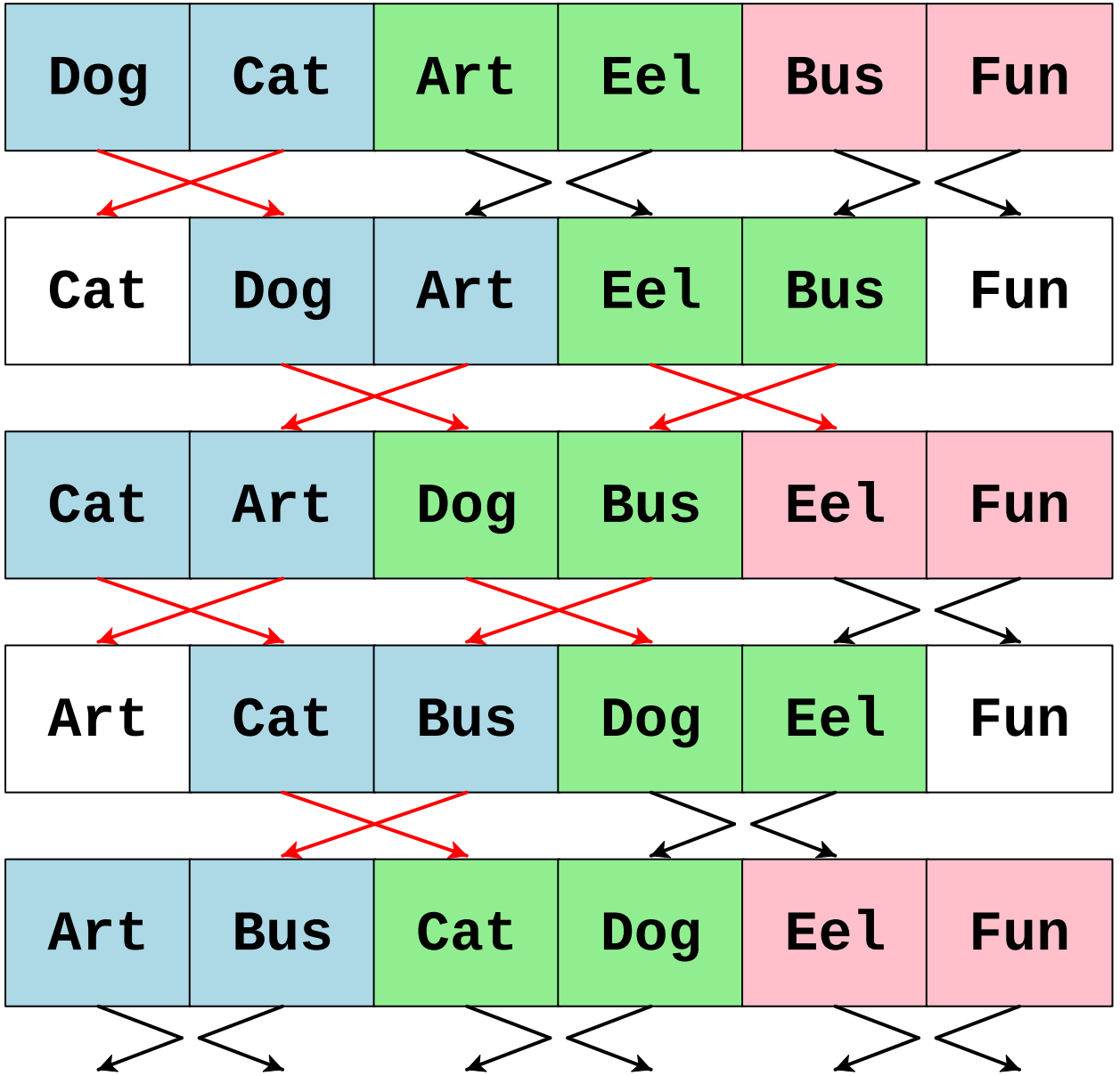
Parallel Sorting

Three solutions to sorting in parallel:

- Unlimited parallelism — similar to bubble sort
- Fixed parallelism — 26 threads, one per letter
- Scalable parallelism — Batcher's Bitonic Sort

Using Unlimited Parallelism



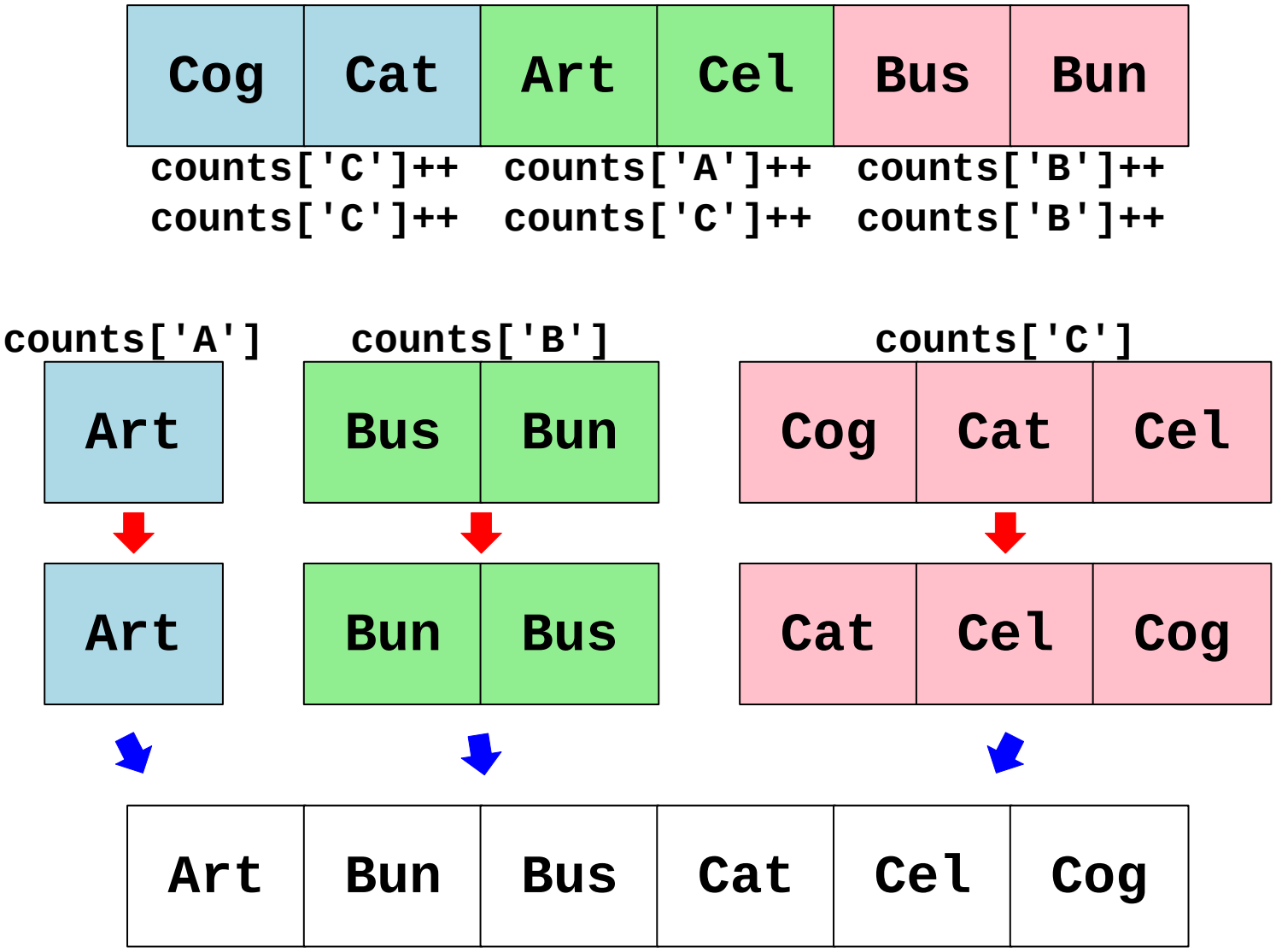
Swap Loop

```
bool continue = true;
rec L[n];

while (continue)
{
  swap_pairs(1);
  swap_pairs(0); /* sets continue */
}
```

Swaping Pairs

```
void swap_pairs(int start)
{
  forall(i in(start...n-2:2)) /* stride by 2 */
  {
    rec temp;
    bool done = true;
    if (strcmp(L[i].x, L[i+1].x) > 0) {
      temp = L[i];
      L[i] = L[i+1];
      L[i+1] = temp;
      done = false;
    }
    if (!start) continue = !(&&/done);
  }
}
```

Fixed Parallelism



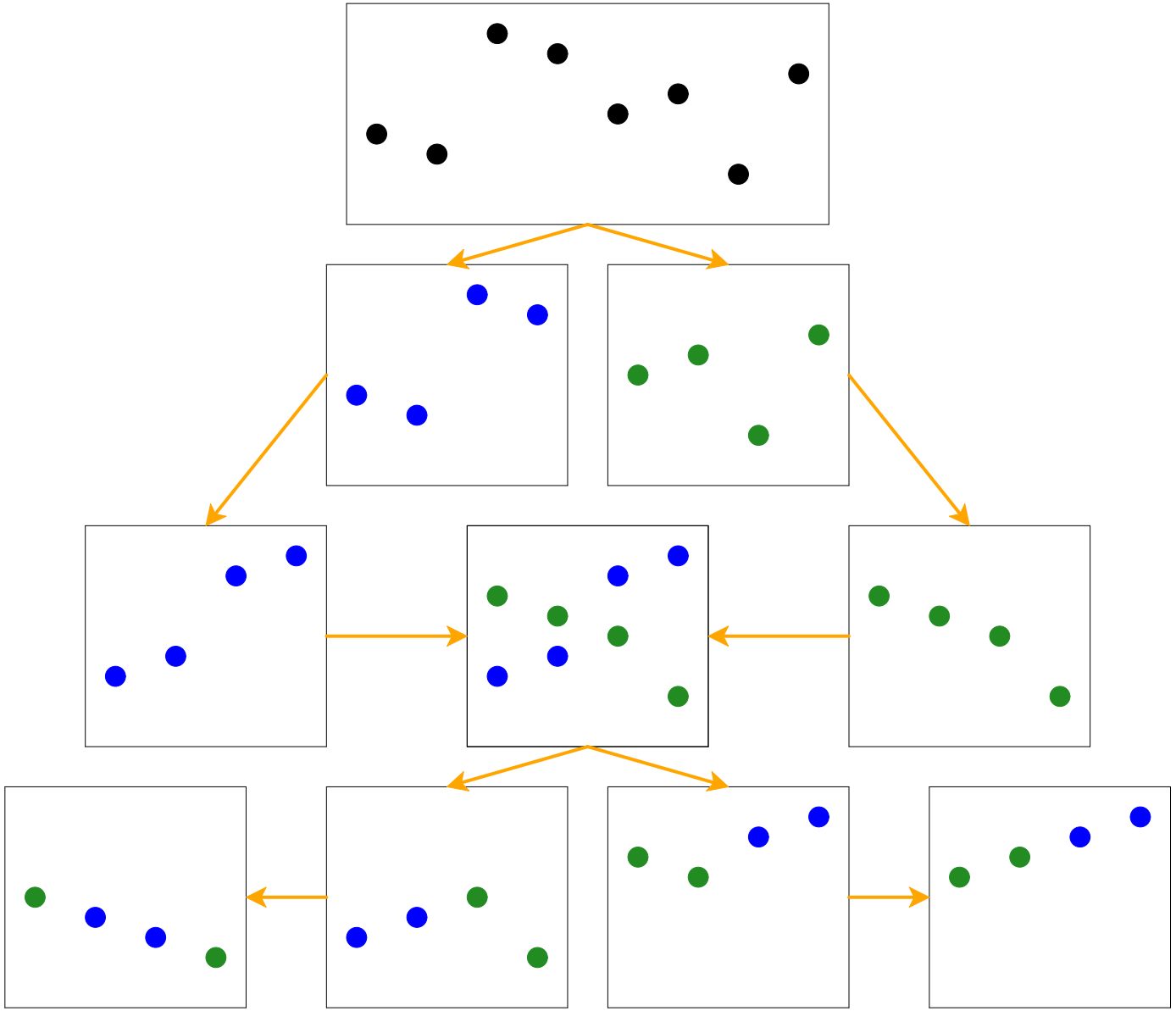
Counting

```
rec L[n];
forall(index in(0..25)) {
  int size = mySize(L, 0);
  rec myL[] = localize(L), temp[];
  int counts[26] = 0, myCount, myStart;

  for (int i = 0; i < size; i++)
    counts[letRank(myL[i].x[0])]++;

  myCount = +/counts[index]; /* implies sync */

  temp = localSort(index, myCount);

  myStart = (+\myCount) - myCount;

  for (int i = 0; i < myCount; i++)
    L[i + myStart] = temp[i];
}
```

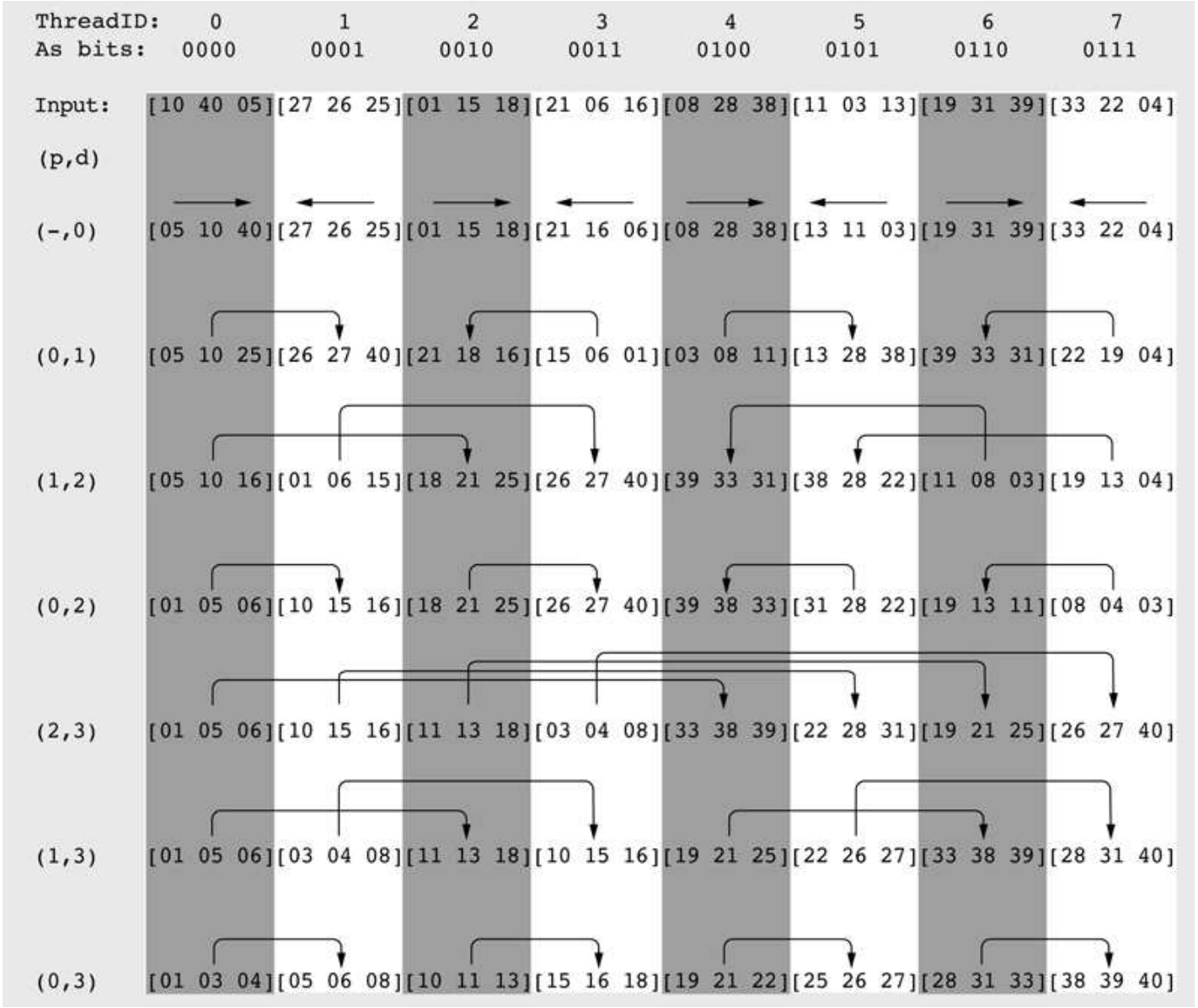
Local Sorting

```
rec[] localSort(int index, int myCount)
{
   rec temp[] = new rec[myCount];
   int j = 0;

   for (int i = 0; i < n; i++)
     if (letRank(L[i].x[0]) == index)
       temp[j++] = L[i];

   alphabetizeInPlace(temp, myCount);

   return temp;
}
```

Scalable Parallelism



Scalable Parallelism



Batcher's Setup

```
rec L[n];

int t;
int m = log2(t);

int size = n / t;

typedef struct { char x[MAXLEN]; int home; } key;
key BufK[m][size];
bool free'[m] = false, ready'[m];
```

Assume **L** localized into equal contiguous segments

Assume **BufK** localized by first index

Batcher's Thread Outline

```
forall (index in(0..t-1)) {
  rec myL[size] = localize(L), inputCopy[size];
  key Kn[][] = localize(BufK), K[size];

  for (int i = 0; i < size; i++) {
    K[i].x = myL[i].x;
    K[i].home = localToGobal(myL, i, 0);
  }

  alphabetizeInPlace(K, size, bit(index, 0) /* up or down */);

  .... /* main loop */

  for (int i = 0; i < size; i++)
    inputCopy[i] = L[K[i].home];
  barrier;
  for (int i = 0; i < size; i++)
    myL[i] = inputCopy[i];
}
```

Batcher's Thread Main Loop Outline

```
for (int d = 1; d <= m; d++) {
  for (p = d - 1; p > 0; p--) {
    free'[neigh(index, p)]; /* wait to send */

    for (int i = 0; i < size; i++)
      BufK[neigh(index, p)][i] = K[i];

    ready'[neigh(index, p)] = true; /* sent */

    ready'[index]; /* wait to receive */

    merge(index, d, p, Kn, K);
    realphabetizeInPlace(K, size, bit(index, p));

    free'[index] = true; /* ready to receive */
  }
}
```

Batcher's Merge

```
void merge(int index, int d, int p, key Kn[][], key K[])
{
  for (int i = 0; i < size; i++) {
    bool want;
    int cmp;

    cmp = strcmp(Kn[index][i].x, K[i].x);

    if (bit(index, d) == bit(index, p))
      want = (cmp > 0);
    else
      want = (cmp < 0);

    if (want)
      K[i] = Kn[i];
  }
}
```