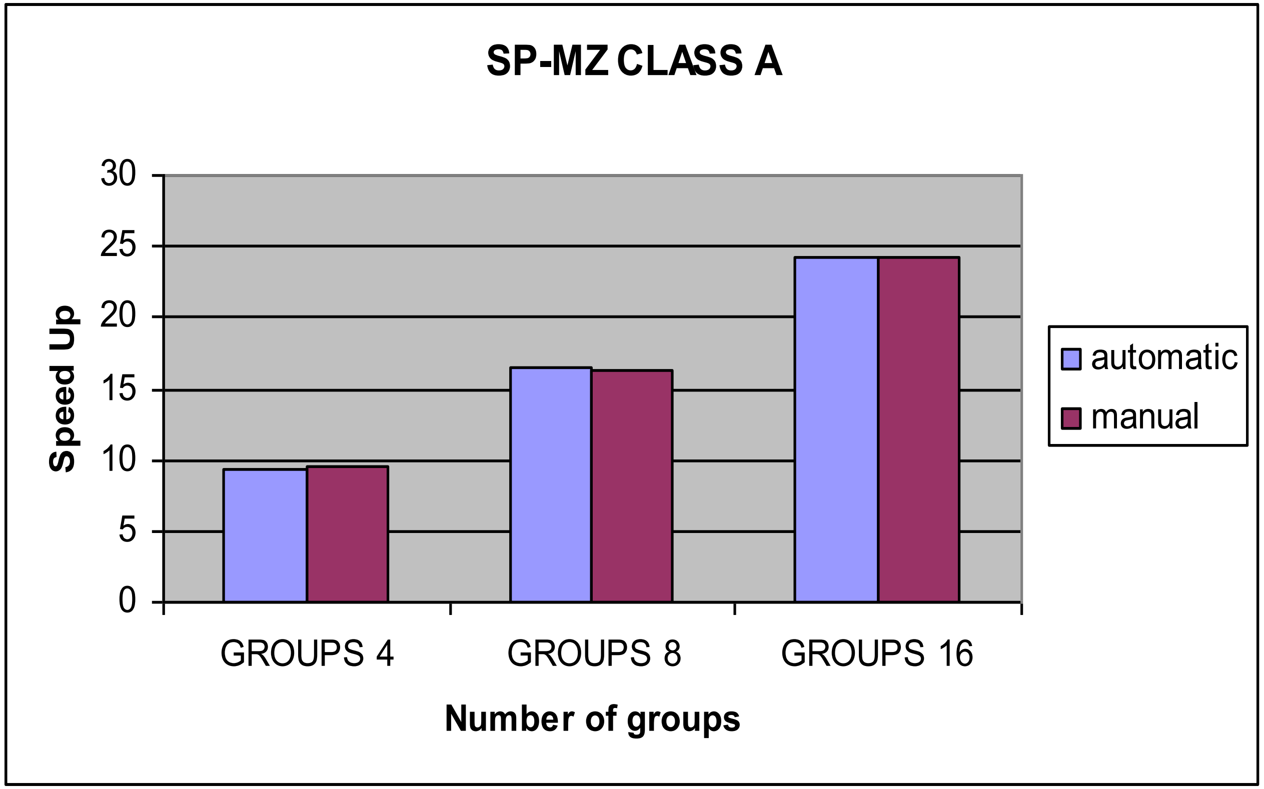
- **Programmer group definition**
  - Uniform
  - Non uniform
- **Automatic group definition**



Block	Proportions Sizes	Threads	Proportions Execution Time	Threads
1	1	1	1	1
2	1.61	1	1,61	1
3	2.76	1	2,76	1
4	4.46	2	4,65	2
5	1.61	1	1,64	1
6	2.61	1	2,69	1
7	4.47	2	4,86	1
8	7.20	2	8,27	2
9	2.76	1	3,01	1
10	4.47	2	4,97	1
11	7.66	2	9,31	2
12	12.35	4	16,10	4
13	4.46	2	5,41	1
14	7.20	2	9,65	2
15	12.35	3	17,24	4
16	19.9	5	29,20	7

Evaluation

- **SP-MZ 16 uniform zones & automatic**
  - 32 threads



END