HPCx
CAPABILITY COMPUTING

Is MPI-2 suitable for Quantum Chemistry? Performance of passive target one-sided communications

H.J.J. van Dam[1], M. Wang[2], A.G. Sunderland[1], I.J. Bush[3], P.J. Knowles[2], M.F. Guest[4]

[1]CSE, STFC Daresbury Laboratory, Warrington WA4 4AD, UK
[2]School of Chemistry, University of Cardiff, Park Place, Cardiff CF10 3AT, UK
[3]NAG Ltd, Wilkinson House, Jordan Hill Road, Oxford OX2 8DR, UK
[4]ARCCA, Redwood Building, King Edward VII Ave, Cardiff CF10 3NB, UK

Abstract

Quantum chemistry calculations have a compelling requirement for one-sided communication given their irregular data access patterns and irregular task sizes. Hence the MPI-2 passive target one-sided communications seem attractive as a standards based foundation for the implementation of the main algorithms in the field. For successful deployment it is important that good performance can be achieved reliably across a wide variety of platforms. We have tested this performance using a simple shared counter kernel across a number of machines and MPI implementations, including both open source MPI implementations - OpenMPI and MPICH2 - and those on the current and previous generation of the UK's national academic supercomputers, HECToR and HPCx. We find that the performance varies greatly with the MPI implementation in question. The speed with which communications are progressed was found to vary up to 4 orders of magnitude in the kernel program depending on the MPI library used. As a result the "time to solution" for the kernel could vary by as much as about a factor 2, although greater impacts are anticipated for more complex algorithms such as Fock-builders.

This spread in performance relates to two aspects of the MPI-2 standard. The first is whether the MPI implementation takes a minimalistic approach to satisfy rule 11.7.2 on the progress of one-sided communications or goes beyond that. Secondly, does the MPI library implement full multi-threading support, as specified in section 12.4, or not. Overall, we find that the MPI-2 standard is not strong enough to guarantee that MPI libraries provide acceptable performance characteristics to use one-sided communications effectively. We suggest that the standard be strengthened to address these performance issues – without this it seems unlikely that the MPI-2 one-sided communications will be useful for real world applications.

This is a Technical Report from the HPCx Consortium.

Report available from http://www.hpcx.ac.uk/research/publications/HPCxTR0807.pdf

1 Introduction

Quantum chemistry is a discipline that has traditionally made extensive use of high performance computing systems. Because of this demand for compute cycles the field has also been an early adopter of parallel computing. Experience over the last 20 years has shown that the main challenges for parallel computing in quantum chemistry lay in the irregular data access patterns and the irregular task sizes. Both arise for example in the Hartree-Fock method, where the exchange contributions cause irregular data access requirements. The irregularity in the task sizes stems from the calculation of the 2-electron repulsion integrals over shell quartets of Gaussian basis functions. The cost of evaluating a shell quartet of integrals depends on how many Gaussian functions are used to approximate an exponential function and the angular momentum of the functions which determines their number - 1 for an S-function, 3 for a P-function, 6 for a D-function, etc. Ultimately, even the relative positions of the 4 centres at which the Gaussian functions are sited are important as this determines which terms can be omitted.

In the example described above it is extremely difficult to use two-sided communication approaches effectively. The complexity required to address the irregular data access pattern is very high. The situation is made worse by the fact that the differences in task sizes lead to major load balancing problems. Hence it has been widely recognized that the only way to arrive at a relatively simple and efficient quantum chemistry code is to use one-sided communications. This realisation is what in part has driven developments such as the Global Arrays [1]. Our experience to date suggests that one-sided communications are indeed very effective in addressing the communication requirements - indeed quantum chemistry codes based on such a paradigm, such as NWChem [2,3], MOLPRO [4], GAMESS [5,6], and GAMESS-UK [7], are some of the most effective parallel applications in the field.

The advent of MPI-2 [8] and its one-sided communication facilities offer an alternative standards based approach to implementing quantum chemistry codes. Indeed, Gropp et al. considered how the Global Arrays could be implemented using MPI-2 [9] and thereby, in principle, quantum chemistry codes. However, as the effort involved in adapting large codes is substantial it is sensible to first assess the performance on simple test cases. Typical test cases for quantum chemistry involve the shared counter algorithm and the Fock-builder. In this report we will concentrate on the first of these, the shared counter. Typically in our field this is used to distribute work on a first come first serve basis across all the processors, a work distribution we refer to as dynamic load balancing. It is important to realize that the shared counter test is a rather modest test because only a single communication is required per compute task. By contrast the Fock-builder requires six communications per compute task. Clearly the latter operation will be much more sensitive to any delays in the communication.

An important factor here is that we do not want to sacrifice a whole processor just to manage the shared counter. Instead we want all processors to be able to do useful work. Historically we have managed this through the use of the `ga_read_inc` function in the Global Arrays, and have found this to work very well. Here we will look at the issues involved in duplicating this functionality using MPI-2. Thus section 2 considers the implementation of the shared counter functionality and how this has been tested, while section 3 presents and analyses the associated performance data, section 4 examines the MPI-2 standard and discusses the performance measured in the light of this. The code that we used to measure the performance is presented in appendix A.

2 The shared counter

As discussed in the introduction quantum chemistry has historically made extensive use of dynamical load balancing approaches based on shared counters. Initially we used the `NXTVAL` function in TCGMSG [10], and more recently the atomic read-increment function in the Global Arrays for the same purpose. In both cases the concept is very simple. A single integer value is managed by process 0, with every process in a parallel calculation having access to this integer. Every access provides a process with the value of the integer, and increments the integer's value atomically. As a result it is guaranteed that a sequence of accesses provides a sequence of consecutive integers. With both the TCGMSG and the Global Array implementation the realisation of this counter is such that process 0 which holds the counter can do useful work without noticeably delaying the response to other processes trying to access the counter.

The implementation of a similar functionality based on MPI-2 is slightly more involved however. Obviously as it is unknown when an access to the shared counter is requested the only sensible implementation is to use passive target MPI-2 functions. The requirement to read the current value and increment it poses a challenge however. There are two things that need to happen and they have to happen always in the same order and atomically, i.e. without another process being able to interfere. However in MPI-2 these two requirements clash. If two messages are issued within a single epoch, e.g. `MPI_Get` and `MPI_Accumulate`, then these are not guaranteed to execute in the order they were issued. Breaking the operation up into two epochs however opens up the risk that another process may interfere between the `MPI_Get` epoch and the `MPI_Accumulate` epoch. Both events would result in an incorrect number sequence and are hence unacceptable.

The approach adopted in this report was suggested by David Henty [11] from EPCC and is essentially the same as the one used by Gropp et al. [9] in `fetchandadd.c` as provided with MPICH-2. In this approach process 0 holds an integer array `task_ctr` with an element for each process, with each element giving the number of tasks executed by the corresponding process. In addition every process holds a local counter `my_ctr` which counts the number of tasks the process has executed. The number of the next task is obtained by getting the values of the array associated with the other processes and inserting the value of the local counter for the current process - the number is given by the sum of all the array values. To update the shared counter the local counter is incremented by one and its value stored in the `task_ctr` element for the current process. The important characteristic of the approach is that every element of `task_ctr` is accessed only once per value obtained. Hence the whole operation can be completed in a single exposure epoch avoiding ordering problems and guaranteeing atomicity. The most obvious down side of this approach is the fact that the communication requirements grow linearly with the number of processes whereas they are constant with the `NXTVAL` function from TCGMSG and the similar function in the Global Arrays. Nevertheless one might hope that the performance of this shared counter is sufficient to scale to a few thousand processors, say, provided the task sizes are large enough. Gropp et al. [9] discuss a much more scalable tree based implementation (see `fetchandadd_tree.c`) as well but as our discussion focuses on the message progression characteristics the difference is not relevant to this discussion.

To test the performance of the MPI-2 one-sided communications based shared counter a small kernel, `program call_counters`, has been written. This kernel creates the counter, initialises it and then iteratively obtains values from it. After a counter value is obtained the process is kept busy in a compute loop provided by `subroutine compute` for some time. The kernel considers three different scenarios:

- Case 1, in which all processes request values from the shared counter and execute the compute loop.
- Case 2, which is the same as case 1 except that process 0 which holds the shared counter data remains idle waiting in `MPI_Barrier`. Only the other processes request counter values and perform the compute loop.
- Case 3, which is the same as case 1 except that every process now has an additional thread that is waiting in a blocking `MPI_Send` to progress messages.

In all three cases the average wall clock time to obtain a counter value is measured as well as the average wall clock time taken to execute the compute loop. In order for case 3 to work the kernel program always initialises MPI requesting full multi-threading support - the program will report if that is not available. Every processor will write its own file reporting the results.

This kernel has been run on a variety of platforms using a number of MPI libraries. The list is not exhaustive but includes enough settings to establish whether MPI one-sided communications are likely to work well or if portability problems are to be anticipated. The results of the runs are discussed in the next section.

3 Performance of the MPI-2 based shared counter kernel

The results obtained from running the kernel program on various platforms were analysed and a few characteristic quantities derived. For all cases it was found to be important to know the duration of the compute loop, for messages in a number of MPI implementations cannot be progressed unless the processor required to progress it is in an MPI call. As these calls are separated by executions of the compute loop the time this takes becomes an important factor in the communication performance.

Note that today machines built from multi-core nodes are the norm as a result there usually are different kinds of access to the shared counter. First there will be processes on the same node as the process holding the shared counter that can reach the latter without having to use the network. Secondly, there are processes that are reside on remote nodes and therefore have to use the network to access the shared counter. Obviously, the performance of both kinds of access may be different. In practice this difference does not qualitatively change the results. Hence the timings given below are always for process 1 accessing the shared counter on process 0. We'll comment where access from remote nodes shows significantly different performance.

In case 2 the performance obtained is interesting only in that it provides an idea of the response times under ideal conditions. We have little interest in using this approach in earnest, even for a shared counter. In practice we would like to employ MPI-2 one-sided messages for distributed data algorithms in which all processes hold parts of the overall data. In this case it is clearly not feasible to leave idle all processes that hold data for there would be no processes left to do work!

The kernel was run on the following platforms:

- The open source MPI implementations MPICH2 1.0.7 [12] and OpenMPI 1.2.8 [13] were tested on a single node Intel Core 2 Quad Q6700 processor. The MPICH2 library was built in three different ways, with the sockets channel, the Nemesis channel and the shm channel. The OpenMPI library was built with the default settings.
- The IBM p5-575 MPI 4.3.1.6 implementation was tested on HPCx [14].
- The Cray XT4 MPI 2.0.62 implementation was tested on HECToR [15].
- The IBM BlueGene/P MPI implementation was tested using the V1R2M0 software stack [16].
- The Bull MPI 2-1.7-2.t and Intel MPI 3.1 implementations were tested on Merlin a 256 node cluster providing 2048 Xeon cores connected by Infiniband (Connect-X) hardware [17].
- The HP MPI implementation was tested on HAPU a 128 core Opteron HP Cluster Platform 4000 machine using HP MPI version 02.02.00.02 [18].
- The SGI Message Passing Toolkit (MPT) 1.13 implementation was tested on CSESGI1 a 10 processor Itanium 2 SGI Prism Extreme system.

The SGI Prism Extreme machine is a shared memory machine unlike most of the other systems. We also tried an SGI Altix Ice system but the shared counter kernel failed because `MPI_Win_lock` and `MPI_Win_unlock` are not supported for the InfiniBand interconnects [19].

The results are presented in Table 1 and Figure 1. The former shows the average wall-clock time in seconds for executing the compute loop and the shared counter accesses for the different cases. In the instances under Case 3 where no number is given the MPI library did not support full multi-threading. The main observation from Table 1 is that Case 2 where process 0 is kept idle to progress the messages always results in good communication performance. Deviations from this in Cases 1 and 3 must be a function of the MPI implementation. Also note that the Cray XT4 MPI implementation seems to progress messages in half the time it takes to execute the compute loop. However, the reported time is particularly favourable as the intra-node access is considerably faster than the inter-node access on this machine the latter taking 4.24 seconds, i.e. nearly 4 times as long.

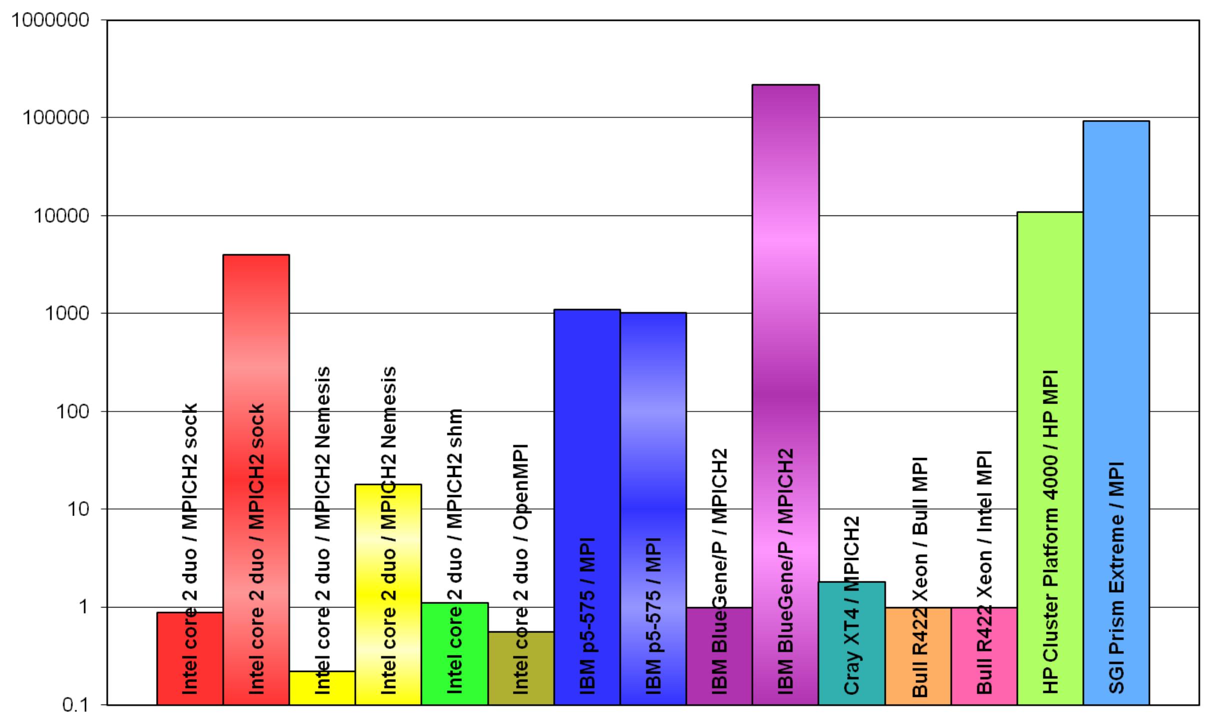
Figure 1 shows communications performance, defined as the time taken to execute the compute loop divided by the time taken to obtain a value of the shared counter. The solid columns show the performance for Case 1 where process 0 participates in doing work. Where available the shaded columns show Case 3 performance where an additional thread is available to progress the communication. The Case 3 results are presented directly to the right of the corresponding Case 1 results. The precise values are actually not that important in comparison to the trends. However, it is worth noting that a column height of 1 means that it takes just as long to get a shared counter value as it does to execute the compute loop.

Table 1: Communication times for different shared counter application cases.

Platform			Times (s)			
Hardware	MPI	# proc	Work	Case 1	Case 2	Case 3
Intel core 2 duo	MPICH2 sock	4	3.09	3.496540	0.000125	0.000767
Intel core 2 duo	MPICH2 nemesis	4	3.07	13.954472	0.000022	0.169123
Intel core 2 duo	MPICH2 shm	4	3.08	2.750339	0.000017	N/A
Intel core 2 duo	OpenMPI	4	3.17	5.609268	0.000034	N/A
IBM p5-575 / HPS	MPI	32	2.59	0.002357	0.000070	0.002528
IBM BlueGene/P	MPICH2	32	11.45	11.452927	0.000049	0.000052
Cray XT4	MPICH2	32	2.01	1.107902	0.000020	N/A
Bull R422 QC Intel Xeon E5472 / Connect X	Bull MPI	32	0.24	0.245136	0.000009	N/A
Bull R422 QC Intel Xeon E5472 / Connect X	Intel MPI	32	0.24	0.245279	0.000009	N/A
HP Cluster Platform 4000	HP MPI	32	4.12	0.000377	0.000470	N/A
SGI Prism Extreme	MPT	10	1.75	0.000019	0.000006	N/A

The figure shows that when process 0 is involved in progressing work and no additional threads are used to progress the communications (Case 1), then only the IBM p5-575, HP Cluster Platform 4000, and the SGI Prism Extreme systems with their proprietary MPI implementations maintain good communication performance. All other MPI implementations appear to keep processes (other than process 0) waiting while process 0 performs work. Finally, when multi-threading is tried to alleviate this problem (Case 3) most MPI implementations suffer from lack of multi-threading support. In fact only 4 implementations remain - MPICH2 with the sockets channel and MPICH2 with the Nemesis channel, the IBM p5-575, and the IBM BlueGene/P. Of these the MPICH2 with sockets and the IBM implementations maintain good performance. The MPICH2 with the Nemesis channel suffers considerable performance loss when other processes than process 0 try to access the shared counter.

Figure 1: Communication performance expressed as the compute loop time divided by the communication time on a logarithmic scale, the solid columns show the performance for Case 1 where process 0 participates in doing work, where available the shaded columns show Case 3 performance where an additional thread is available to progress the communication.



Considering the impact of the communication performance on an application the performance degradation can be defined as the time needed to obtain the shared counter value plus the compute time divide by the compute time, i.e. the inverse of the parallel efficiency. For the instances in Case 1 where the degradation factor significantly exceeds 1 this factor ranges from 1.55 for the intra-node accesses on the Cray XT4 to 5.54 for the MPICH2 Nemesis channel. In most cases the degradation factor is close to 2 meaning that the program takes twice as long due to delays in the communication than strictly necessary. This is a very large impact for a scenario where only one message per compute block is required.

Given the large impact that delays in the communication in Case 1 may have on the program performance it is discouraging to see that only 3 of the affected MPI implementations support full multi-threading. It is obvious that the ones that do support this realize major communication performance improvements due to this; 3 orders of magnitude for MPICH2 using sockets, 2 orders of magnitude for MPICH2 using Nemesis and 5 orders of magnitude for BlueGene/P in this particular test.

4 The MPI-2 standard and passive target one-sided communications

The findings of section 3 should be considered in the context of two important sections of the MPI-2 standard - section 11.7.2 entitled “Progress” and section 12.4 entitled “MPI and threads”.

Section 11.7.2 states that while a process is waiting in a blocking MPI call it must progress messages that involve this process. This statement is necessary to ensure that a correctly written MPI program using a standards compliant MPI library will correctly run to completion. Without it a correct MPI program could deadlock. This is easy to see when one considers a job with 2 processes. For example process 0 will only execute an `MPI_Barrier`, whereas process 1 performs an `MPI_Get` from process 0 and then executes the `MPI_Barrier`. Without rule 11.7.2 process 0 could choose not to progress the `MPI_Get` request while waiting in the `MPI_Barrier`, which would mean process 1 would never progress beyond the `MPI_Get` and the program would deadlock. Rule 11.7.2 ensures that process 0 will participate to complete the `MPI_Get` and hence the program will complete correctly.

Therefore we have clearly established that rule 11.7.2 is a minimal requirement for the correct execution of MPI programs and only that. Rule 11.7.2 does not state anything about the performance with which the program will complete, nor does it make any attempt to suggest or imply any performance issues other than that it guarantees that a program completes in finite time.

Indeed the data we have collected suggests that MPI implementations that only meet the minimum requirements set out by rule 11.7.2 lead to extremely poor performance characteristics in one-sided passive target communications. It is therefore highly disappointing to see that of all the tested MPI implementations there are only three, the IBM p5-575, the HP Cluster Platform 4000 and the SGI Prism Extreme implementations that provide message progress support that exceeds the minimum requirement [20]. The importance of this is evident from the potential to gain 4 orders of magnitude communication performance in our tests assuming that essentially Case 2 equivalent performance can be achieved.

Section 12.4 discusses the thread support MPI libraries may provide. However, it only prescribes the thread support a library has to provide if it provides such support. It does not insist that an MPI library has to provide any support for multi-threading. In practice we found that of all the MPI libraries we have tested only 4 supported full multi-threading. These libraries were MPICH2 with the sockets channel and with the Nemesis channel, the IBM p5-575, and the IBM BlueGene/P. The opportunity to exploit multi-threading is extremely valuable as is evident from MPICH2 sockets results where the use of an extra thread with case 3 resulted in an communication performance improvement of 3 orders of magnitude compared to case 1.

The combination of the very limited message progress support and the lack of multi-threading support combine to make attempts to exploit passive target one-sided communications almost completely pointless. If all processes are to participate in work then the weak message progress support will lead to unacceptably poor performance if no additional threads to drive the communication can be deployed.

Hence the only possible conclusion at present is that the MPI-2 standard is too weak to provide usable one-sided communications. Either the message progress requirements stated in rule 11.7.2 need to be strengthened to guarantee better performance, or stronger multi-threading support needs to be enforced. From a users perspective strengthening the message progress support is preferred as this avoids the need for additional program threads. Nevertheless in its current state it is hard to see that MPI-2 single sided communication has any rôle to play in quantum chemistry.

In the light of the above it is encouraging to see that within the MPI forum there are minds considering how one-sided communications can be improved. A particular instance is the recently published MPI-3 remote memory access proposal by Tipparaju et al. [21]. Hopefully this document will contribute to progressing these developments in the right direction.

5 Conclusion

We have considered the possibility of using MPI-2 passive target one-sided communications as the basis for implementing distributed data algorithms with complex data access patterns. The performance of this class of communications was tested using a simple shared counter. It was found that the performance was poor due to the fact that most MPI implementations do not progress messages faster than the minimum required by rule 11.7.2 and lack of multi-threading support as specified in section 12.4 of MPI-2 standard. This will likely lead to a decrease of performance by at least a factor of approximately 2 if a quantum chemistry code was based on this technology. We consider this loss of performance unacceptable. If MPI-2 to be useful for quantum chemistry applications then the standard needs to be strengthened to ensure acceptable levels of performance of one-sided communications for all standard compliant implementations.

Acknowledgements

The authors thank David Henty for providing them with a working C-implementation of the shared counter algorithm on which the one provided in the appendix is based.

The authors thank Cliff J. Noble for enlightening discussions about shared counters and their use in PFARM and PRMAT, and for pointing us to the book by Gropp et al.

This work made use of the facilities of HPCx, the UK's national high-performance computing service, which is provided by EPCC at the University of Edinburgh and by STFC Daresbury Laboratory, and funded by the Department for Innovation, Universities and Skills through EPSRC's High End Computing Programme.

This work made use of the facilities of HECToR, the UK's national high-performance computing service, which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd, and funded by the Office of Science and Technology through EPSRC's High End Computing Programme.

APPENDIX A The shared counter kernel

On the following pages the shared counter kernel code we used to investigate the performance of MPI-2 one-sided communications is presented. The code is a mixture of Fortran90 and C. Although the code is not implemented to be particularly neat it is simple and effective in demonstrating the main performance concerns.

Counter.f90 : the shared counter module

```
Module mpi_global_counter

  ! MPI2 Global counter module originally from A.G.Sunderland,
  ! slightly modified and extended by I.J. Bush. 08/2008 Modified again by AGS.
  ! 08/2008 Modified again by AGS: Deleted IJB's original modifications for IBM.

  Use newscf_modules
  Use mpi            ! should be the prefered way to include fortran interfaces
                     ! not supported widely at present though.

  Implicit None

! Include 'mpif.h'

  Public :: set_tasks, get_task, end_tasks, reset_tasks

  Private

  Integer, Save, Allocatable :: task_ctr(:)  ! task counter
  Integer, Save, Allocatable :: buffer(:)    ! task counter local buffer
  Integer, Save              :: win          ! window handle
  Integer, Save              :: myrank       ! processor number
  Integer, Save              :: com_size     ! # processors
  Integer, Save              :: my_ctr       ! the number of tasks I have done
  Integer, Save              :: size_int     ! stride size

Contains

  Subroutine set_tasks

! set up shared memory area defining the current task - BLOCKING

    Integer                    :: err, info, size_int
    Integer(MPI_ADDRESS_KIND) :: size_addr, lb_addr

! find rank and number of processors:
    Call MPI_COMM_RANK (MPI_COMM_GAMESS, myrank, err)
    Call MPI_COMM_SIZE (MPI_COMM_GAMESS, com_size, err)

    Call MPI_TYPE_GET_EXTENT (MPI_INTEGER, lb_addr, size_addr, err)

! create shared memory window on processor 0:
    size_int = size_addr
    If (myrank /= 0) size_addr = 0

    Allocate ( buffer(0:com_size-1) )

    If (myrank == 0 ) then
      Allocate ( task_ctr(0:com_size-1) )
    Else
      Allocate ( task_ctr(0:0) ) ! allocate 1 integer to create a valid address
    Endif

    Call MPI_WIN_CREATE (task_ctr, com_size*size_addr, size_int, &
         MPI_INFO_NULL, MPI_COMM_GAMESS, win, err)

    Call reset_tasks

  End Subroutine set_tasks

  Subroutine get_task (new_task)
```

```
! get and update task counter
    Integer, Intent(out)    :: new_task
    Integer                 :: err, assert, task, m1, p1, root, i
    Integer(KIND=MPI_ADDRESS_KIND) :: size_addr
    !Integer                            :: msglen
    !Character(LEN=MPI_MAX_ERROR_STRING) :: message

    task = -777
    assert = 0
    m1 = -1
    p1 = 1
    root = 0
    err = 0

    ! Outer window required to lock out other processors
    Call MPI_WIN_LOCK (MPI_LOCK_EXCLUSIVE, root, assert, win, err)
    if (err.ne.0) write(6,*)myrank," error MPI_WIN_LOCK win ",err

    size_addr = 0
    Call MPI_GET (buffer(0:myrank-1), myrank, MPI_INTEGER, &
         root, size_addr, myrank, MPI_INTEGER, win, err)
    if (err.ne.0) write(6,*)myrank," error MPI_GET A win ",err

    size_addr = myrank+1
    Call MPI_GET (buffer(myrank+1:com_size-1), com_size-myrank-1, MPI_INTEGER, &
         root, size_addr, com_size-myrank-1, MPI_INTEGER, win, err)
    if (err.ne.0) write(6,*)myrank," error MPI_GET B win ",err

    size_addr = myrank
    buffer(myrank) = my_ctr
    my_ctr = my_ctr + p1
    Call MPI_PUT(my_ctr, 1, MPI_INTEGER, &
         root, size_addr, 1, MPI_INTEGER, win, err)
    if (err.ne.0) write(6,*)myrank," error MPI_PUT C win ",err

    Call MPI_WIN_UNLOCK (root, win, err)
    if (err.ne.0) write(6,*)myrank," error MPI_WIN_UNLOCK D win ",err

    task = 0
    Do i = 0, com_size-1
      task = task + buffer(i)
    Enddo

    new_task = task
  End Subroutine get_task

  Subroutine reset_tasks

    ! Reset counter - BLOCKING

    Integer :: err

    Call MPI_BARRIER( MPI_COMM_GAMESS, err )
    my_ctr = 0
    If( myrank == 0 ) Then
       task_ctr = 0
    End If
    Call MPI_BARRIER( MPI_COMM_GAMESS, err )

  End Subroutine reset_tasks

  Subroutine end_tasks

! Kill MPI windows - BLOCKING

    Integer   :: err

    Call MPI_BARRIER (MPI_COMM_GAMESS, err)

    Call MPI_WIN_FREE (win , err)

    Deallocate( buffer)
    Deallocate( task_ctr )

  End Subroutine end_tasks

End Module mpi_global_counter
```

Dummy.f90 : a data module to pass the communicator

```
module newscf_modules
! AGS dummy version for testing counters.f90

Integer :: MPI_COMM_GAMESS

end module newscf_modules
```

Thread.c : the message progressing helper threads

```
#include <pthread.h>
#include <mpi.h>

#define TRUE  1
#define FALSE 0

pthread_t thread;     /* the thread object */
MPI_Comm thread_comm; /* communicator used to manage thread */
MPI_Status status;
int send_buffer = 0;
int recv_buffer;
int data_server_running = FALSE;
int thread_exit = 0;
int *thread_exit_ptr;
int source; /* where the terminate message will come from */
int destination; /* where the terminate message will go to */
int msg_id = 123;
int msg_len = 1;

void data_server(void)
{
   /* Provide the data server functionality by diving into and waiting in
      a MPI_recv call. Use the communicator set up in "create_thread" so
      that only the corresponding message from "destroy_thread" can
      satisfy the MPI_recv. When the message is received it releases the
      thread leading it to terminate.
   */
   if (MPI_Send(&send_buffer,msg_len,MPI_INTEGER,source,msg_id,thread_comm))
     MPI_Abort(MPI_COMM_WORLD,999);

   pthread_exit((void*)&thread_exit);
}

void create_thread(void)
{
   /* Create a new thread to handle the MPI comms

      - create a new communicator
      - create a new thread that will just dive into and wait in MPI_recv
   */
   int colour; /* colour needed to define processor group */
   int rank = 0;

   if (data_server_running) return;
   data_server_running = TRUE;

   /* The new communicator will include only one process so set the colour
      to the rank of this process */
   if (MPI_Comm_rank(MPI_COMM_WORLD,&colour)) MPI_Abort(MPI_COMM_WORLD,905);
   if (MPI_Comm_split(MPI_COMM_WORLD,colour,rank,&thread_comm))
MPI_Abort(MPI_COMM_WORLD,901);

   source = 0;
   destination = 0;

   /* Create the thread to hang in MPI to act as a data server */
   if (pthread_create(&thread,NULL,(void *(*)(void*))data_server,NULL))
MPI_Abort(MPI_COMM_WORLD,902);
}

void destroy_thread(void)
{
   /* Destroy the thread that hangs in the MPI_recv

      - send the message that release the MPI_recv and terminates the thread
      - clean up the MPI communicator
   */

   if (!data_server_running) return;

   /* Send message to release the data server thread */
   if (MPI_Recv(&recv_buffer,msg_len,MPI_INTEGER,destination,msg_id,thread_comm,&status))
     MPI_Abort(MPI_COMM_WORLD,998);

   /* Wait for the data server thread to terminate (should not be necessary) */
   if (pthread_join(thread,(void**)&thread_exit_ptr)) MPI_Abort(MPI_COMM_WORLD,911);

   /* Tidy up the communicator we used to manage the data server thread */
   if (MPI_Comm_free(&thread_comm)) MPI_Abort(MPI_COMM_WORLD,910);

   data_server_running = FALSE;
}

void create_thread_(void)
{
    create_thread();
}
void destroy_thread_(void)
{
    destroy_thread();
}

void create_thread__(void)
{
    create_thread();
}
void destroy_thread__(void)
{
    destroy_thread();

}
```

Drive_counter.f90 : the main test program

```
program call_counters

use newscf_modules
use mpi_global_counter
use mpi

implicit none

!include 'mpif.h'

logical :: opr
integer :: ier, iam , nprocs, iunit
integer, parameter :: n = 10
integer :: itask, itask_old
character(len=80) :: outputname
double precision cpu0,cpu1,cpu2,cpu3,cpu4 ! timers
double precision elapsed_comm   ! time spend in communications
double precision elapsed_work   ! time spend doing work
double precision elapsed_wait   ! time spend waiting for barrier
double precision elapsed_total  ! total time spend
double precision sum
integer :: ntask ! the number of tasks done by this processor
integer :: requested, provided  ! level of MPI thread support
common/flop/sum

opr = .false.
```

```
requested = MPI_THREAD_MULTIPLE
call MPI_INIT_THREAD(requested,provided,ier)
call MPI_COMM_RANK(MPI_COMM_WORLD,iam,ier)

if (iam.eq.0) then
  open(UNIT=10,FILE="counter_correct.out")
  do itask = 0, n
    write(10,*) ' Processor ',iam,' doing task ', itask
  enddo
  close(10)
  itask = 0
endif

write(outputname,'("counter.",i5,".out")')iam+10000
iunit=11
if (iunit.ne.6) open(UNIT=iunit,FILE=outputname)
opr = (iam == 0)
opr = .true.
if (requested /= provided) then
   write(iunit,*)'Thread support requested is not available'
   write(iunit,*)'Requested=',requested,' provided=',provided
endif

call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ier)

call MPI_COMM_DUP(MPI_COMM_WORLD,MPI_COMM_GAMESS,ier)

call set_tasks()
itask_old = -1
itask = -999

if (opr) write(iunit,*) 'Case 1: Processor root(0) does CPU work ',MPI_WTICK()
elapsed_comm  = 0.0
elapsed_work  = 0.0
elapsed_wait  = 0.0
elapsed_total = 0.0
ntask = 0
call MPI_Barrier(MPI_COMM_WORLD,ier)
cpu0 = MPI_Wtime()
call MPI_Barrier(MPI_COMM_WORLD,ier)
task_loop: do
  cpu1 = MPI_Wtime()
  call get_task(itask)
  cpu2 = MPI_Wtime()

  if  (itask == itask_old) then
    if (opr) write(iunit,*) ' Processor ',iam,' doing task ', itask ,' did task ',itask_old,'
TROUBLE'
    exit task_loop
  endif

  if  (itask <= n*nprocs) then
    if (opr) write(iunit,*) ' Processor ',iam,' doing task ', itask
    call compute(sum)
    cpu3 = MPI_Wtime()
    ntask = ntask + 1
  else
    exit task_loop
  end if

  elapsed_comm  = elapsed_comm  + cpu2-cpu1
  elapsed_work  = elapsed_work  + cpu3-cpu2

  itask_old = itask
end do task_loop
cpu3 = MPI_Wtime()
call MPI_Barrier(MPI_COMM_WORLD,ier)
cpu4 = MPI_Wtime()
elapsed_wait  = elapsed_wait  + cpu4-cpu3
elapsed_total = elapsed_total + cpu4-cpu0

write(iunit,100) iam,ntask,elapsed_comm/ntask,elapsed_work/ntask,elapsed_wait,elapsed_total
100 format('Times. rank=',i3,' my # tasks=',i6,' time/get=',f16.10,&
 &  ' time/compute=',f16.10,' time wait=',f16.10,' time tot.=',f16.10)

call reset_tasks()
```

```
if (opr) write(iunit,150)
150 format(/' Case 2: Processor root(0) does not do CPU work')
elapsed_comm  = 0.0
elapsed_work  = 0.0
elapsed_total = 0.0
ntask = 0
call MPI_Barrier(MPI_COMM_WORLD,ier)
cpu0 = MPI_Wtime()
call MPI_Barrier(MPI_COMM_WORLD,ier)
if (iam /= 0) then
  task_loop2: do
    cpu1 = MPI_Wtime()
    call get_task(itask)
    cpu2 = MPI_Wtime()

    if  (itask == itask_old) then
      if (opr) write(iunit,*) ' Processor ',iam,' doing task ', itask ,' did task
',itask_old,' TROUBLE'
      exit task_loop2
    endif

    if  (itask <= n*nprocs) then
      if (opr) write(iunit,*) ' Processor ',iam,' doing task ', itask
      call compute(sum)
      cpu3 = MPI_Wtime()
      ntask = ntask + 1
    else
      exit task_loop2
    end if

    elapsed_comm  = elapsed_comm  + cpu2-cpu1
    elapsed_work  = elapsed_work  + cpu3-cpu2

    itask_old = itask
  end do task_loop2
endif
cpu3 = MPI_Wtime()
call MPI_Barrier(MPI_COMM_WORLD,ier)
cpu4 = MPI_Wtime()
elapsed_wait  = elapsed_wait  + cpu4-cpu3
elapsed_total = elapsed_total + cpu4-cpu0

if (iam==0) then
  write(iunit,100) iam,ntask,elapsed_comm,elapsed_work,elapsed_wait,elapsed_total
else
  write(iunit,100) iam,ntask,elapsed_comm/ntask,elapsed_work/ntask,elapsed_wait,elapsed_total
endif

call reset_tasks()

if (opr) then
  write(iunit,*)
  write(iunit,*) 'Case 3: Processor root(0) does CPU work but data server present'
endif
elapsed_comm  = 0.0
elapsed_work  = 0.0
elapsed_wait  = 0.0
elapsed_total = 0.0
ntask = 0

call create_thread

call MPI_Barrier(MPI_COMM_WORLD,ier)
cpu0 = MPI_Wtime()
call MPI_Barrier(MPI_COMM_WORLD,ier)
task_loop3: do
  cpu1 = MPI_Wtime()
  call get_task(itask)
  cpu2 = MPI_Wtime()

  if  (itask == itask_old) then
    if (opr) write(iunit,*) ' Processor ',iam,' doing task ', itask ,' did task ',itask_old,'
TROUBLE'
    exit task_loop3
  endif

  if  (itask <= n*nprocs) then
```

```
    if (opr) write(iunit,*) ' Processor ',iam,' doing task ', itask
    call compute(sum)
    cpu3 = MPI_Wtime()
    ntask = ntask + 1
  else
    exit task_loop3
  end if

  elapsed_comm  = elapsed_comm  + cpu2-cpu1
  elapsed_work  = elapsed_work  + cpu3-cpu2

  itask_old = itask
end do task_loop3
cpu3 = MPI_Wtime()
call MPI_Barrier(MPI_COMM_WORLD,ier)
cpu4 = MPI_Wtime()
elapsed_wait  = elapsed_wait  + cpu4-cpu3
elapsed_total = elapsed_total + cpu4-cpu0

call destroy_thread

write(iunit,100) iam,ntask,elapsed_comm/ntask,elapsed_work/ntask,elapsed_wait,elapsed_total

call end_tasks()

if (iunit.ne.6) close(iunit)

call MPI_FINALIZE(ier)

end program call_counters

subroutine compute(sum)
   parameter(nn=600)
   integer i,j,k
   double precision sum
   sum=0.0
   do i=1,nn
      do j=1,nn
         do k=1,nn
            sum=sum+23.7*i+j/10.0-k/2.8
         enddo
      enddo
   enddo

end
```

References

[1] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease and Edo Apra, “Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit”, International Journal of High Performance Computing Applications, Vol. 20, No. 2, 203-231p, 2006 (http://www.emsl.pnl.gov/docs/global/)

[2] E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, D. Wang, E. Apra, T. L. Windus, J. Hammond, P. Nichols, S. Hirata, M. T. Hackler, Y. Zhao, P.-D. Fan, R. J. Harrison, M. Dupuis, D. M. A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan, Q. Wu, T. Van Voorhis, A. A. Auer, M. Nooijen, E. Brown, G. Cisneros, G. I. Fann, H. Fruchtl, J. Garza, K. Hirao, R. Kendall, J. A. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell, D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan, K. Dyall, D. Elwood, E. Glendening, M. Gutowski, A. Hess, J. Jaffe, B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield, X. Long, B. Meng, T. Nakajima, S. Niu, L. Pollack, M. Rosing, G. Sandrone, M. Stave, H. Taylor, G. Thomas, J. van Lenthe, A. Wong, and Z. Zhang, "NWChem, A Computational Chemistry Package for Parallel Computers, Version 5.1" (2007), Pacific Northwest National Laboratory, Richland, Washington 99352-0999, USA.

[3] R.A. Kendall, E. Apra, D.E. Bernholdt, E.J. Bylaska, M. Dupuis, G.I. Fann, R.J. Harrison, J. Ju, J.A. Nichols, J. Nieplocha, T.P. Straatsma, T.L. Windus and A.T. Wong, "High Performance Computational Chemistry: An Overview of NWChem a Distributed Parallel Application", Computer Phys. Comm. 2000, 128, 260-283.

[4] H.-J. Werner, P.J. Knowles, R. Lindh, F.R. Manby, M. Schütz, P. Celani, T. Korona, A. Mitrushenkov, G. Rauhut, T.B. Adler, R.D. Amos, A. Bernhardsson, A. Berning, D.L. Cooper, M.J.O. Deegan, A.J. Dobbyn, F. Eckert, E. Goll, C. Hampel, G. Hetzer, T. Hrenar, G. Knizia, C. Köppl, Y. Liu, A.W. Lloyd, R.A. Mata, A.J. May, S.J. McNicholas, W. Meyer, M.E. Mura, A. Nicklass, P. Palmieri, K. Pflüger, R. Pitzer, M. Reiher, U. Schumann, H. Stoll, A.J. Stone, R. Tarroni, T. Thorsteinsson, M. Wang and A. Wolf, "MOLPRO, version 2008.1, a package of ab initio programs", http://www.molpro.net, Cardiff, UK, 2008.

[5] M.W.Schmidt, K.K.Baldridge, J.A.Boatz, S.T.Elbert, M.S.Gordon, J.H.Jensen, S.Koseki, N.Matsunaga, K.A.Nguyen, S.Su, T.L.Windus, M.Dupuis, and J.A.Montgomery, "General Atomic and Molecular Electronic Structure System", J. Comput. Chem., 14, 1347-1363 (1993).

[6] M.S.Gordon and M.W.Schmidt, "Advances in electronic structure theory: GAMESS a decade later", pp. 1167-1189, in "Theory and Applications of Computational Chemistry: the first forty years", C.E.Dykstra, G.Frenking, K.S.Kim and G.E.Scuseria (editors), Elsevier, Amsterdam, 2005.

[7] GAMESS-UK is a package of ab initio programs. See: "http://www.cfs.dl.ac.uk/gamess-uk/index.shtml", M.F. Guest, I. J. Bush, H.J.J. van Dam, P. Sherwood, J.M.H. Thomas, J.H. van Lenthe, R.W.A Havenith, J. Kendrick, "The GAMESS-UK electronic structure package: algorithms,

developments and applications", Molecular Physics, Vol. 103, No. 6-8, 20 March-20 April 2005, 719-747.

[8] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 2.1). Technical report, 2008. (http://www.mpi-forum.org)

[9] W. Gropp, E. Lusk, R. Thakur, “Using MPI-2: Advanced Features of the Message-Passing Interface”, MIT Press, 1999, ISBN 0-262-057133-1.

[10] R.J. Harrison, The TCGMSG Message-Passing Toolkit, Pacific Northwest National Laboratory, version 4.04, 1994.

[11] David Henty, HPCx Query Q71214 response, 26 August 2008.

[12] MPICH2 version 1.0.7 (http://www.mcs.anl.gov/mpi/mpich2)

[13] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall, “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”, In Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004; OpenMPI version 1.2.8.

[14] HPCx (http://www.hpcx.ac.uk)

[15] HECToR (http://www.hector.ac.uk)

[16] BlueGene/P, hosted at STFC Daresbury Laboratory.

[17] Merlin, (http://www.arcca.cf.ac.uk)

[18] HAPU, (http://request.dl.ac.uk/hosts/hapu.live)

[19] SGI MPT 1.20, release notes, http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=linux&db=relnotes&fname=/usr/relnotes/sgi-mpt-1.20

[20] Su-Hsuan Huang, Chulko Kim, Richard R. Treumann, and William G. Tuel, “Method for implementing MPI-2 one sided communication”, US Patent 2008/0127203 A1, 29 May 2008.

[21] Edo Apra, Ronald Brightwell, Richard Graham, Robert Harrison, Jarek Nieplocha, Howard Pritchard, Galen Shipman, Vinod Tipparaju, and Jeffrey Vetter, “MPI3-RMA: A flexible, high-performance RMA interface for MPI”, 4 September 2008 (https://svn.mpi-forum.org/trac/mpi-forum-web/raw-attachment/wiki/RmaWikiPage/ORNL_mpi3-RMA_draft1.pdf)