Optimising Communication Routines in Parallel X-Agents

CCLRC Software Engineering Group
L.S. Chin, C. Greenough, and D.J. Worth

January 16, 2007

Abstract:

This report describes the work done in analysing the communication routines within parallel x-agent models used for modelling biological systems. Based on the analysis, several changes are proposed to overcome possible deadlocks due to communication synchronisation, as well as to optimise performance and memory utilisation within the communication routines.

Keywords: X-Agents, Intelligent Agents, MPI, Message Passing

1 Background

Researchers at the University of Sheffield have developed a framework for building x-agent models of biological systems based on the X-Machine Agent Markup Language (XMML) [8, 9]. XMML files define the types of agents, their memory data and functions involved in a model. This file is parsed by the x-parser to create a C program for simulating the system.

The generated C program can be either a serial or a parallel program, with the parallel version using MPI [4] to pass messages between agents on different nodes.

A more detailed introduction to the topics of x-agents and computational systems biology is provided by [7], while notes on previous analysis and optimisation of the x-agent model is available in [1].

2 Overview of communication routines

MPI Communication routines appear in seven routines, and are used to perform three distinct operations:

1. **Propagation of messages**
   - `propagate_messages()` – uses `MPI_Recv`, calls:
     - `sent_message_pack()` – uses `MPI_Send`
     - `receive_message_pack()` – uses `MPI_Recv`
2. **Propagation of agents**
   - `propagate_agents()` – uses `MPI_Recv`, calls:
     - `sent_agent_pack()` – uses `MPI_Send`
     - `receive_agent_pack()` – uses `MPI_Recv`
3. **Distribution of partitioning information**
   - `send_space_partitions()` – uses `MPI_Bcast()`

Propagation of agents and messages are both performed using similar methods, while the distribution of partitioning information involves only a simple broadcast call. As such, optimisation activities were focused on the propagation of messages, the results of which would be equally applicable to the propagation of agents.

3 Dissecting `propagate_messages()`

Figure 1 illustrates the steps involved in `propagate_messages()`.

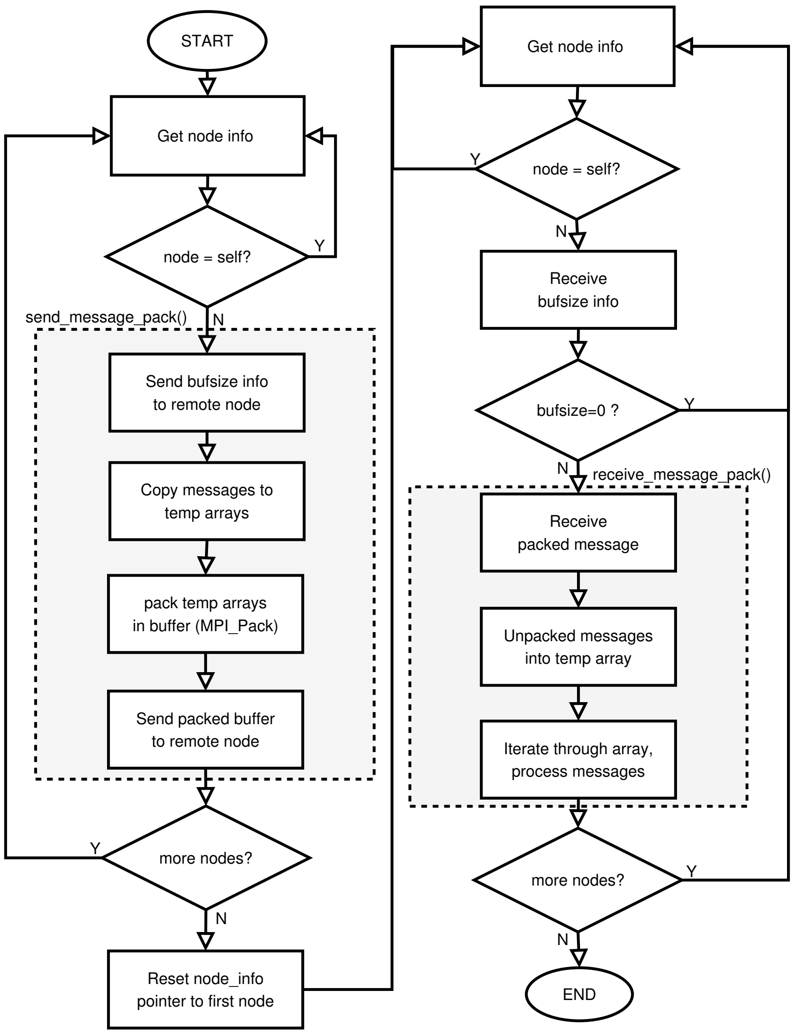

Figure 1: Flowchart for the `propagate_messages()` routine

The `propagate_messages()` routine, which calls `send_message_pack()` and `receive_message_pack()`, traverses the the list of nodes[1] and sends out the number of pending messages for each node. This is done by issuing an initial `MPI_Send` with information on the size of the pending messages. The actual messages are then packed into a buffer and sent using another `MPI_Send` call.

Once all messages have been sent out, the routine then traverses the list a second time to receive messages sent in by other nodes. For each remote node, it first receives information on the size of the incoming messages, allocates the appropriate buffers, and then issues matching `MPI_Recv` calls to complete the communication.

[1]The term *nodes* is used in this text to represent instances of the model running in separate processes. In addition, we use *local node* to refer to the *node* performing the actions in question, and *remote node* to refer to the rest of the *nodes*

3.1 Possible deadlock

Note: Some of the terminology used in the following sections to describe MPI communication are defined in Appendix A (page 11)

In the current code, a series of `MPI_Send` calls are issued in advance followed by the `MPI_Recv` calls. Since `MPI_Send` is a blocking routine, the `MPI_Recv` calls will not be issued unless all the `MPI_Send` calls are completed. As such, the viability of this method essentially relies on `MPI_Send` being used in an asynchronous manner, whereby `MPI_Send` must complete even when a matching receive has not yet been issued on the remote node.

However, since asynchronous communication depends on message buffering which in turn is used at the discretion of the *MPI* implementation, it is unsafe to assume that communication will always be asynchronous. If the message being sent is larger than a predefined threshold, or when too many asynchronous calls deplete the system buffer, `MPI_Send` would resort to a synchronous communication mode. This results in a deadlock where every node would be waiting for the others to issue an `MPI_Recv`.

An example code which illustrates this behaviour is available in Appendix D (page 24). In this sample application, messages are repeatedly exchanged using `MPI_Send` followed by `MPI_Recv`. In each iteration, the message size gets bigger up to a point where `MPI_Send` becomes asynchronous and the application deadlocks.

3.2 Data duplication

An examination of the propagation routines reveal that data is duplicated up to four times during message staging.

Within each node, messages and agent memory are stored in *C structures* chained together as linked-lists. This means that data intended for propagation (in `propagate_messages()` or `propagate_agents()`) is fragmented and non-contiguous in memory. Since `MPI_Send` requires the outgoing data to be contiguous in memory, the packing of messages makes it possible to send a whole list of messages of different datatypes via the same call.

```
i = 0;
temp_message_location = node_info->location_messages;
while(temp_message_location)
{
        location_message_list[i].id = temp_message_location->id;
        location_message_list[i].x  = temp_message_location->x;
        location_message_list[i].y  = temp_message_location->y;
        location_message_list[i].z  = temp_message_location->z;
        /* --- <snip> ---- */

        i++;
        temp_message_location = temp_message_location->next;
}

MPI_Pack(location_message_list, node_info->location_message_no, \
        messagelocationType, buf, buffersize, &position, MPI_COMM_WORLD);

MPI_Send(buf, position, MPI_PACKED, node_info->node_id, 19, MPI_COMM_WORLD);
```

As seen above in the code snippet taken from the current implementation, data from the linked-list referenced to by `temp_message_location` is first copied to the `location_message_list` array. This ensures that data is contiguous in memory and ready for packing.

Data from `location_message_list` is then packed into `buf`, and eventually buffered by the `MPI_Send` call into the system buffer[2].

[2] If system buffer is unavailable, the `MPI_Send` call would block leading to a possible deadlock (as described in section 3.1)

Figure 2 depicts the data duplication that would occur when multiple message types are packed for sending.

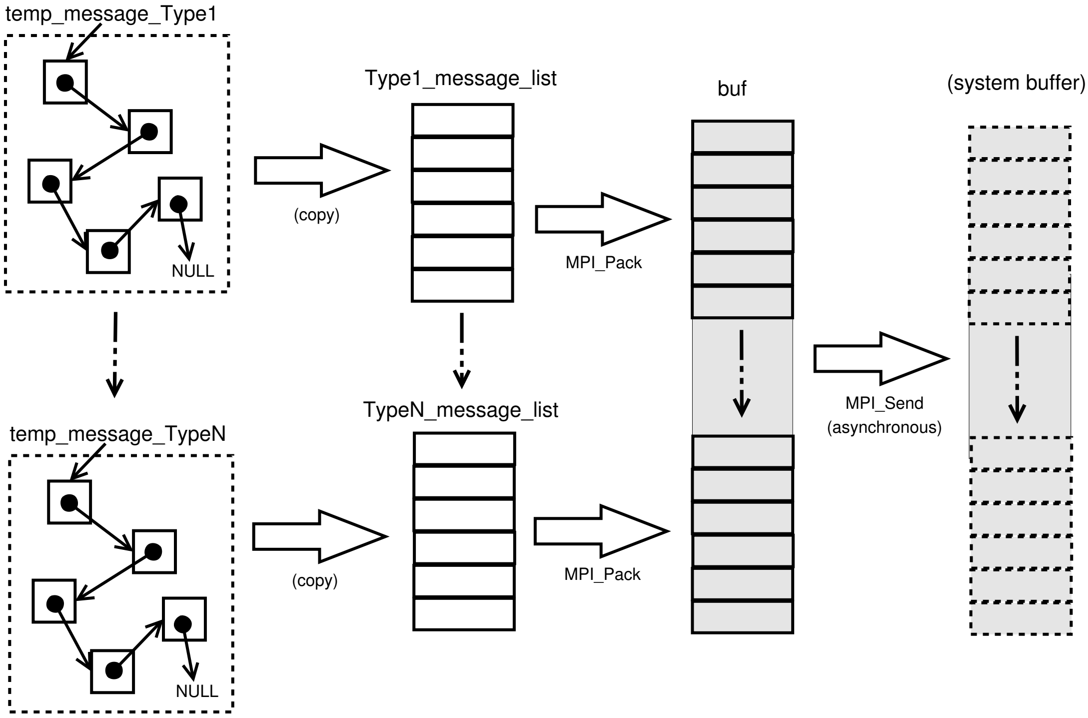

Figure 2: Memory utilisation of `send_message_pack()` for multiple messages types

Similar steps are performed (in reverse order) in `receive_message_pack()`. Data is unpacked from the receive buffer into a temporary array, which is then duplicated and attached to an appropriate linked-list.

When large amounts of messages are involved, this repeated duplication of memory is likely to have an impact on performance.

4 Proposed optimisation

This chapter presents several optimisation techniques than may overcome issues discussed in previous sections.

A sample implementation which incorporates these ideas are included in Appendix B and Appendix C. This code has not yet been thoroughly tested or benchmarked, and as such does not represent an end-product, but rather a reference point for further analysis and optimisation.

4.1 Using non-blocking communication

Non-blocking routines should be used for communication in order to obtain the following benefits:

- **Reduce synchronisation overheads**: By decoupling the completion of each send from the receipt, overhead imposed by the synchronisation between communicating nodes[3] can be reduced.
- **Avoid deadlocks**: Non-blocking routine calls return immediately even if the message cannot yet be sent. This prevents the deadlock condition described in section 3.1 from occurring.

[3]Asynchronous communication can also be achieved using buffer-oriented sends, e.g. `MPI_Bsend`, and under certain circumstances, `MPI_Send`

- **Latency hiding**: After a non-blocking routine call returns, actual communication would continue in the background while the calling process can proceed with other tasks, returning at a later point to confirm that communication has completed successfully. This technique, often known as latency hiding, masks the overhead of waiting for communication completion by overlapping the idle time with useful computation.

In the sample implementation, `propagate_messages()` is split up into two routines – `propagate_messages_init()` which packs outgoing messages and issues the non-blocking communication calls, and `propagate_messages_complete()` which processes the input messages as well as confirms the completion of all communication.

Splitting the routine as so would simplify the overlapping of communication with other routines. Routines that can be scheduled during communication would include file I/O routines, run-time analysis routines for determining load balancing strategies, or even stages of the *X-machine* that are independent of the messages being communicated.

```
/* initiate propagation of messages */
propagate_messages_init();

/* write iteration data to file */
if (iteration_loop%output_frequency == 0)
{
        saveiterationdata(iteration_loop);
}

/* other routines that do not depend on the messages */
analyse_load_balance(); /* fictional routine */

/* complete the propagation of messages */
propagate_messages_complete();
```

One drawback of using non-blocking communication is the increased memory requirements. Between the time when communication is initiated and when it is completed, application buffers that are allocated for every send and receive has to be preserved.

4.2 Message buffering strategies

To reduce the overhead incurred during data packing (covered in section 3.2), the following strategies were used:

- Packing the data *by hand* rather than using `MPI_Pack` – this involves allocating a single buffer that acts as both the temporary array as well as the send buffer.
  Addresses of strategic points in the buffer are casted to the different message types, which can then be used as the temporary array for copying data from the linked-list.
- Use of non-blocking *synchronous MPI* sends to prevent system buffering.
- Freeing up memory used by the linked-list as soon as possible.

Figure 3 illustrates the memory usage of this buffering strategy when applied to `propagate_messages()`. Similar techniques can be applied to the packing of different types of agents in `propagate_agents()`.

The following code snippet is an example of how the different message arrays can be assigned to the appropriate portions in `sendbuf`.

```
/* declaration of variables used in the following code snippet */
int bufsize;
void * sendbuf;
struct_message_type1 * msg_type1_array;
```

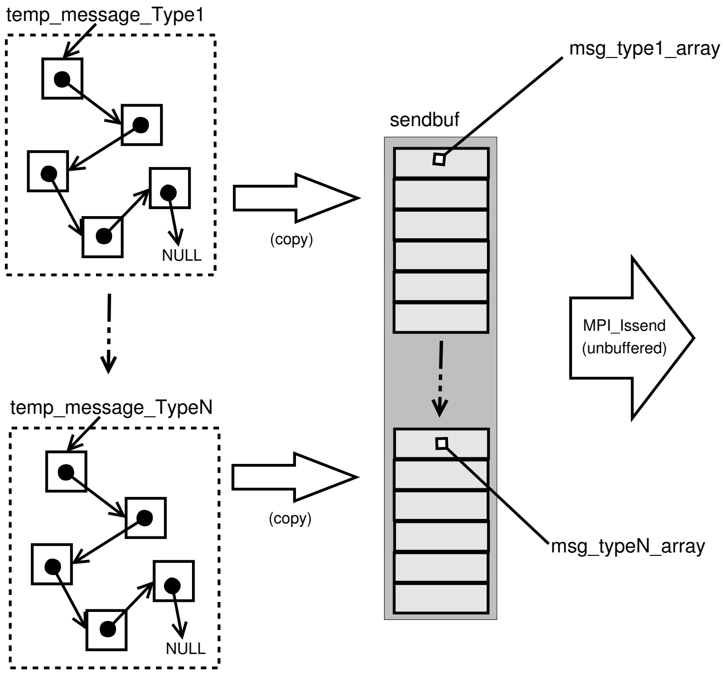

Figure 3: Memory usage of proposed optimisation for multiple message types

```
struct_message_type2 * msg_type2_array;
/* ... for subsequent message types ....
struct_message_typeN * msg_typeN_array;
*/
int msg_type1_count, msg_type1_count /* ..., msg_typeN_count*/;

/* ---- <snip> ----*/

/* Determine required buffer size */
bufsize = 0;
bufsize += sizeof(struct_message_type1) * msg_type1_count;
bufsize += sizeof(struct_message_type2) * msg_type2_count;
/* .... for subsequent message types ....
bufsize += sizeof(struct message_typeN) * msg_typeN_count;
*/

/* allocate memory for send buffer */
sendbuf = (void *) malloc(bufsize);

/* assign message array pointers */
msg_type1_array = (struct_message_type1 *) sendbuf;
msg_type2_array = (struct_message_type2 *) \
        &msg_type1_array[msg_type1_count];
/* .... for subsequent message types ....
msg_typeN_array = (struct_message_typeN *) \
        &msg_type(N-1)_array[msg_typ(N-1)_count];
*/

/* ---- < copy data from linked list into arrays > ---- */
/* ---- < free memory of linked list >               ---- */
/* ---- < send data in 'sendbuf' to recipient >     ---- */

free(sendbuf);
```

4.3 Reducing communication cost

In the current implementation, every node would issue an MPI_Send to all other nodes with information on how much data to expect in the actual propagation of messages. This means that at each propagation step, $N(N-1)$ sets of communication has to be established followed by $\sum M_{ij}$ more for sending the actual messages (where N is the number of nodes, and $\sum M_{ij}$ the sum of all outgoing messages for every node). This would be a huge performance bottleneck, especially when the simulation is scaled up to large numbers of nodes.

Nearest Neighbour communication

This overhead can be greatly reduced if the communication pattern can be predetermined. For example, if the simulation domain is spatially decomposed and communication is limited to only the nearest neighbours, communication cost can be reduced from that of $O(n^2)$ to $O(n)$.

Each node would have to keep track of nodes that are adjacent to it, and limit its communication to only these nodes. Neighbouring nodes can be determined manually during the partitioning stage, or obtained automatically if the *Virtual Topology* mechanism in *MPI* were used (see Chapter 7 of [6]).

While we believe that the biological model being analysed can greatly benefit from this optimisation, the required changes has not been incorporated into the sample code provided (Appendix B and Appendix C) as it would involve modification of other routines which are not directly related to communication.

For other simulations where the communication pattern is unknown and the above-mentioned optimisation not applicable, we have considered several other strategies for making the propagation more efficient.

Strategy A: Using MPI_Alltoall

The first strategy is to propagate information on message sizes using MPI_Alltoall. This would allow us to replace the $(N-1)$ MPI_Sends and $(N-1)$ MPI_Recvs on each node with a single collective operation. Collective routines are generally more efficient compared to functionally equivalent sets of point-to-point routines as the communication algorithms used are often optimised based on message sizes and characteristics of the hardware.

A possible implementation of this strategy might involve the following code:

```
/* Constants generated by parser depending on number of message types */
#define MESSAGE_TYPE1 0
#define MESSAGE_TYPE2 1
#define MESSAGE_TYPE_COUNT 2
/* --- <snip> --- */

/* for in/out message count table, we need 2d arrays that are
   contiguous in memory (for MPI_Alltoall). First dim needs to be
   dynamically sized based on node count, while second dim is
   pre-defined by the model and can be static.
*/
int (*in_mcount)[MESSAGE_TYPE_COUNT];
int (*out_mcount)[MESSAGE_TYPE_COUNT];

/* Allocate required memory for in/out message count table */
in_mcount  = (int (*)[MESSAGE_TYPE_COUNT]) malloc\
        (sizeof(int) * totalnodes * MESSAGE_TYPE_COUNT);
out_mcount = (int (*)[MESSAGE_TYPE_COUNT]) malloc\
        (sizeof(int) * totalnodes * MESSAGE_TYPE_COUNT);

/* How many messages do we have to send for each node? */
node_info = *p_node_info;
while(node_info)
{
```

```
        out_mcount[node_info->node_id][MESSAGE_TYPE1] = \
                node_info->location_type1_no;
        out_mcount[node_info->node_id][MESSAGE_TYPE2] = \
                node_info->location_type2_no;
        node_info = node_info->next;
}

/* Propagate list */
MPI_Alltoall(out_mcount, MESSAGE_TYPE_COUNT, MPI_INT, in_mcount,\
        MESSAGE_TYPE_COUNT, MPI_INT, MPI_COMM_WORLD);

/* we would now be able to obtain expected message count from
   each node via  in_mcount[ node_id ][ MESSAGE_TYPE? ]
*/

/* ---- <snip> ---- */

free(in_count);
free(out_count);
```

Strategy B: Using MPI_Probe

Another possible strategy is to pack message count information and actual message data within the same communication. This would remove the need for establishing a second connection, and thus reduce the total communication count from $[N(N-1)+\sum M_{ij}]$ to only $[N(N-1)]$ (where N is the number of nodes, and $\sum M_{ij}$ the sum of all outgoing messages for every node).

Since $\sum M_{ij} \leq N(N-1)$, this strategy could potentially reduce the number of required communication by up to 50%.

On each node, messages for each remote node are packed into the send buffers along with a counter of messages for each message type. If there are no messages for a particular remote node, an empty (zero bytes) data is sent.

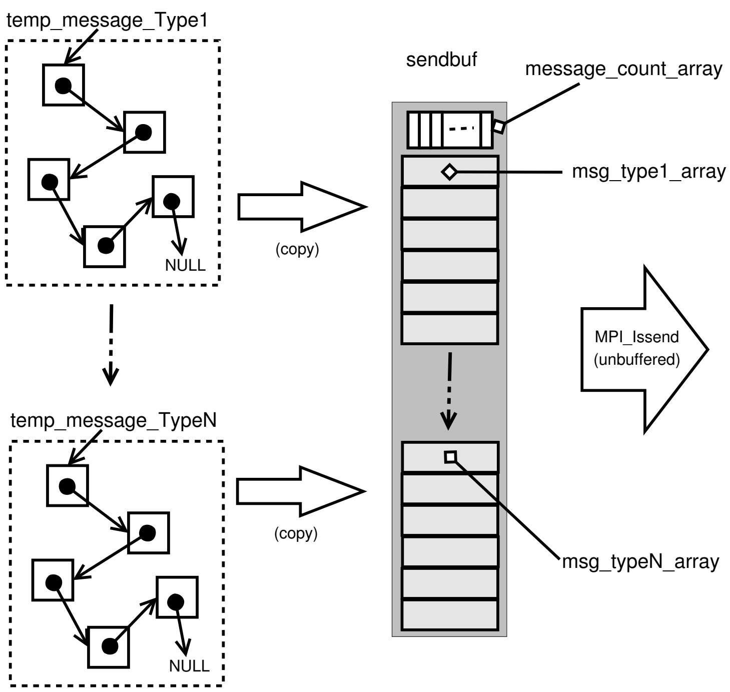

Figure 4: Counter of each message type is included at the beginning of the buffer. This assists the recipient in unpacking the rest of the buffer.

After issuing all non-blocking sends, each node then proceeds to receive data from every remote node. `MPI_Probe` is used to determine the size of data to expect from each communication. If the input data size is non-zero, an appropriate amount of memory is allocated for the receive buffer. All communication can then completed using non-blocking receives.

Based on out preliminary benchmarks[4], this strategy led to slightly better performance compared to *Strategy A*, and has therefore been implemented in our sample implementation (Appendix B and Appendix C).

[4]performed on SCARF (`http://hpcsg.esc.rl.ac.uk/scarf/index.html`), a 128 node dual Opteron cluster with 8GB memory per node, using *MPICH-GM* (MPI over Myrinet).

References

[1] D.J. Worth and C. Greenough, *"Optimising X-Agent Models in Computational Biology"*, Software Engineering Group Note SEG-N-001 (2006).
`http://www.cse.clrc.ac.uk/seg/html/wrap_pubs.shtml`

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *"MPI – The Complete Reference"*
`http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`

[3] *"MPI Performance Topics"*
`http://www.llnl.gov/computing/tutorials/mpi_performance/`

[4] *"The Message Passing Interface (MPI) standard"*
`http://www-unix.mcs.anl.gov/mpi/`

[5] Supercomputing Institute, University of Minnesota, *"Point to Point Communications with MPI"*
`http://www.msi.umn.edu/tutorial/scicomp/general/MPI/content1.html`

[6] N. MacDonald, E. Minty, T. Harding, and S. Brown, *"Writing Message-Passing Parallel Programs with MPI"*
`http://www2.epcc.ed.ac.uk/epic/mpi/notes/`

[7] *"Agent-based modelling – a tool for replicating biological systems"*
`http://www.dcs.shef.ac.uk/~rod/Integrative_Systems_Biology.html`

[8] *"X-Machine Agents"*
`http://www.dcs.shef.ac.uk/~stc/x-agents/index.shtml`

[9] *"FLAME – The Flexible Agent Modelling Environment"*
`http://www.flame.ac.uk`

Appendix A MPI Terminology

Application Buffer :
Address space allocated by user program and used to store data that is to be sent or received.

System Buffer :
System address space used for buffering messages. This buffer is not visible to the programmer, and accessible only be the *MPI* library. The primary purpose of system buffer space is to enable asynchronous communications.

Asynchronous Communication :
In an asynchronous communication, send operations may complete even when the receiving process has not received the message. Messages are copied to the system buffer, and scheduled for transmission when a matching receive is issued.
After the completion of an asynchronous send, the application send buffer can be safely overwritten, but there is no guarantee that the receiving process has received the message.
`MPI_Send` can be used to attempt an asynchronous send. However, depending on message sizes and the availability of system buffer, `MPI_Send` may fall back to a synchronous mode.

Synchronous Communication :
In a synchronous communication, send operations will only be completed when the message has been completely transferred to the receiving process.
After the completion of a synchronous send, it is guaranteed that the application send buffer can be safely overwritten, and that the receiving process has a copy of the message.
`MPI_Ssend` (note the extra 's' which stands for *synchronous*) can be used to issue a synchronous send.

Blocking :
A communication routine is blocking if its call only returns when the communication is completed. For example, `MPI_Send` will only return when the message has been moved to the system buffer or when the receiving process has received the message, while `MPI_Recv` will only return when incoming data is safely stored in the application buffer so that it is ready for use.

Non-Blocking :
A communication routine is non-blocking if its call returns without waiting for the communication to complete. It is not safe to modify or use the application buffer after a non-blocking call returns, and it is the programmer's responsibility to ensure that the communication has completed before the application buffer is free for reuse.
Non-blocking routines are primarily used to overlap computation with communication. Every *MPI* send/receive routine has a non-blocking counterpart, e.g. `MPI_ISsend` and `MPI_Irecv`.

Appendix B Source Code for modified propagate_messages()

```c
/* ====================================================================== *
 * File: propagate_messages.c
 * Desc: Modified version of propagate_messages(). Uses non-blocking
 *       communication to avoid deadlocks, and customised buffering
 *       strategies to reduce memory requirements.
 *       Routine is split into propagate_messages_init() and
 *       propagate_messages_complete() so that non-dependent computation
 *       may be overlapped with the communication.
 *
 * Auth: L.S. Chin (l.s.chin@rl.ac.uk)
 *
 * NOTES: Annotations on ‘dynamic’ (model dependent) sections of this code
 *        are included in annotated_propagate_messages.c
 * ====================================================================== */

#include "header.h"

/* ========== Declaration of required constants =================== */

/* These values would be set in header.h by x-parser, depending on number of
   message types */
#define MESSAGE_LOCATION 0
#define MESSAGE_TYPE_COUNT 1


/* message tag used for propagating messages */
#define TAG 12


/* ========== Declaration of file level global vars ================ */

/* buffer space for packed message from/to each node */
void ** inbuf;
void ** outbuf;

/* vars associated to MPI communication */
MPI_Request *in_req;
MPI_Request *out_req;


/* ========== Begin implementation code ========================== */

/** \fn void propagate_messages_init()
 *  \brief Initiate propagation of messages between nodes using
 *      non-blocking communication routines.
 */
void propagate_messages_init() {
    /* <NOTE> This routine uses the following external global variables
     * + xmachine_message_location ** p_location_message (Pointer to
     *       message struct for looping through location message list)
     * + node_information ** p_node_info (Pointer to head of node list)
     * + int totalnodes  (Result of MPI_Comm_size)
     * + int node_number  (Result of MPI_Comm_rank)
     */

    /* ------------ Begin Variable declarations -------------- */
    int i, j, outcount, bufsize;
    int probed;
    int *message_count_list;
    MPI_Status status;

    node_information * node_info;
```

```
    /* pointers to temporary messages. One set for each message type */
    xmachine_message_location *message_location_temp;
    xmachine_message_location_data *message_location_list;


    /* ------------ Allocate Required Memory ----------------- */

    /* Memory for MPI send/receive requests */
    in_req  = (MPI_Request *)malloc(sizeof(MPI_Request) * totalnodes);
    out_req = (MPI_Request *)malloc(sizeof(MPI_Request) * totalnodes);

    /* Memory for buffer arrays */
    inbuf  = (void **)malloc(sizeof(void *) * totalnodes);
    outbuf = (void **)malloc(sizeof(void *) * totalnodes);


    /* -------- Fill send buffers and post non-blocking sends ---- */

    /* iterate through node list */
    node_info = *p_node_info;
    while(node_info)
    {
        i = node_info->node_id;
        /* don't send messages to self */
        if (i == node_number)
        {
            outbuf[i] = NULL;
            out_req[i] = MPI_REQUEST_NULL;
            node_info = node_info->next;
            continue;
        }

        bufsize = 0;
        outcount = 0;

        /* for each message type */
        outcount += node_info->location_message_no;


        /* build output buffer */
        if (outcount > 0)
        {
            /* mem requirements for message_count_list array */
            bufsize += sizeof(int) * MESSAGE_TYPE_COUNT;

            /* mem requirements for each message type messages */
            bufsize += sizeof(xmachine_message_location_data) * \
                       node_info->location_message_no;

            /* allocate required memory */
            outbuf[i] = (void *) malloc(bufsize);

            /* assign array pointers to relevant points in buffer */
            message_count_list = (int *) outbuf[i];

            message_count_list[MESSAGE_LOCATION] = node_info->location_message_no;
            message_location_list = (xmachine_message_location_data *)\
                           &message_count_list[MESSAGE_TYPE_COUNT];


            /* traverse message list and populate buffer */
            message_location_temp = node_info->location_messages;
            j = 0;
            while (message_location_temp)
```

```
            {
                message_location_list[j].id = message_location_temp->id;
                message_location_list[j].x = message_location_temp->x;
                message_location_list[j].y = message_location_temp->y;

                message_location_temp = message_location_temp->next;
                j++;
            }
            /* clear location messages from internal list */
            p_location_message = &node_info->location_messages;
            freelocationmessages();
            node_info->location_message_no = 0;
            p_location_message = &current_node->location_messages;


        }
        else /* nothing to send */
        {
            outbuf[i] = NULL;
        }

        /* post non-blocking send */
        MPI_Issend(outbuf[i], bufsize, MPI_BYTE, i, TAG, \
                MPI_COMM_WORLD, &out_req[i]);

        /* select next node */
        node_info = node_info->next;
    }



    /* ----- Prepare and post non-blocking receives ------------ */

    /* we expect (totalnodes - 1) incoming messages */
    probed = 0;
    inbuf[node_number] = NULL;
    in_req[node_number] = MPI_REQUEST_NULL;
    while (probed < (totalnodes - 1))
    {
        /* probe incoming messages for bufsize */
        MPI_Probe(MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, &status);

        /* get sender's id */
        i = status.MPI_SOURCE;

        /* get bufsize */
        MPI_Get_count(&status, MPI_BYTE, &bufsize);

        /* allocate memory for recv buffer */
        if (bufsize > 0)
        {
            inbuf[i] = (void *)malloc(bufsize);
        }
        else
        {
            inbuf[i] = NULL;
        }

        /* post non-blocking receive */
        MPI_Irecv(inbuf[i], bufsize, MPI_BYTE, i, TAG, \
                MPI_COMM_WORLD, &in_req[i]);

        probed ++;
    }
```

```

}


/** \fn void propagate_messages_complete()
 * \brief Complete propagation of messages between nodes
 */
void propagate_messages_complete() {

    /* <NOTE> This routine uses the following external global variables
     * + node_information ** p_node_info (Pointer to head of node list)
     * + int totalnodes  (Result of MPI_Comm_size)
     */

    /* ------------ Begin Variable declarations -------------- */
    int i, j, bufsize, sender;
    int *message_count_list;
    MPI_Status status;
    node_information * node_info;

    /* pointers to temporary messages. One set for each message type */
    xmachine_message_location *message_location_temp;
    xmachine_message_location_data *message_location_list;



    /* ----- Complete and process pending receives ------------------ */

    /* process received messages */
    /* instead of a Waitall, use Waitany so that we may overlap data
       handling time with other pending receives */
    for (j = 0; j < totalnodes - 1; j++)
    {
        /* wait for any receive to complete */
        MPI_Waitany(totalnodes, in_req, &sender, &status);

        /* determine size of message */
        MPI_Get_count(&status, MPI_BYTE, &bufsize);

        /* if size = 0, there's nothing for us to do */
        if (bufsize == 0) continue;

        /* Assign array pointers to apropriate locations in buffer */
        message_count_list = (int *)inbuf[sender];

        /* for each message type */
        message_location_list = (xmachine_message_location_data *)\
                                &message_count_list[MESSAGE_TYPE_COUNT];


        /* process received messages */
        for (i = 0; i < message_count_list[MESSAGE_LOCATION]; i++)
        {
            message_location_temp = (xmachine_message_location*) \
                                    add_location_message_internal();

            message_location_temp->id = message_location_list[i].id;
            message_location_temp->x = message_location_list[i].x;
            message_location_temp->y = message_location_list[i].y;
        }

        free(inbuf[sender]);
    }

```

```c
    free(inbuf);
    free(in_req);



    /* ------- Complete non-blocking sends ----------------------- */

    /* wait for all non-blocking sends to complete */
    MPI_Waitall(totalnodes, out_req, MPI_STATUSES_IGNORE);

    /* free allocated send buffers */
    for(i = 0; i < totalnodes; i++)
    {
        if (outbuf[i]) free(outbuf[i]);
    }
    free(outbuf);
    free(out_req);

}
```

Appendix C Source Code for modified propagate_agents()

```
/* ==================================================================== *
 * File: propagate_agents.c
 * Desc: Modified version of propagate_agents(). Uses non-blocking
 *       communication to avoid deadlocks, and customised buffering
 *       strategies to reduce memory requirements.
 *
 *       While not implemented here, routine can be easily split up into
 *       propagate_agents_init() and propagate_agents_complete() to allow
 *       for overlapping of computation with agent propagation.
 *
 *       Algorithm for moving agents from main list to outbound list has
 *       also been modified (simplified?)
 *
 * Auth: L.S. Chin (l.s.chin@rl.ac.uk)
 *
 * NOTES: Techniques for generating this code dynamically with the x-parser
 *    can be implied from the annotations in annotated_propagate_messages.c
 * ==================================================================== */

#include "header.h"

/* ========== Declaration of required constants =================== */

/* These values would be set in header.h by x-parser, depending on number of
   agent types */
#define AGENT_CIRCLE 0
#define AGENT_CIRCLE2 1
#define AGENT_TYPE_COUNT 2

/* message tag used for propagating agents */
#define TAG 11



/* ========== Begin implementation code ========================== */

/** \fn void propagate_agents()
 * \brief Check agent positions to see if any need to be moved to a another node.
 */
void propagate_agents() {

    /* ------------ Begin Variable declarations -------------- */

        int agent_type;
    int remove_agent;
    int sender, outcount;
    int *agent_count_list;
    int probed, i, j, bufsize;
    node_information *node_info;
        xmachine *prev_xmachine, *next_xmachine;
        double x_xmachine, y_xmachine, z_xmachine;


    /* variables used for MPI calls */
    MPI_Status  status;
    MPI_Request * in_req;
    MPI_Request * out_req;

    /* buffer space for packed agents from/to each PE */
    void ** inbuf;
    void ** outbuf;

```

```
    /* pointers to temporary agents */
    xmachine_memory_Circle *agent_Circle_list;
    xmachine_memory_Circle2 *agent_Circle2_list;



    /* ---- determine which agents should be moved ---------------- */

    /* loop through agent list */
    prev_xmachine = NULL;
    remove_agent = 0;
    current_xmachine = current_node->agents;
    while(current_xmachine)
    {
        next_xmachine = current_xmachine->next;

        if(current_xmachine->xmachine_Circle != NULL)
                {
                        x_xmachine = current_xmachine->xmachine_Circle->x;
                        y_xmachine = current_xmachine->xmachine_Circle->y;
                        z_xmachine = 0.0;
                        agent_type = 0;
                }
                else if(current_xmachine->xmachine_Circle2 != NULL)
                {
                        x_xmachine = current_xmachine->xmachine_Circle2->x;
                        y_xmachine = current_xmachine->xmachine_Circle2->y;
                        z_xmachine = 0.0;
                        agent_type = 1;
                }

        /* if any agent is located beyond our partition data, move it */
        if (
           x_xmachine < current_node->partition_data[0] ||
                   x_xmachine > current_node->partition_data[1] ||
                   y_xmachine < current_node->partition_data[2] ||
                   y_xmachine > current_node->partition_data[3] ||
                   z_xmachine < current_node->partition_data[4] ||
                   z_xmachine > current_node->partition_data[5] )
        {
            /* determine which node we should move agent to */

            /* loop through node list */
            node_info = *p_node_info;
            while(node_info)
            {
                if (node_info->node_id == node_number)
                { /* skip my own node */
                    node_info = node_info->next;
                    continue;
                }

                /* should agent be in this node? */
                if (
                    x_xmachine > node_info->partition_data[0] &&
                    x_xmachine < node_info->partition_data[1] &&
                    y_xmachine > node_info->partition_data[2] &&
                            y_xmachine < node_info->partition_data[3] &&
                            z_xmachine > node_info->partition_data[4] &&
                            z_xmachine < node_info->partition_data[5] )
                {
                    /* move agent to target node */
                    if (agent_type == 0)
                    {
```

```
                        node_info->Circle_agent_no ++;
                        current_xmachine->next = node_info->Circle_agents;
                        node_info->Circle_agents = current_xmachine;
                    }
                    if (agent_type == 1)
                    {
                        node_info->Circle2_agent_no ++;
                        current_xmachine->next = node_info->Circle2_agents;
                        node_info->Circle2_agents = current_xmachine;
                    }

                    /* flag agent for removal from current node */
                    remove_agent = 1;

                    /* no need to traverse the node list further */
                    node_info = NULL;
                }
                else
                {   /* continue searching */
                    node_info = node_info->next;
                }
            }

        }

        /* remove agent from current node if necessary */
        if (remove_agent)
        {
            remove_agent = 0; /* reset value */

            /* decrement agent count */
            current_node->agent_total --;

            /* if first agent is removed */
            if (prev_xmachine == NULL)
            {
                current_node->agents = next_xmachine;
            }
            else
            {
                prev_xmachine->next = next_xmachine;
            }
        }
        else
        {
            prev_xmachine = current_xmachine;
        }

        /* move on to next agent in list */
        current_xmachine = next_xmachine;
    }



    /* ------------ Allocate Required Memory ----------------- */

    /* MPI related dynamic memory requirements */
    in_req  = (MPI_Request *)malloc(sizeof(MPI_Request) * totalnodes);
    out_req = (MPI_Request *)malloc(sizeof(MPI_Request) * totalnodes);

    /* Memory for buffer arrays */
    inbuf  = (void **)malloc(sizeof(void *) * totalnodes);
    outbuf = (void **)malloc(sizeof(void *) * totalnodes);

```

```

    /* ------ fill send buffers and post non-blocking sends ------ */
    node_info = *p_node_info;
    while(node_info)
    {
        i = node_info->node_id;
        if (i == node_number)
        {
            outbuf[i] = NULL;
            out_req[i] = MPI_REQUEST_NULL;
            node_info = node_info->next;
            continue;
        }

        bufsize = 0;
        outcount = 0;

        /* for each agent type */
        outcount += node_info->Circle_agent_no;
        outcount += node_info->Circle2_agent_no;

        /* build output buffer */
        if (outcount > 0)
        {
            /* mem requirements for agent_count_list array */
            bufsize += sizeof(int) * AGENT_TYPE_COUNT;

            /* for each message type */
            /* mem requirements for Circle agents */
            bufsize += sizeof(xmachine_memory_Circle) * \
                       node_info->Circle_agent_no;
            /* mem requirements for Circle2 agents */
            bufsize += sizeof(xmachine_memory_Circle2) * \
                       node_info->Circle2_agent_no;

            /* allocate required memory */
            outbuf[i] = (void *) malloc(bufsize);

            /* assign array pointers to relevant points in buffer */
            agent_count_list = (int *) outbuf[i];

            agent_count_list[AGENT_CIRCLE]  = node_info->Circle_agent_no;
            agent_Circle_list = (xmachine_memory_Circle *) \
                 &agent_count_list[AGENT_TYPE_COUNT];

            agent_count_list[AGENT_CIRCLE2] = node_info->Circle2_agent_no;
            agent_Circle2_list = (xmachine_memory_Circle2 *) \
                 &agent_Circle_list[agent_count_list[AGENT_CIRCLE]];


            /* traverse agent list and populate buffer */
            j = 0;
            temp_xmachine = node_info->Circle_agents;
            while (temp_xmachine)
            {
                agent_Circle_list[j].id = temp_xmachine->xmachine_Circle->id;
                agent_Circle_list[j].x = temp_xmachine->xmachine_Circle->x;
                agent_Circle_list[j].y = temp_xmachine->xmachine_Circle->y;
                agent_Circle_list[j].fx = temp_xmachine->xmachine_Circle->fx;
                agent_Circle_list[j].fy = temp_xmachine->xmachine_Circle->fy;
                agent_Circle_list[j].radius = temp_xmachine->xmachine_Circle->radius;
                agent_Circle_list[j].iradius = temp_xmachine->xmachine_Circle->iradius;

                temp_xmachine = temp_xmachine->next;
```

```
                j++;
            }
            /* free list early to conserve memory */
            p_xmachine = &node_info->Circle_agents;
            freexmachines();
            node_info->Circle_agent_no = 0;


            /* repeat for next agent type */
            j = 0;
            temp_xmachine = node_info->Circle2_agents;
            while(temp_xmachine)
            {
                agent_Circle2_list[j].id = temp_xmachine->xmachine_Circle2->id;
                agent_Circle2_list[j].x = temp_xmachine->xmachine_Circle2->x;
                agent_Circle2_list[j].y = temp_xmachine->xmachine_Circle2->y;
                agent_Circle2_list[j].fx = temp_xmachine->xmachine_Circle2->fx;
                agent_Circle2_list[j].fy = temp_xmachine->xmachine_Circle2->fy;
                agent_Circle2_list[j].radius = temp_xmachine->xmachine_Circle2->radius;
                agent_Circle2_list[j].iradius = temp_xmachine->xmachine_Circle2->iradius;

                temp_xmachine = temp_xmachine->next;
                j++;
            }
            /* free list early to conserve memory */
            p_xmachine = &node_info->Circle2_agents;
            freexmachines();
            node_info->Circle2_agent_no = 0;


            /* Reset xmachine pointer */
            p_xmachine = &current_node->agents;

        }
        else
        {
            outbuf[i] = NULL;
        }

        /* post non-blocking send */
        MPI_Issend(outbuf[i], bufsize, MPI_BYTE, i, TAG, \
                MPI_COMM_WORLD, &out_req[i]);

        /* select next node */
        node_info = node_info->next;
    }



    /* ----- Prepare and post non-blocking receives ------------ */

    /* we expect (totalnodes - 1) incoming messages */
    probed = 0;
    inbuf[node_number] = NULL;
    in_req[node_number] = MPI_REQUEST_NULL;

    while (probed < (totalnodes - 1))
    {
        /* probe incoming messages for bufsize */
        MPI_Probe(MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, &status);

        /* get sender's id */
        i = status.MPI_SOURCE;

```

```
        /* get bufsize */
        MPI_Get_count(&status, MPI_BYTE, &bufsize);

        if (bufsize > 0)
        {
            inbuf[i] = (void *)malloc(bufsize);
        }
        else
        {
            inbuf[i]  = NULL;
        }

        /* post non-blocking receive */
        MPI_Irecv(inbuf[i], bufsize, MPI_BYTE, i, TAG, \
                MPI_COMM_WORLD, &in_req[i]);

        probed ++;
    }


    /* NOTE: routine can be split up here if we want to separate
     *       into _init() and _complete(). Some of the variables
     *       would have to be declared as global within this file.
     *       (as implemenented in propagate_messages.c)
     */


    /* ----- Complete and process pending receives ------------------ */

    /* process received messages */
    /* instead of a Waitall, use Waitany so that we may overlap data
       handling time with other pending receives */
    for (j = 0; j < totalnodes - 1; j++)
    {
        /* wait for any receive to complete */
        MPI_Waitany(totalnodes, in_req, &sender, &status);

        /* determine size of message */
        MPI_Get_count(&status, MPI_BYTE, &bufsize);

        /* if size = 0, there's nothing for us to do */
        if (bufsize == 0) continue;

        /* Assign array pointers to apropriate locations in buffer */
        agent_count_list = (int *) inbuf[sender];
        /* for each message type */
        agent_Circle_list = (xmachine_memory_Circle *) \
             &agent_count_list[AGENT_TYPE_COUNT];
        agent_Circle2_list = (xmachine_memory_Circle2 *) \
             &agent_Circle_list[agent_count_list[AGENT_CIRCLE]];


        /* process received messages */
        for (i = 0; i < agent_count_list[AGENT_CIRCLE]; i++)
        {
            add_Circle_agent( \
                    agent_Circle_list[i].id, \
                    agent_Circle_list[i].x, \
                    agent_Circle_list[i].y, \
                    agent_Circle_list[i].fx, \
                    agent_Circle_list[i].fy, \
                    agent_Circle_list[i].radius, \
                    agent_Circle_list[i].iradius \
                    );
```

```c

        }

        /* repeat for next agent type */
        for (i = 0; i < agent_count_list[AGENT_CIRCLE2]; i++)
        {
            add_Circle2_agent( \
                    agent_Circle2_list[i].id, \
                    agent_Circle2_list[i].x, \
                    agent_Circle2_list[i].y, \
                    agent_Circle2_list[i].fx, \
                    agent_Circle2_list[i].fy, \
                    agent_Circle2_list[i].radius, \
                    agent_Circle2_list[i].iradius \
                    );

        }

        free(inbuf[j]);
    }

    free(inbuf);
    free(in_req);



    /* ------- Complete non-blocking sends ----------------------- */

    /* wait for all non-blocking sends to complete */
    MPI_Waitall(totalnodes, out_req, MPI_STATUSES_IGNORE);

    /* free allocated send buffers */
    for(i = 0; i < totalnodes; i++)
    {
        if (outbuf[i]) free(outbuf[i]);
    }
    free(outbuf);
    free(out_req);

}
```

Appendix D Source Code for test application – mpisend.c

```
#include <mpi.h>

#define BUFMIN 10
#define BUFMAX 3000
#define STEPSIZE 10

/* simple test to determine if and when MPI_Send becomes blocking */
/* Since send buffer implementation is not fixed by the MPI standards.
   a program which runs successfully under one set of conditions may
   fail under another set */

int main(int argc, char ** argv) {

    /* dynamically sized buffer for send/receive */
    int *outbuf, *inbuf;
    int target;
    int dsize, i, j;

    /* MPI related vars */
    MPI_Comm w_comm = MPI_COMM_WORLD;
    MPI_Status status;
    int      w_size, w_rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(w_comm, &w_size);
    MPI_Comm_rank(w_comm, &w_rank);

    if (w_size < 2)
    {
        printf("This test is meant to be run with at least 2 procs\n");
        MPI_Finalize();
        return 1;
    }


    /* prep */
    dsize = sizeof(double);
    target = (w_rank == 0) ? 1 : 0; /* select other proc as target */

    if (w_rank == 0) printf("Testing message exchange using MPI_Send and \
            MPI_Recv for data size %d to %d bytes (in steps of %d bytes) :\n" \
            , BUFMIN * dsize, BUFMAX * dsize, STEPSIZE * dsize);

    /* Have proc 0 and 1 repeatedly exchange messages of increasing size */
    for (i = BUFMIN; i <= BUFMAX; i += STEPSIZE)
    {
        /* other procs do nothing except participate in Barriers. */
        if (w_rank > 1)
        {
            MPI_Barrier(w_comm);
            continue;
        }

        /* prepare dummy data */
        outbuf = malloc(dsize * i);
        inbuf  = malloc(dsize * i);
        for (j = 0; j < i; j++) inbuf[j] = (double)(w_rank + j);

        if (w_rank == 0) printf("Exchanging %d bytes\n", i * dsize);

        /* send message to target */
        MPI_Send(outbuf, i, MPI_DOUBLE, target, 0, w_comm);
```

```c

        /* receive message from target */
        MPI_Recv(outbuf, i, MPI_DOUBLE, target, 0, w_comm, &status);

        if (w_rank == 0) printf("----- DONE -----\n");

        /* Make sure exchange completed before proceeding to next round */
        MPI_Barrier(w_comm);

        /* release dummy data */
        free(outbuf);
        free(inbuf);

    }

    MPI_Finalize();

    return 0;
}
```