Divide-and-Conquer Parallelization Paradigm

Divide-and-Conquer Simulation Algorithms

- **Divide-and-conquer (DC) algorithms:** Recursively partition a problem into subprogram of roughly equal size. If subprogram can be solved independently, there is a possibility of significant speed up by parallel computing.

The DC paradigm has been used widely to design a number of efficient algorithms for broad scientific and engineering applications. Examples include a hierarchy of particle simulation methods, in which spatially localized subproblems are solved in a global embedding field, which is efficiently computed with tree-based algorithms (see Figure). Examples of the embedding field are the electrostatic field in molecular dynamics (MD) simulations[1] and the self-consistent Kohn-Sham potential in quantum mechanical (QM) simulations in the framework of the density functional theory (DFT).[2]

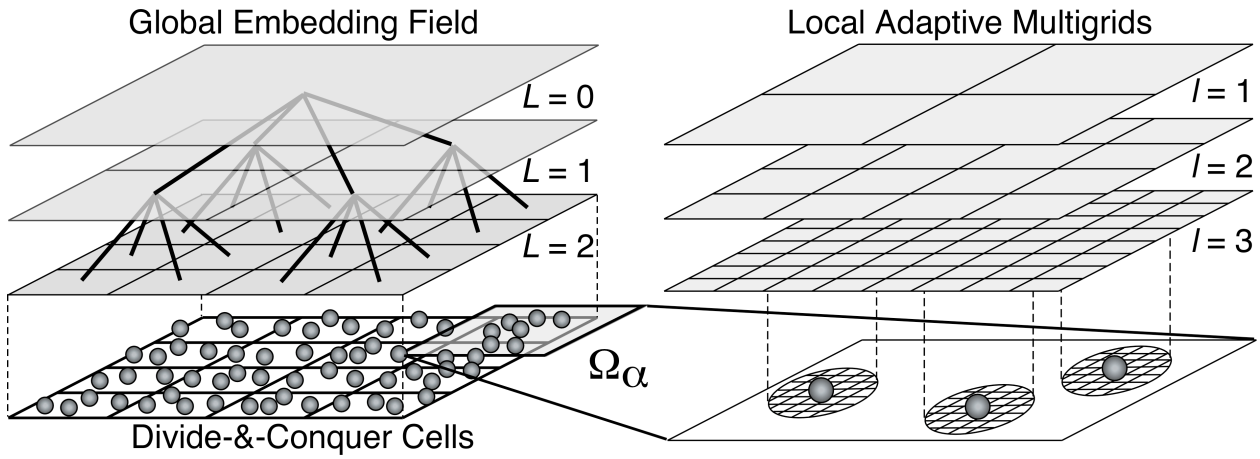


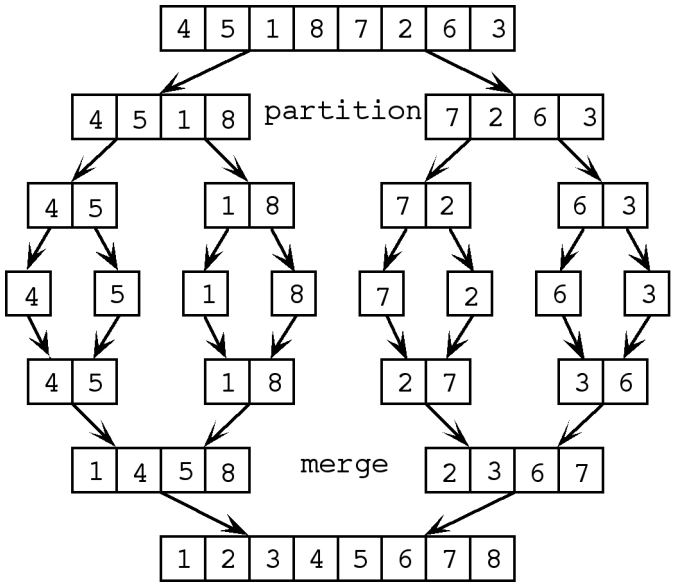
Figure. Schematic of a divide-and-conquer (DC) particle simulation algorithm. (Left) The physical space is subdivided into spatially localized cells, with local particles constituting subproblems (bottom), which are embedded in a global field (shaded) solved with a tree-based algorithm. (Right) To solve the subproblem in domain Ω_α in the DC-DFT algorithm, coarse multigrids (gray) are used to accelerate iterative solutions on the original real-space grid (corresponding to the grid refinement level, $l = 3$). The bottom panel shows fine grids adaptively generated near the atoms (spheres) to accurately operate the ionic pseudopotentials on the electronic wave functions.

In the following, we use the sorting problem as an example to practice the programming of parallel divide-and-conquer algorithms and examine their communication patterns.

Divide-and-Conquer Paradigm I: Parallel Bitonic Mergesort

MERGE SORT

- **Merge:** Given two ascending sorted sublists, $L[0{:}N\text{-}1]$ and $L[N{:}2N\text{-}1]$, obtain a combined ascending sorted list, $L[0{:}2N]$.
- **Recursive merge sort:** Given a list, divide it in half, sort the two halves (recursively), and then merge the two halves together.




```
void mergesort(int list[],int left,int right)
{
  int i,j,k,t,middle,temp[N];

  if (left < right) {
    middle = (left + right)/2;
    mergesort(list,left,middle);
    mergesort(list,middle+1,right);

    k = i = left; j = middle+1;
    while (i<=middle && j<=right)
      temp[k++] = list[i]<list[j] ? list[i++] : list[j++];
    t = i>middle ? j : i;
    while (k <= right) temp[k++] = list[t++];
    for (k=left; k<=right; k++) list[k] = temp[k];
  }
}
```

Note that this algorithm would incur too much communication on parallel computers. In the following, we introduce another sorting algorithm that is better suited for parallel implementation.

BITONIC SPLIT

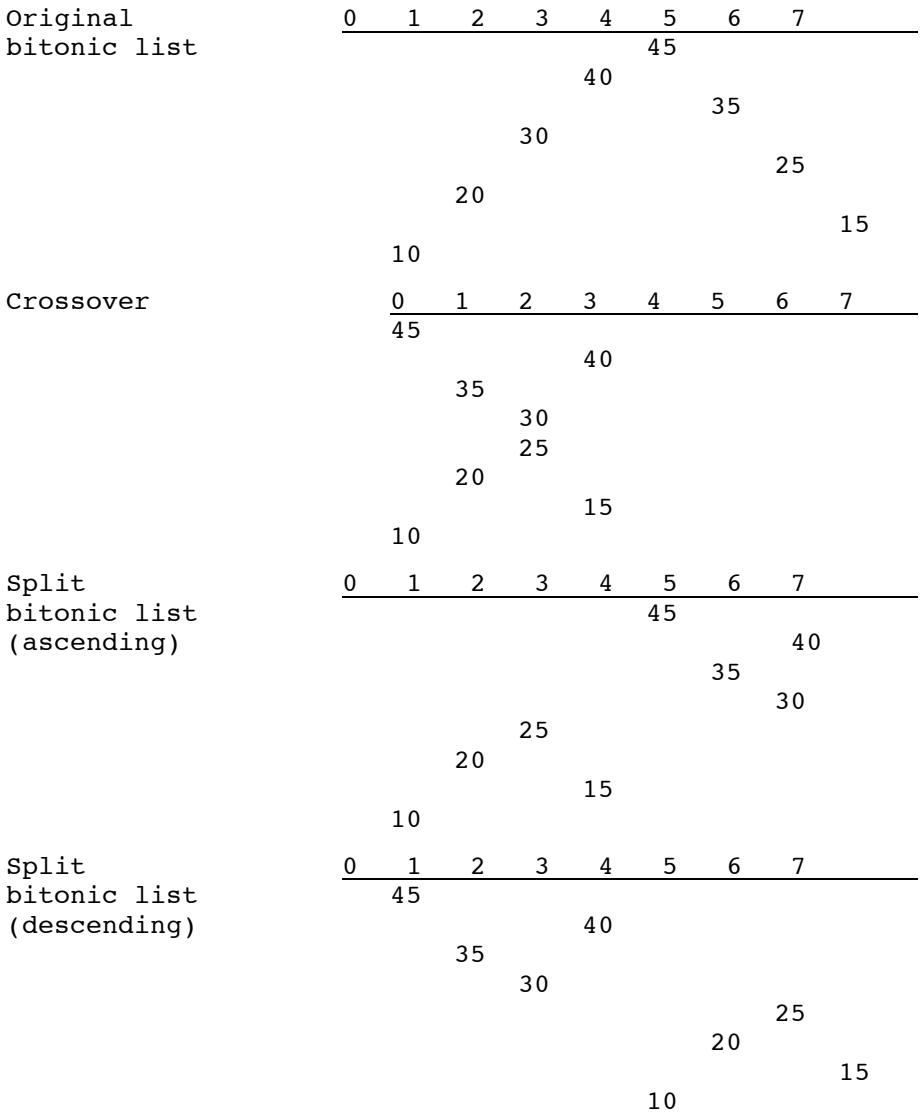
- **Bitonic list:** A list with no more than one local maxima and no more than one local minima. One important type of bitonic list has the first half sorted in ascending (descending) order and the second half in descending (ascending) order.
- **Bitonic split** is defined as follows:

  1. Each element in the first half of the list is assigned a partner, which is the same relative position from the second half of the list.
  2. Each pair of partner compares themselves. If the first half of the list has a larger (smaller) item, then exchange them.

  When applied to a bitonic list of length *N*, the bitonic split results in a new list with the following properties:

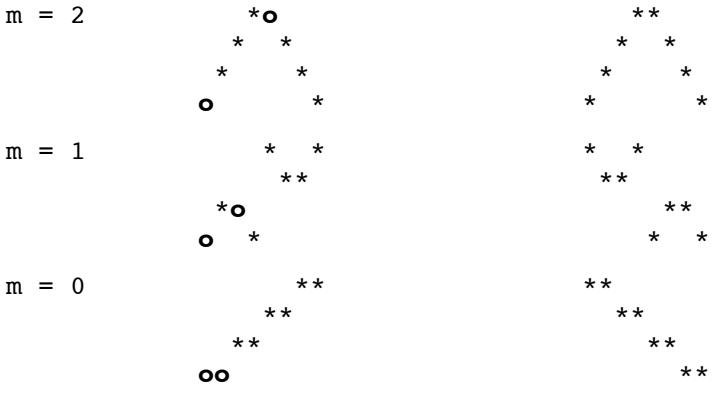
  1. Each item in the first half of the list is less than every item in the second half.
  2. The first half and the second half of the list are each a bitonic list of length *N*/2.

Example: bitonic split



BITONIC MERGE = RECURSIVE BITONIC SPLIT

The list is fully sorted after log*N* levels of recursive application of bitonic split.



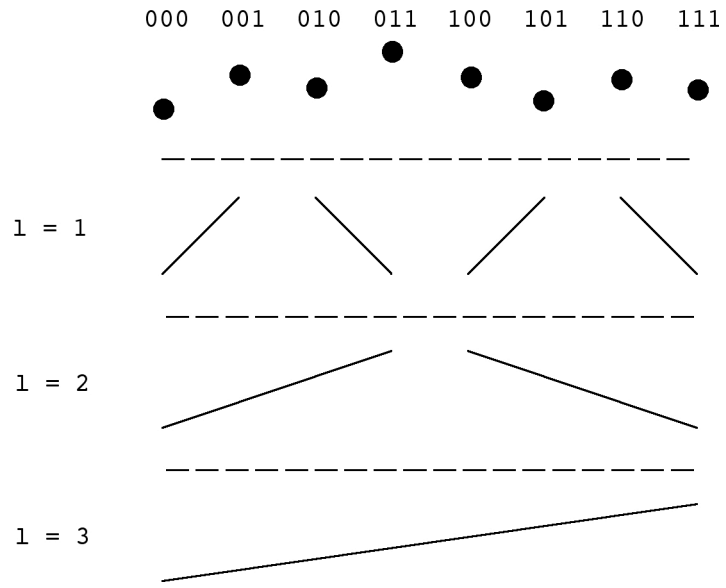
BITONIC MERGE IN PARALLEL

```
for m := dim-1 downto 0 do
begin
  partner := me XOR 2^m;
  if me AND 2^m = 0 then
    retain smaller (larger) of the two items
  else
    retain larger (smaller) of the two items
end;
```

Here, *me* = 0:*N*-1, and XOR and AND are bitwise exclusive OR and AND operators. Note that taking XOR with 1 flips a bit (i.e., **0** XOR 1 = **1** and **1** XOR 1 = **0**).

BITONIC MERGE SORT

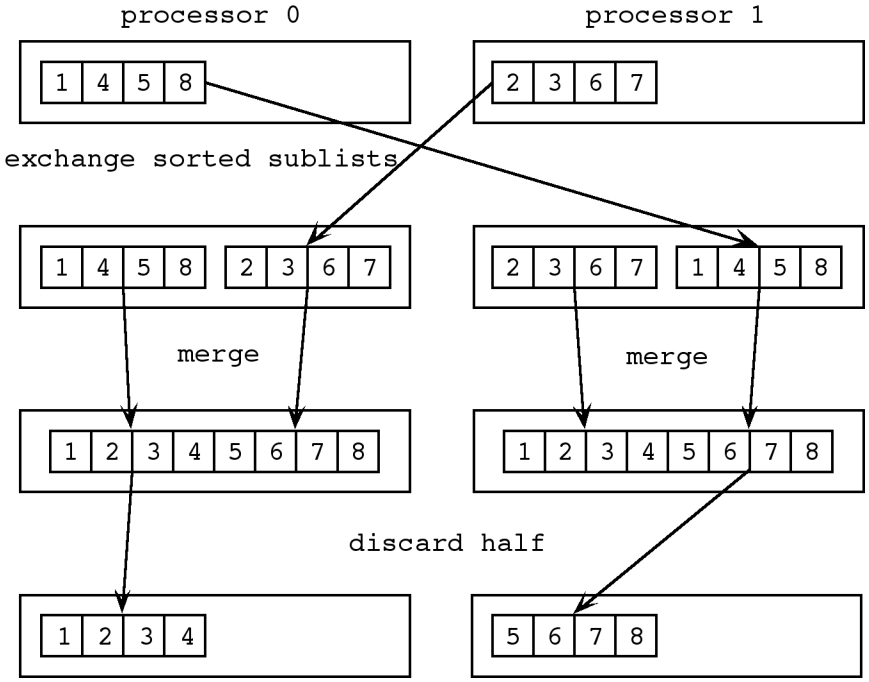
Consider an unsorted list with $N = 2^{dim}$ items. Any list with only two items is a bitonic list. Therefore, this unsorted list consists of *N*/2 bitonic lists of length 2. By applying the bitonic merge to pairs of adjacent lists, the result is *N*/4 bitonic lists of length 4. After log*N* repetitions of the bitonic merge, the list is completely sorted.



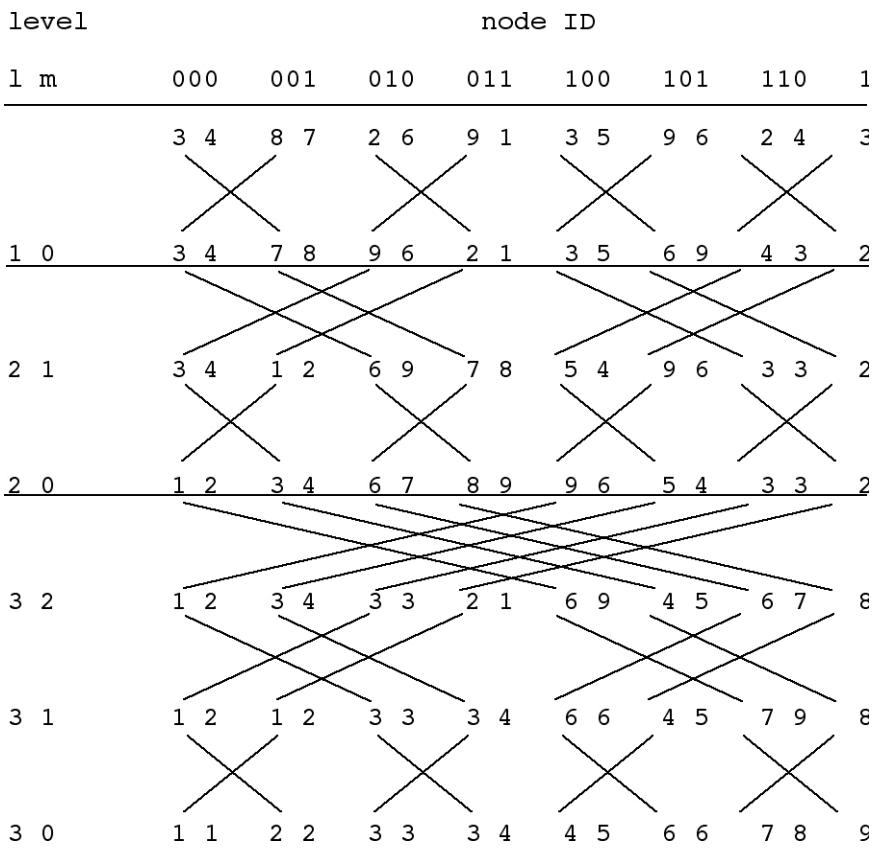
Bitonic Merge Sort in Parallel

```
for L := 1 to dim do
  for m := L-1 downto 0 do
  begin
    partner := myid XOR 2^m;
    if me AND 2^L = 0 then
      ascending merge sort for me ∪ partner
    else
      ascending merge sort for me ∪ partner;
    if me AND 2^m = 0 then
      retain the first half of the merged list
    else
      retain the second half of the two items
  end;
```

COMPARE-EXCHANGE OPERATION FOR BLOCK PARALLEL SORT



Example: parallel block bitonic merge sort



- **Program `pbmerge.c` using MPI**

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

#define N 1024
#define MAX 99

int nprocs,dim,myid; /* Cube size, dimension, & my node ID */

/* Sequential mergesort (either ascending or descending) */
void mergesort(int list[],int left,int right,int descending)
{
  int i,j,k,t,middle,temp[N];

  if (left < right) {
    middle = (left + right)/2;
    mergesort(list,left,middle,descending);
    mergesort(list,middle+1,right,descending);

    k = i = left; j = middle+1;
    if (descending)
      while (i<=middle && j<=right)
        temp[k++] = list[i]>list[j] ? list[i++] : list[j++];
    else
      while (i<=middle && j<=right)
        temp[k++] = list[i]<list[j] ? list[i++] : list[j++];
    t = i>middle ? j : i;
    while (k <= right) temp[k++] = list[t++];
    for (k=left; k<=right; k++) list[k] = temp[k];
  }
}

/* Parallel mergesort */
void parallel_mergesort(int myid,int list[],int n)
{
  int listsize, l, m, bitl = 1, bitm, partner, i;
  MPI_Status status;

  listsize = n/nprocs;
  mergesort(list,0,listsize-1,myid & bitl);

  for (l=1; l<=dim; l++) {
    bitl = bitl << 1;
    for (bitm=1, m=0; m<l-1; m++) bitm *= 2;
    for (m=l-1; m>=0; m--) {
      partner = myid ^ bitm;
      MPI_Send(list,listsize,MPI_INT,partner,l*dim+m,MPI_COMM_WORLD);
      MPI_Recv(&list[listsize],listsize,MPI_INT,partner,l*dim+m,
               MPI_COMM_WORLD,&status);
      mergesort(list,0,2*listsize-1,myid & bitl);
      if (myid & bitm)
        for (i=0; i<listsize; i++) list[i] = list[i+listsize];
      bitm = bitm >> 1;
    }
  }
}
```

```
int main(int argc,char *argv[])
{
  int list[N],n=16,i;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  dim = log(nprocs+1e-10)/log(2.0);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  srand((unsigned) myid+1);
  for (i=0; i<n/nprocs; i++) list[i] = rand()%MAX;

  printf("Before: Node %2d :",myid);
  for (i=0; i<n/nprocs; i++) printf("%3d ",list[i]);
  printf("\n");

  parallel_mergesort(myid,list,n);

  printf("After:  Node %2d :",myid);
  for (i=0; i<n/nprocs; i++) printf("%3d ",list[i]);
  printf("\n");

  MPI_Finalize();
  return 0;
}
```

Divide-and-Conquer Paradigm II: Hypercube Quicksort

QUICKSORT

Quicksort is a "divide-and-conquer" method for sorting. It works by partitioning a file into two parts, then sorting the parts independently.

```
quicksort(int list[],int left,int right)
{
  int j;

  if (left < right) {
    j = partition(list,left,right);
    quicksort(list,left,j-1);
    quicksort(list,j+1,right);
  }
}
```

The `partition` procedure works as follows: Given a sublist `list[left:right]`, it first chooses the left-most element as a `pivot`. When returned, the `pivot` element is placed at the `j`-th position, and: i) `a[left],...,a[j-1]` are less than or equal to `a[j]`; ii) `a[j+1],...,a[right]` are greater than or equal to `a[j]`.

```
  0    1    2    3    4    5    6    7    8    9
[5    7    2    9    6    8    3    4    1    0]
[3    0    2    1    4]   5   [8    6    9    7]
[1    0    2]   3   [4]   5   [7    6]   8   [9]
[0]   1   [2]   3   [4]   5   [6]   7 [] 8   [9]
```

```
void quicksort(int list[],int left,int right)
{
  int pivot,i,j;
  int temp;

  if (left < right) {
    i = left; j = right + 1;
    pivot = list[left];
    do {
      while (list[++i] < pivot && i <= right);
      while (list[--j] > pivot);
      if (i < j) {
        temp = list[i]; list[i] = list[j]; list[j] = temp;
      }
    } while (i < j);
    temp = list[left]; list[left] = list[j]; list[j] = temp;
    quicksort(list,left,j-1);
    quicksort(list,j+1,right);
  }
}
```

HYPERCUBE QUICKSORT

Let n be the number of elements to be sorted and $p = 2^d$ be the number of processors in a d-dimensional hypercube (see the Appendix for the definition of a hypercube). Each processor is assigned a block of n/p elements.

The algorithm starts by selecting a common pivot value, which is broadcast to all processors. Each processor partitions its local elements into two blocks, one with elements smaller than the pivot, and the other with elements larger than the pivot. Then the processors connected along the d-th communication link exchange blocks: Each processor with a 0 in the d-th bit retains the smaller elements, and each processor with a 1 in the d-th bit retains the larger elements. After this step, each processor in the (d-1)-dimensional hypercube whose d-th bit is 0 has elements smaller than the pivot, and each processor in the other (d-1)-dimensional hypercube has elements larger than the pivot.

At the next level, a pivot is chosen in each (d-1)-dimensional hypercube separately, and it is broadcast to all the processors in each hypercube. Each processor partitions its local elements into two blocks, one smaller and the other larger than the pivot. Appropriate blocks are exchanged through the (d-1)-th communication link so that each processor with a 0 in the (d-1)-th bit retains the smaller elements than the pivot, and each processor with a 1 in the (d-1)-th bit retains the larger.

This procedure is performed recursively. After d such splits, the sequence is sorted with respect to the global ordering imposed on the processors. Then each processor sorts its local elements by using sequential quicksort.

1.
Dimension	Master of Subcube
3	000
2	x00
1	xx0

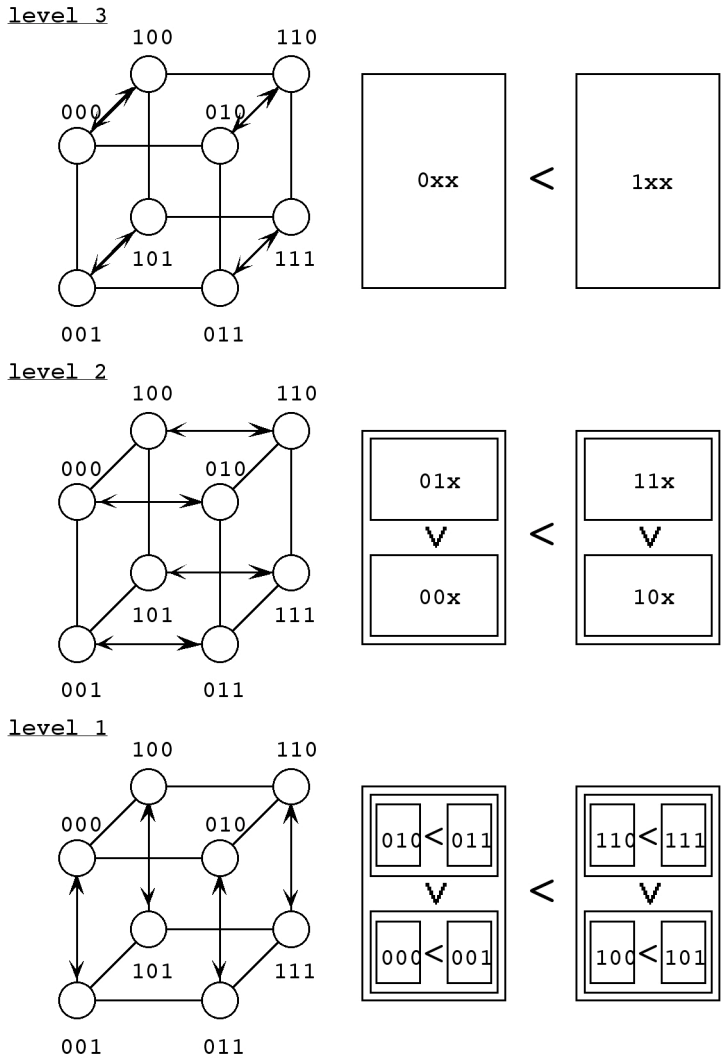
2. Partner is obtained by flipping the d-th bit.

PIVOT SELECTION

Master of each subcube determines the pivot value and broadcasts it to all the processors in the subcube. Bad choice of pivot at early stages degrades the performance significantly (no recovery from it). Let us use the average value elements in the master processor as a pivot.

pivot = (Σ elements) / (number of elements)

EXAMPLE: 3D HYPERCUBE QUICKSORT




```
{Hypercube Quicksort}
bitvalue := 2^(dimension-1);
mask := 2^dimension - 1;
for L := dimension downto 1
begin
  if myid AND mask = 0 then
    choose a pivot value for the L-dimensional subcube;
  broadcast the pivot from the master to the other members of the subcube;

  partition list[0:n_element-1] into two sublists such that
  list[0:j] ≤ pivot < list[j+1:n_element-1];
```

```
  partner := myid XOR bitvalue;
  if myid AND bitvalue = 0 then
    begin
      send the right sublist list[j+1:n_element-1] to partner;
      receive the left sublist of partner;
      append the received list to my left list
    end
  else
    begin
      send the left sublist list[0:j] to partner;
      receive the right sublist of partner;
      append the received list to my right list
    end
  n_element := n_element - n_send + n_receive;
  mask = mask XOR bitvalue;
  bitvalue = bitvalue/2;
end

sequential quicksort to list[0:n_element-1]
```

BROADCASTING TO A SUBCUBE

```
  bitvalue = nprocs >> 1;
  mask = nprocs - 1;

  for (L=dimension; L>=1; L--) {
    ...
    if ((myid & mask) == 0)
      Calculate the pivot as the average of the local list element values
    MPI_Bcast(&pivot,1,MPI_INT,0,cube[L][myid/nprocs_cube]);
    ...
    mask = mask ^ bitvalue;   /* Flip the current bit to 0 */
    bitvalue = bitvalue >> 1; /* Next significant bit      */
  }
```

For L dimension, we define $2^{dimension-L}$ subcubes, each containing $nprocs\_cube = 2^L$ processes. A hierarchy of subcubes can be implemented as MPI communicators, `cube[L][c]`, by nested calls to `MPI_Comm_create()`. Specifically `cube[dimension][0]` = `MPI_COMM_WORLD`, and `cube[L][c]` is decomposed into two communicators of the same size, `cube[L-1][2*c]`and `cube[L-1][2*c+1]`, which contain the lower- and upper-half processes (in terms of the rank) of `cube[L][c]`, respectively. At level L, the p-th process (p is the rank in `MPI_COMM_WORLD`) belongs to the c-th subcube, `cube[L][c]`, where $c = p/nprocs\_cube$.

References

1. L. Greengard and V. Rokhlin, "a fast algorithm for particle simulations," *J. Comput. Phys.* **73**, 325 (1987).
2. F. Shimojo, R. K. Kalia, A. Nakano, and P. Vashishta, "embedded divide-and-conquer algorithm on hierarchical real-space grids: parallel molecular dynamics simulation based on linear-scaling density functional theory," *Comput. Phys. Commun.* **167**, 151 (2005).

Appendix: Hypercube Topology

As we have seen, divide-and-conquer algorithms are often implemented conveniently with the hypercube topology. This section formally defines the hypercube topology.

NETWORK TOPOLOGY

- **Topology:** Which processors are directly connected to which other processors.
- **Distance:** The number of communication links a message must traverse between two processors in the most direct path.
- **Diameter:** The maximum distance between any two processors in the network. The diameter measures the maximum delay for transmitting a message from one processor to another.
- **Connectivity:** The number of incident links on each interface. High connectivity is desirable, because it lowers contention for communication resources, but it also increases the cost.
- **Bisection width:** The minimum number of communication links that have to be removed to partition the network into two equal halves. The bisection width measures the largest number of messages, which can be sent simultaneously.

HYPERCUBE TOPOLOGY

A d-dimensional hypercube consists of $n = 2^d$ processors. Each processor has a number whose binary representation has d digits.

- **Hamming distance:** The total number of bit positions at which two binary numbers differ.

In a hypercube, two processors are connected if their Hamming distance is 1. The connectivity of a d-dimensional hypercube is thus d. Since each link can change only one digit, the diameter is d, or $\log_2 n$.

A hypercube topology is constructed recursively as follows.

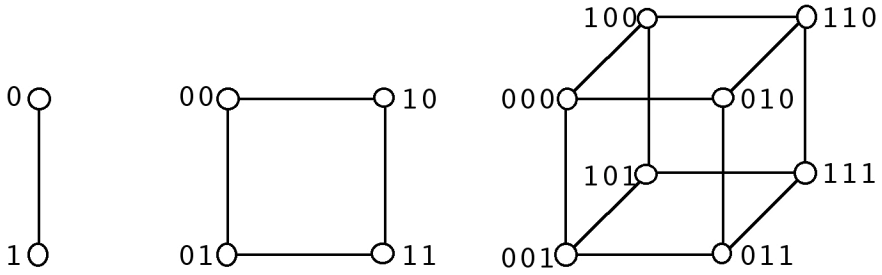
(1) First a one-dimensional hypercube has two connected processors 0 and 1.

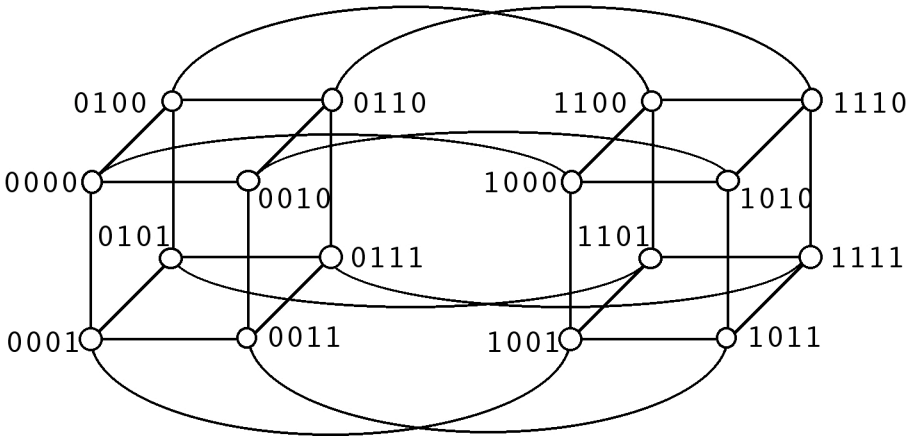
(2) A (d+1)-dimensional hypercube is defined from a d-dimensional hypercube as follows:

a. Duplicate the d-dimensional hypercube including processor numbers.

b. Create links between processors with the same number in the original and duplicate.

c. Append a binary 1 to the left of each processor number in the duplicate, and a binary 0 to left of each processor number in the original.





HYPERCUBE EMBEDDINGS

Each parallel algorithm has its own natural communication structure when executed on a multicomputer (or a distributed-memory computer). If this **logical topology** of the algorithm matches the **physical topology** of the multicomputer, then performance is enhanced.

- **Topological embedding:** A topology X can be embedded in a topology Y, such that every communication link in X has a corresponding communication link in Y.
- **Gray code:** A sequence of numbers such that each successive numbers have Hamming distance 1.

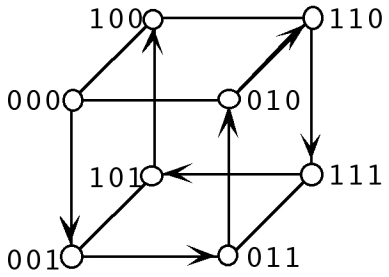
  The *k*-bit Gray code *G*(*k*) is defined recursively.

  (1) *G*(1) is a sequence: 0 1.
  (2) *G*(*k*+1) is constructed from G(k) as follows.
  a. Construct a new sequence by appending a 0 to the left of all members of *G*(*k*).
  b. Construct a new sequence by reversing *G*(*k*) and then appending a 1 to the left of all members of the sequence.
  c. *G*(*k*+1) is the concatenation of the sequences defined in steps a and b.

- Examples:
  > Two-bit Gray code
  `00 01 11 10`
  > Three-bit Gray code
  `000 001 011 010 110 111 101 100`

EMBEDDING A LINE TOPOLOGY INTO A HYPERCUBE

Map the processor *i* of the line topology (size 2^d) onto the *i*-th entry of the *d*-dimensional hypercube.



- **`long gray(long j)`** returns the binary reflected Gray code for the input.

For example, `gray(4) = 6`.

```
j           000 001 010 011 100 101 110 111  or    0 1 2 3 4 5 6 7
gray(j)     000 001 011 010 110 111 110 100  or    0 1 3 2 6 7 5 4
```

`long ginv(long j)` returns the position of an element in the binary-reflected Gray code sequence. For example, `ginv(6) = 4`.

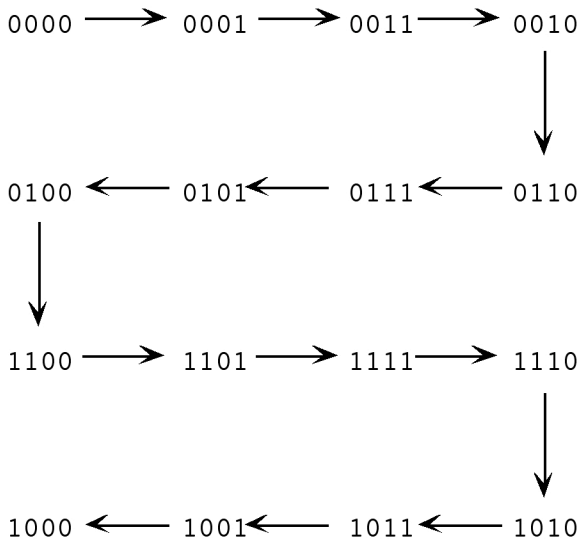
- **Ring** can be embedded using the Gray code, too. Note that the Hamming distance between the first and the last elements is 1.

EMBEDDING A LINE TOPOLOGY INTO A HYPERCUBE

Let us fold a 3-bit Gray code into two as follows:

```
000 → 001 → 011 → 010
                  ↓
100 ← 110 ← 111 ← 110
```

Note that there are also vertical links in the hypercube topology. All the pairs that are directly above and below each other have Hamming distance 1, and are connected in the underlying hypercube topology. This is because the Gray code is constructed by reversing the smaller Gray-code sequence and appending it to the original sequence. This property is used to embed a two-dimensional mesh onto a hypercube.

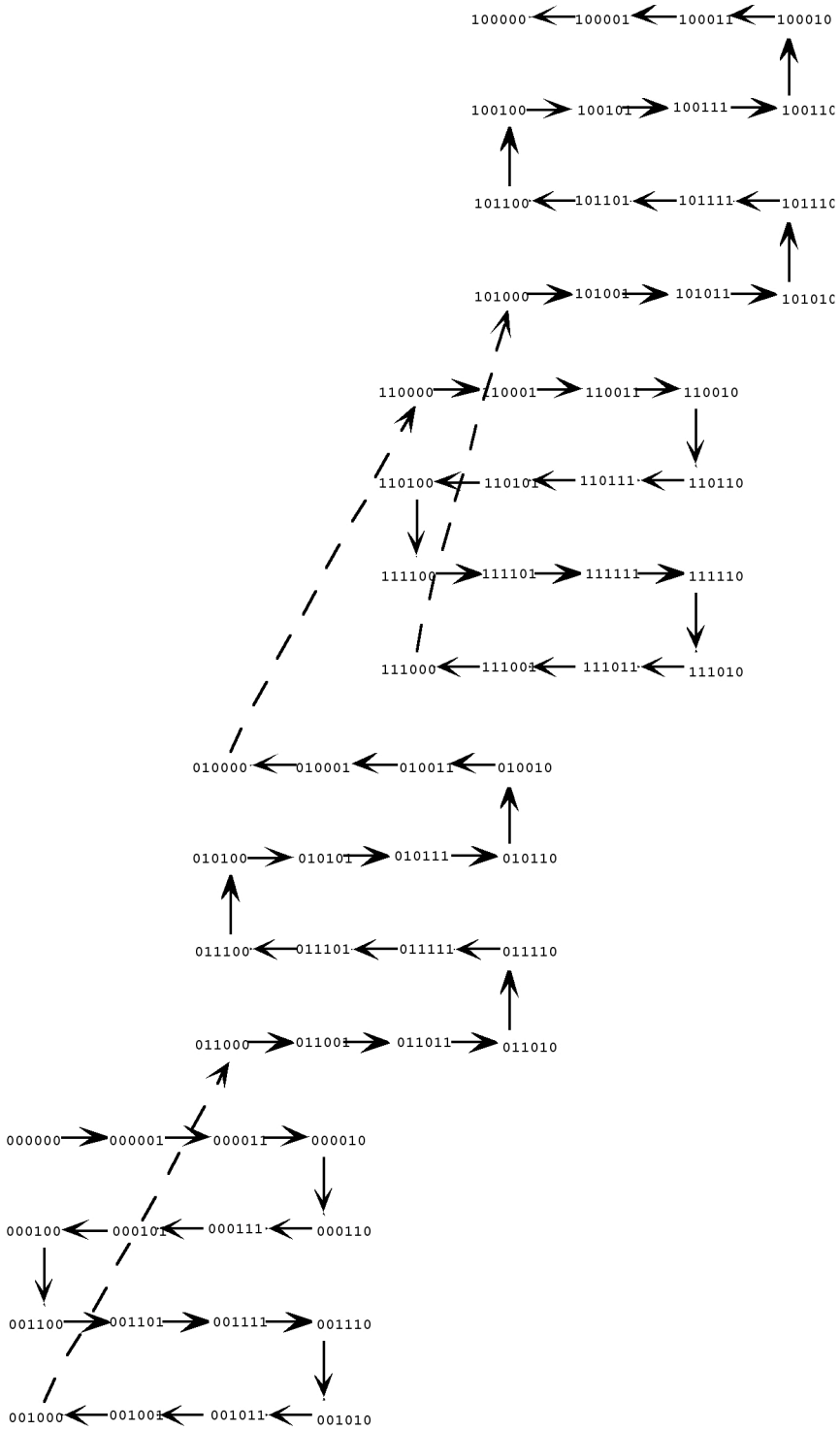


- **2D torus** can be embedded in a hypercube using the same Gray code.

EXAMPLE: EMBEDDING 4×4×4 IN 6-DIMENSIONAL HYPERCUBE

- In the example below, *G*(2) guarantees vertical connectivity; *G*(4) guarantees in-plane connectivity.

```
        G(2)|G(4)
a         00|xxxx
b         01|xxxx
¬b        11|xxxx
¬a        10|xxxx
```

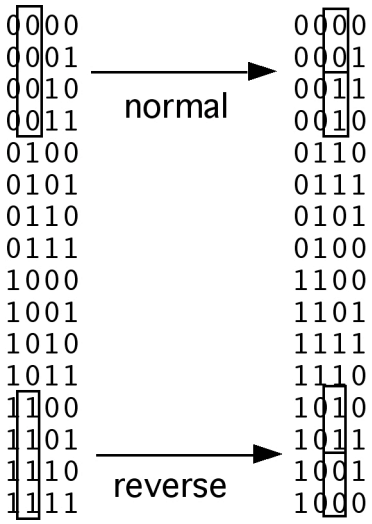


WRITE OUR OWN GRAY-CODE FUNCTION

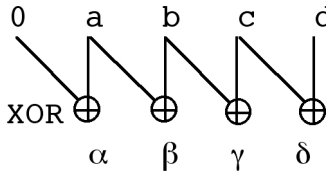
- Program: `gray.c`

```
int gray(int i) {
  return (i ^ (i/2));
}
```

Due to the recursive construction, a bit in a Gray code depends on whether it is in the first half or the second half of the sequence at the next level. Note that the order is inverted in the second half.



If the left bit is 0, then it is in the first half and that the bit must be conserved. Otherwise (the left bit is 1), the bit must be complemented). This is achieved by using the bitwise XOR operation. Comparison with the left bit is achieved by the right shift.



WRITE OUR OWN INVERSE-GRAY-CODE FUNCTION

- Program: `gray_inverse.c`

```
int gray_inverse(int i) {
  int answer,mask;
  answer = i;
  mask = answer / 2;
  while (mask > 0) {
    answer = answer ^ mask;
    mask = mask / 2;
  }
  return answer;
}
```

First note that if `a XOR x =` α then `x = a XOR` α.

```
a  x  α = a XOR x
0  0  0
0  1  1
1  0  1
1  1  0
```

Let's invert the mapping `abcd` $\rightarrow \alpha\beta\gamma\delta$

First α = `0 XOR a` and therefore `a = 0 XOR` α = α.
Next β = `a XOR b` and therefore `b = a XOR` β = α `XOR` β.
Similarly, `c =` α `XOR` β `XOR` γ, etc. This is achieved by repeated right shift and bitwise XOR.