6.1 Suzaku Generalized Pattern Concept

Message passing patterns connect sources and destinations together in various ways. Rather than implement every pattern in a unique way, in the Suzaku generalized pattern, a directed graph called here a \textit{connection graph} is defined that specifies the pattern. Then this graph is used subsequently with a generalized send routine that will send data to all processes connected according to the connection graph. Any connection pattern can be created this way. Of course one has to avoid messaging deadlock in the pattern implementation and it may be the implementation is not as efficient as specific implementations for specific patterns.

Most patterns are repeated as iterative synchronous patterns terminating when a termination condition occurs, usually either a fixed number of iterations or when the computed values converge sufficiently (i.e. do not change by more than a given value). The Suzaku generalized pattern is an iterative synchronous pattern with a master-slave structure as illustrated in Figure 1. The master sends initial data to all slaves and collects results from all slaves at the end of the computation. The slaves compute and send values to those slaves that are interconnected, repeatedly until the termination condition exists. The master also acts as one slave as in the master-slave pattern. For greatest flexibility, the programmer implements the iteration loop. Suzaku routines are provided so that the programmer to construct the pattern and to send data to

![Figure 1 Generalized iterative synchronous pattern](image-url)
connected processes. Broadcast/scatter/gather between the master and the slave processes rely on using existing low-level Suzaku routines.

6.2 Connection Graph

To be able to use the generalized pattern routines, one needs to appreciate the connection graph. The connection graph specifies which slaves are interconnected and the location in the destination for the incoming data. It is a directed graph so it would be possible for the connection to be in one direction although that would be unusual. The graph is a P x P adjacency matrix, \( \text{connection_graph}[P][P] \), where there are P processes (slaves, as the master acts as one slave) and illustrated below for eight slaves:

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The source data being transferred is a 1-D array of doubles, \( \text{output}[N] \). The destination array is a 2-D array, \( \text{input}[P][N] \). The graph entry at \( \text{connection_graph}[i][j] \) indicates:

-1  No connection
x  A connection process \( i \) to process \( j \), and the value, \( x \), indicates the row in the destination array where the data is to be placed, i.e. \( \text{input}[x][N] \).

This would allow a fully connected graph with each value received held in a separate location. In a partially connected graph, not all rows in \( \text{input}[][N] \) would be used. The graph can be set up to create any pattern including all-to-all, pipeline, stencil, binary tree etc. The all-to-all, pipeline, and stencil patterns have been created so far. Once the pattern is created, a generalized send routine uses the connection graph and send \( \text{output}[] \) to all connected processes, storing the data in the designated row of \( \text{input}[][N] \).

The basic version sends 1-D arrays as in the workpool version 1 as this is the most likely data structure and most efficient implementation although there is no technical reason why version 2 put and get mechanism could not be used. The arrays hold doubles.
Sample connection graph patterns

1. "all-to-all"

<table>
<thead>
<tr>
<th>Source process</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>-1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>-1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>-1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>-1</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>-1</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>-1</td>
</tr>
</tbody>
</table>

i.e. the array output[N] from slave $i$ will be sent to the $i$th row of input (input[$i$][N]) and all locations of input will be used.

2. "pipeline" (note: a ring)

<table>
<thead>
<tr>
<th>Source process</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

i.e. the array output[N] from slave $i$ will be sent to the first row of input (input[0][N]) and input[1][N] … input[N-1][N] will not be used.

3. 2-D "stencil"

Slaves are arranged in a square 2-D mesh. The number of slaves must have an integer squareroot. Nine slaves gives a 3 x 3 stencil. Processes are numbered in natural order:

0 1 2
3 4 5
6 7 8
Apart from slaves at the edges, each slave connects to the four neighbors on left, right, up and down, e.g. process 4 connect to 1, 3, 5 and 7. The edges only connect to those slaves that exist, e.g. process 1 connects to 0, 2, and 4. In most stencil computations, a constant boundary value is used by the process in the computation where it does not have neighboring process.

Processes will receive up to four output[] arrays, one from each neighbor loaded into input[0][]], input[1][]], input[2][]], and input[3][]]. Values for x:

- From the process to the left x = 0
- From the process to the right x = 1
- From the process above it x = 2
- From the process below it x = 3

to all each to be placed in different location.

Below is shown for a 3 x 3 stencil (9 x 9 connection graph):

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>Source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

### 6.3 Generalized Pattern Routines

(a) Setting up pattern

(i) For setting up a standard pattern:

```c
void SZ_Pattern_init(const char* pattern, int N)
```

*Purpose:* To initialize the connection graph, connection_graph[P][P] to one of various selectable patterns and create message buffer space for the generalized send routines. This routine provides a copy of the connection graph and message buffer space to all processes and is called within a parallel section before using the pattern with `SZ_Generalized_send()`.

*Parameters:*
**pattern** Name of the pattern as a string constant (input parameter). So far:
- "all-to-all"
- "pipeline" or “ring”
- "stencil"

**N** Number of data items, i.e. size of **output[]** (input parameter).

**Limitation:** The **connection_graph[P][P]** is statically declared as 20 x 20 elements, setting the maximum number of slaves (processes) to be 20. It is not expected that \( P \) would be very large in most applications, but it can be altered in **suzaku.c**. The actual size of \( P \) being used is established with **SZ_Init()**. The size \( N \) is not so limited as the message buffer is declared dynamically in **suzaku.c**. **SZ_Pattern_init()** must only be called within a parallel section.

(ii) For setting up a user-defined pattern:

```c
void SZ_Set_conn_graph(int *g)
```

**Purpose:** To set the **connection_graph[P][P]** to the values given by the user-supplied input array, \( g[][] \). **SZ_Pattern_init()** must be called first (with any standard pattern) to set \( N \) and create the message buffers. Then **SZ_Set_conn_graph()** will overwrite the connection graph.

**Parameter:**

* \( g \) Pointer to the array \( g[P][P] \) holding the pattern where \( P \) is the number of processes

**Limitation:** It is assumed the size of the provided array \( g \) is a \( P \times P \) integer array. Each process needs a copy of this array as in this routine, each process will set up its own connection graph locally.

(b) Generalized send routine

```c
void SZ_Generalized_send(double *output, double *input)
```

**Purpose:** To send the array **output[N]** to all connected processes as specified in the connection graph. The destination process stores the array in row of **input[P][N]** given by the connection graph.

**Parameters:**

* **output** Pointer to the array **output[N]** in source process
* **input** Pointer to the array **input[P][N]** in destination
**Limitation:** It is assumed that \( N \) is the value set in `SZ_Pattern_init()` and \( P \) is the value set in `SZ_Init()` for indexing into the array `output[][]`. This routine must only be called within a parallel section.

### 6.4 Overall program structure

The overall program structure is shown below:

```c
SZ_Parallel_begin     // parallel section, all processes do this
    SZ_Pattern_init("pattern_name",T,D,R);  // set up slave interconnections in each slave
                                                // add `SZ_Set_conn_graph()` if required
    ...
    SZ_Broadcast(input);                     // initialize data, input and output arrays
    for (i = 0; i < steps; i++) {
        compute(i,input,output);
        SZ_Pattern_send(output,input);         // slaves execute compute, master acts as a slave
                                                // sent compute results to connected slaves
    }
    SZ_Gather(input,result);                // collect results from slaves
SZ_Parallel_end;     // end of parallel
```

Note: Each slave must maintain two arrays `input[][]` and `output[]`. `Input[][]` holds the data sent from connected slaves. Slaves create results in `output[]` to be send to connected slaves. The connection graph specifies how the data is arranged in these arrays.

The broadcast corresponds to *diffuse* in the general case and could be coded inside a routine called `diffuse()`. Similarly, gather corresponds to *gather* in the general case and could be coded inside a routine called `gather()`.

### 6.5 Debugging

One routine currently available:

```c
void SZ_Print_connection_graph(void)
```

*Purpose:* To cause the master to print the current connection graph, `connection_graph[P][P]`, for test purposes.

*Parameters:* None

### 6.6 Implementation

Messaging is done point to point and a barrier in present at the end to ensure all processes complete before returning, i.e. the routine is synchronous as are low-level Suzaku message passing routines but it is implemented as a routine and not as a macro. If all `MPI_send()`'s
precede `MPI_recv()`’s in a process, there is a possible deadlock if sends become synchronous because of lack of buffer storage. To avoid possible deadlock, the implementation uses MPI buffered sends with explicit buffer space. Beforehand calling `MPI_BSend()` for the first time in a process, it is necessary to call `MPI_Buffer_attach()` to attach a buffer. The size of the buffer needs to be only big enough for all pending sends in a process. Here each process just needs space for one message. At end of all sends `MPI_Buffer_detach()` should be called. `SZ_Pattern_finalize()` will do this if the programmer wants to use it.

6.7 Compilation and Execution

The generalized pattern routines are found in `suzaku.c`. Application code using them must be compiled with the math libraries, `-lm` option even if `suzuku.o` is recompiled.
Sample programs

1. all-to-all pattern, pattern_test.c

This program simply tests the all-to-all pattern.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "suzaku.h"  // Basic Suzaku macros

#define D 2  // # of data items in slave data.
#define P 4  // Number of processes -- this code must be run only with this number of processes

void compute(int taskID, double B[P][D], double A[D]) {  // each slave
    printf("Slave %d step %d A[0]=%5.2f, B[0][0]=%5.2f, B[1][0]=%5.2f, B[2][0]=%5.2f, B[3][0]=%5.2fn",
        SZ_Get_process_num(), taskID, A[0], B[0][0], B[1][0], B[2][0], B[3][0]);
    return;
}

int main(int argc, char *argv[]) {
    int i,j,p;    // p is actual number of processes when executing program
    double A[D],B[P][D];   // A is the slave data, B holds data sent from other slaves
    int steps = 2;    // number of time steps

    SZ_Init(p);    // initialize MPI message-passing environment

    if (p != P) printf("ERROR Program must be run with %d processes\n",P);

    for (i = 0; i < D; i++) {   // all processes
        A[i] = SZ_Get_process_num();
        for (j = 0; j < P; j++){
            B[i][j] = 0;
        }
    }

    SZ_Pattern_init("all-to-all",D);  // set up slave interconnections
    SZ_Print_connection_graph();  // for checking

    for (i = 0; i < steps; i++) {
        compute(i, B, A);  // slaves execute compute, master acts as one slave
        SZ_Generalized_send(A, B); // sent compute results to connected slaves
    }

    SZ_Gather(A,A);  // collect results from slaves, gather()
    return 0;
}
```
Notice compute simply prints out the input and output arrays. The first iteration, there are at their initialized values. The second iteration shows them updated after the messaging done by `SZ_Generalized_send(A, B)`.
2. Sorting using a generalized pipeline pattern – gen_pipeline_sort.c

The basic pipeline is shown below described in terms of diffuse, compute and gather:

Here, the master does not act as one slave. It generates numbers and receives the final results.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "suzaku.h" // Basic Suzaku macros
#define D 1 // # of data items in slave data.
#define P 4 // Number of processes -- must be run only with this number of processes,
int main(int argc, char *argv[]) {
    int i, j, p,pid;   // p is actual number of processes when executing program
    int T = 3 * P;   // number of time steps
    double A[D];   // data to send (D = 1).
    double B[P][D];   // received data, from each source
    static double largest = 0;
    for (i = 0; i < D; i++) A[i] = 0; // initialize to zero
    for (i = 0; i < P; i++)   // initialize receive so can see what received
        for (j = 0; j < D; j++)
            B[i][j] = -999;
    srand(1);    // initialize rand()
    SZ_Init(p);    // initialize MPI message-passing environment
    if (p != P)     // number of processes hardcoded
        printf("ERROR number of processes must be %d\n",P);
    SZ_Parallel_begin   // parallel section, all processes do this
    SZ_Pattern_init("pipeline",D);  // set up slave interconnections
    SZ_Print_connection_graph();  // for checking
    for (i = 0; i < T; i++) {
        pid = SZ_Get_process_num();   // identify process
        if (pid ==0) {  // master generates next number to sort, ends with a terminator
            if (i < P) A[0] = rand()% 100;  // P numbers, a number between 0 and 99
            ...  // master computes
        }
    }
}
```
else A[0] = 999;  // otherwise terminator
printf("Master sends %3.0f and receives %3.0f\n", A[0], B[0][0]);
} else {
   // slaves execute compute, using B to create A.
   if (B[0][0] > largest) {
      A[0] = largest;  // copy current largest into send array
      largest = B[0][0];  // replace largest with received number
   } else {
      A[0] = B[0][0];  // copy received number into send array
   }
}

SZ_Generalized_send(A, B);  // sent results, includes master to slave, slave to master
SZ_Barrier();  // wait for every process to complete

SZ_Parallel_end;  // end of parallel
SZ_Finalize();

return 0;
}

Notice A and B are static arrays to match the generalized send routine. The program could have been written with specific diffuse, compute and gather routines.

Sample output:

```
Master sends 83 and receives 999
Master sends 86 and receives 999
Master sends 77 and receives 999
Master sends 15 and receives 999
Master sends 999 and receives 0
Master sends 999 and receives 0
Master sends 999 and receives 0
Master sends 999 and receives 15
Master sends 999 and receives 77
Master sends 999 and receives 83
Master sends 999 and receives 86
Master sends 999 and receives 999
```
3. Stencil pattern – **gen_heat.c**

The following solve the two-dimensional heat distribution problem. For simplicity, only 16 points are used and one of 16 processes for each point. The approach can be be extended to have each process handle multiple points. This is left as an exercise.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "suzaku.h"  // Basic Suzaku macros
#define D 1 // # of data items in slave data
#define P 16 // Number of processes -- this code must be run only with this number of processes
#define N 6  // Number of pts in each dimension, to include border 6 x 6
#define M 4 // Number of pts in each dimension, not including border 4 x 4

int main(int argc, char *argv[]) {
    int i,j,x,y,t;    // loop counters
    int T = 100;    // time period
    int p, pid;
    double pts[N][N];   // array of points to include fixed borders
    double A[1];    // point being computed in slave, output array
    double B[P][D];    // input array
    double temp[M][M];   // hardcoded for 4 x 4
    double pts_seq[2][N][N];  // array to do computation sequentially.
    int current = 0;
    int next = 1;

    SZ_Init(p);    // initialize MPI message-passing environment
    if (p != P) printf("ERROR Program must be run with %d processes\n",P);
    printf("Number of points in each dimension = %d\n",N);
    printf("Number of time steps = %d\n",T);

    /* -------------------------- Set up initial values ---------------*/
    for(i = 0; i < N; i++)   // load initial values into array
        for(j = 0; j < N; j++) { // border and inner points = 20
            pts[i][j] = 20;  // note C row major order, row i, col j
            if (j == 0) printf("\n");
            printf("%7.2f",pts[i][j]);
        }
    printf("\n");

    for(i = 0; i < N; i++)   // load initial values into array
        for(j = 0; j < N; j++) { // border and inner points = 20
            pts_seq[current][i][j] = pts[i][j];
            pts_seq[next][i][j] = pts[i][j];
        }

    for (t=0; t < T; t++) {    // do computation sequentially, using Jacobi iteration
        for(i=0; i < N-1; i++)
            for(j=0; j < N-1; j++) {
                pts_seq[current][i][j] = 0.25 * (pts_seq[current][i-1][j] + pts_seq[current][i+1][j] + pts_seq[current][i][j-1] + pts_seq[current][i][j+1]);
                current = next;
                next = 1 - current;
            }
        printf("\n");
    }
}
```
/* -------------------------Computation-----------------------------------*/

SZ_Parallel_begin       // parallel section, all processes do this

SZ_Pattern_init("stencil",D);      // set up slave interconnections

SZ_Broadcast(pts);          // synchronous, includes a barrier
                                      // Set up initial values in each slave

pid = SZ_Get_process_num();

x = pid / M;        // row, hardcoded for 16 processes 4 x 4
y = pid % M;        // column

i = x + 1;          // location in pts[i][j]

j = y + 1;

A[0] = pts[i][j];    // copy location in pts[i][j] into A[0]

B[0][0] = pts[i][j-1];   // left

B[1][0] = pts[i][j+1];  // right

B[2][0] = pts[i-1][j];  // up

B[3][0] = pts[i+1][j];  // down

for (t = 0; t < T; t++) {         // compute values over time T

A[0] = 0.25 * (B[0][0] + B[1][0] + B[2][0] + B[3][0]); // slaves execute computation,
\master acts as one slave

SZ_Generalized_send(A,B);       // sent compute results in A to B in connected slaves

}

SZ_Gather(A,temp);       // collect results from slaves (A), into array temp, gather()

/* ------------------------- Results -----------------------------------*/

for (x = 0; x < N; x++)   {  // update inside points

for(y = 0; y < N; y++) {

if ((x > 0) && (x < N-1) && (y > 0) && (y < N-1)) { // inside point

i = x + 1;

j = y - 1;

pts[x][y] = temp[i][j];

}

}

printf("Final numbers"); // print numbers

for (i = 0; i < N; i++) {

for(j = 0; j < N; j++) {

if (j == 0) printf("\n");

printf("%.2f",pts[i][j]);

}

printf("\n");

int error = 0; // check sequential and parallel versions give same answers

for (i = 0; i < N; i++) {

for(j = 0; j < N; j++) {

if ( ((pts[i][j] - pts_seq[current][i][j]) > 0.001) || (pts_seq[current][i][j] - pts[i][j] > 0.001) )

{ error = -1; break; }

}

if (error == -1) break;
}

if (error == -1) printf("ERROR, sequential and parallel versions give different answers\n");

else printf("Sequential and parallel versions give same answers within +0.001\n");

SZ_Finalize();

return 0;

}
Sample output:

```
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$ make gen_heat
mpicc -o gen_heat gen_heat.c suzaku.o -lm
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$ mpiexec -n 16 gen_heat

Number of points in each dimension = 6
Number of time steps = 100

Initial numbers
20.00 20.00 100.00 100.00 20.00 20.00
20.00 20.00 20.00 20.00 20.00 20.00
20.00 20.00 20.00 20.00 20.00 20.00
20.00 20.00 20.00 20.00 20.00 20.00
20.00 20.00 20.00 20.00 20.00 20.00
20.00 20.00 20.00 20.00 20.00 20.00

Final numbers
20.00 20.00 100.00 100.00 20.00 20.00
20.00 31.21 56.36 56.36 31.21 20.00
20.00 28.48 37.88 37.88 28.48 20.00
20.00 24.85 28.79 28.79 24.85 20.00
20.00 22.12 23.64 23.64 22.12 20.00
20.00 20.00 20.00 20.00 20.00 20.00

Sequential and parallel versions give same answers within +/- 0.001
```

```
4 Printout of patterns – gen_connect_test.c

The following prints out the three standard patterns implemented and one user-defined pattern set with `SZ_Set_conn_graph()`.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "suzaku.h"  // Basic Suzaku macros
//Declared as constants to allow static arrays for input and output
#define D 2  // # of data items in slave data.

int main(int argc, char *argv[]) {
    int p,i,j;    // p is actual number of processes when executing program
    SZ_Init(p);     // initialize MPI message-passing environment
    SZ_Parallel_begin    // parallel section, all processes do this
        int g[p][p];
        SZ_PATTERN_INIT("all-to-all",D); // set up slave interconnections
        SZ_Master { printf("all-to-all pattern\n"); }
        SZ_Print_conn_graph();   // for checking
        SZ_PATTERN_INIT("pipeline",D); // set up slave interconnections
        SZ_Master { printf("pipeline pattern\n"); }
        SZ_Print_conn_graph();   // for checking
        SZ_PATTERN_INIT("stencil",D); // set up slave interconnections
        SZ_Master { printf("stencil\n"); }
        SZ_Print_conn_graph();   // for checking
        for (i = 0; i < p; i++)   // set user-defined pattern
            for (j = 0; j < p; j++)
                g[i][j] = i+j;
        SZ_Set_conn_graph(g);
        SZ_Master { printf("User-defined pattern\n"); }
        SZ_Print_conn_graph();   // for checking
    SZ_Parallel_end;    // end of parallel
    SZ_Finalize();
    return 0;
}
```

Sample output:
Terminal

`abw@abw-VirtualBox:/ParallelProg/Suzaku`

```
abw@abw-VirtualBox:/ParallelProg/Suzaku$ mplexec -n 4 gen_connect_test
all-to all pattern
Connection graph
  0 1 2 3
  0 -1 0 0
  1 1 -1 1
  2 2 2 -1
  3 3 3 3
pipeline pattern
Connection graph
  0 1 2 3
  0 -1 0 -1
  1 -1 0 -1
  2 -1 -1 0
  3 0 -1 -1
stencil
Connection graph
  0 1 2 3
  0 -1 0 2
  1 1 -1 1
  2 3 -1 1
  3 -1 3 1
User-defined pattern
Connection graph
  0 1 2 3
  0 0 1 2
  1 1 2 3
  2 2 3 4
  3 3 4 5
abw@abw-VirtualBox:/ParallelProg/Suzaku$
```