Toward MPI Asynchronous Progress on Cray XE Systems

Howard Pritchard – howardp@cray.com
Duncan Roweth
David Henseler
Paul Cassella

Cray, Inc.

Overview

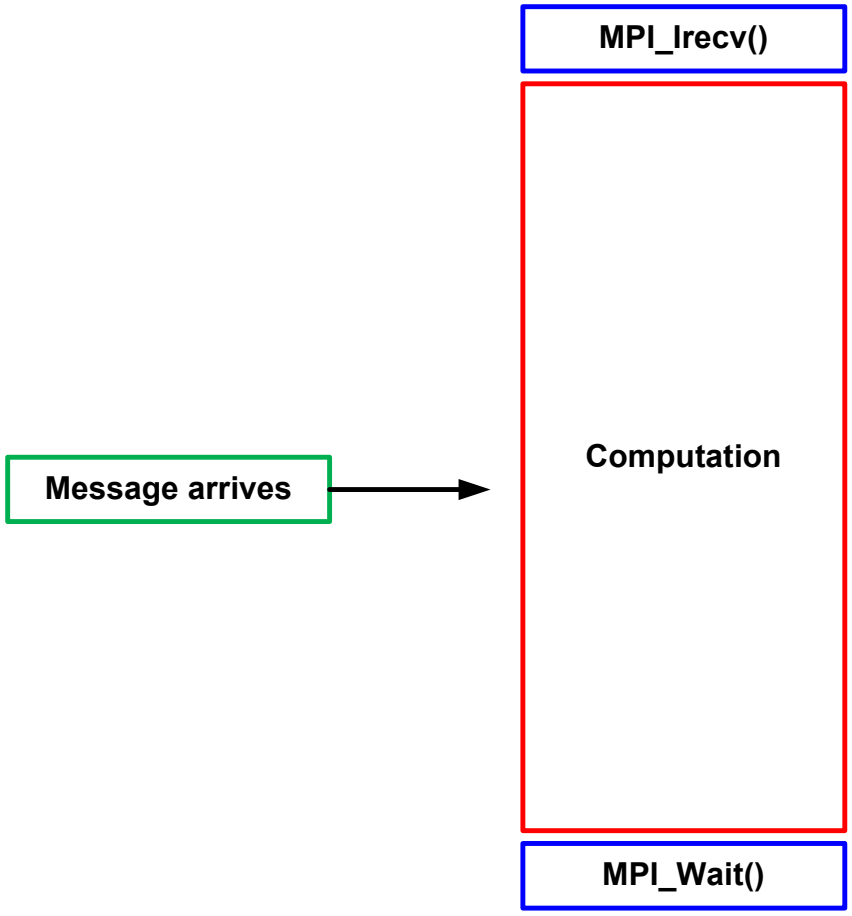
- MPI Progress - what is it?
- XE6 (Gemini) Features relevant to MPI progress
- MPICH2 Enhancements for MPI progress on Gemini
- Core Specialization and MPI progress
- Benchmark and application results
- Future work

MPI Progress Rule

- **Determines how an MPI communication operation completes once it has been initiated**
- **Strict interpretation**
  - *Once a communication has been initiated no further MPI calls are required in order to complete it*
- **Weak interpretation**
  - *An application must make further MPI library calls in order to make progress*
- **Primarily an issue for point-to-point calls, but also relevant to non-blocking collectives – to be introduced in MPI version 3.**

Receive side progression example

- *A well written application should post its receives early so as to overlap data transfer with computation*
- But messages typically arrive during the computation
- Matching occurs and bulk data transfer starts in the next MPI call, MPI_Wait() if only weak MPI progression interpretation supported
- How do we achieve independent progression?
  - Asynchronous progress thread(s)
  - Offload to the NIC



What Gemini Hardware Provides

Gemini Hardware Features and MPI Progress

- **DMA Engine with four virtual channels**
  - Local interrupt can be generated along with a Completion Queue Event (CQE) upon completion of a TX descriptor (RDMA transaction)
  - Remote interrupt and remote CQE can be generated at a target node when a RDMA transaction has completed
  - RDMA fence capability
- **Completion Queues configurable for polling or blocking (interrupt driven)**
- **Gemini has low overhead method for a process operating in user-space to generate an interrupt at a remote node (along with a CQE)**
- **No explicit support for MPI**

MPICH2 Enhancements to Support Asynchronous Progress

MPICH2 on XE - Basics

- Based on Argonne National Lab (ANL) MPICH2 Nemesis CH3 channel
- A GNI Nemesis Network Module (Netmod) interfaces the upper part of MPICH2 to the XE network software stack
- Full support for MPI-2 thread safety level MPI_THREAD_MULTIPLE
- Progress thread infrastructure (although simplistic) already exists in the library

MPICH2 on XE Basics (2)

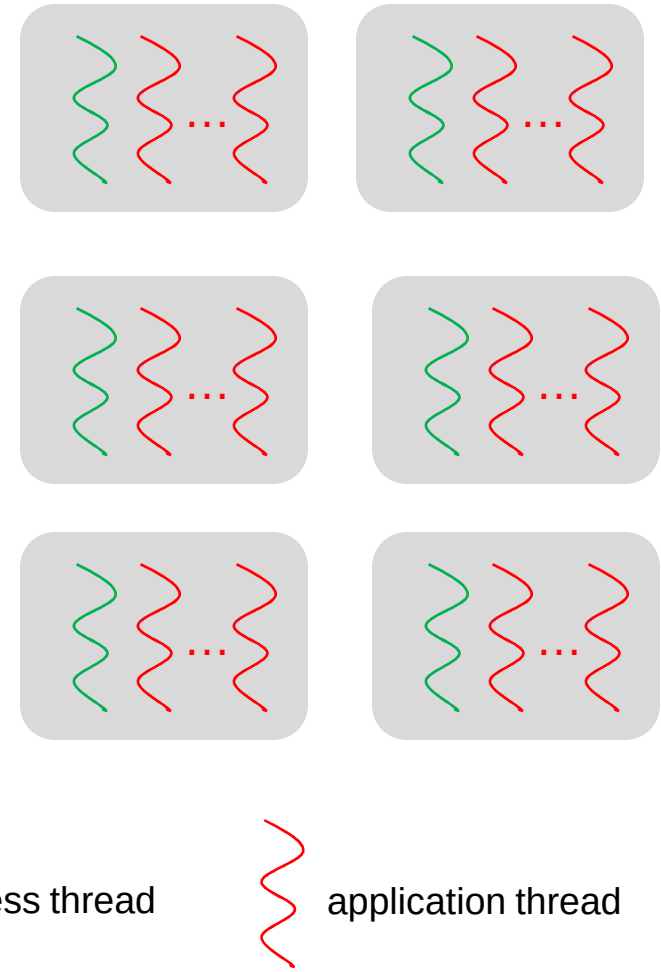
- **The eager protocol both for intra-node and inter-node messaging is CPU intensive and not suitable for asynchronous progress**
- **The rendezvous path for intra-node messages is also CPU intensive and not suitable for asynchronous progress**
- **The rendezvous – Long Message Transfer (LMT) – path for inter-node messages is not CPU intensive (usually):**
  - Rendezvous messages up to 4 MB in size normally use a RDMA read LMT protocol
  - Messages larger than 4 MB, or in the case of shorter messages when hardware resource depletion is occurring, use a cooperative RDMA write LMT protocol
  - The Gemini DMA engine (BTE) is normally used for these transfers

Enhancing MPICH2 on XE for Better MPI Progress - Goals

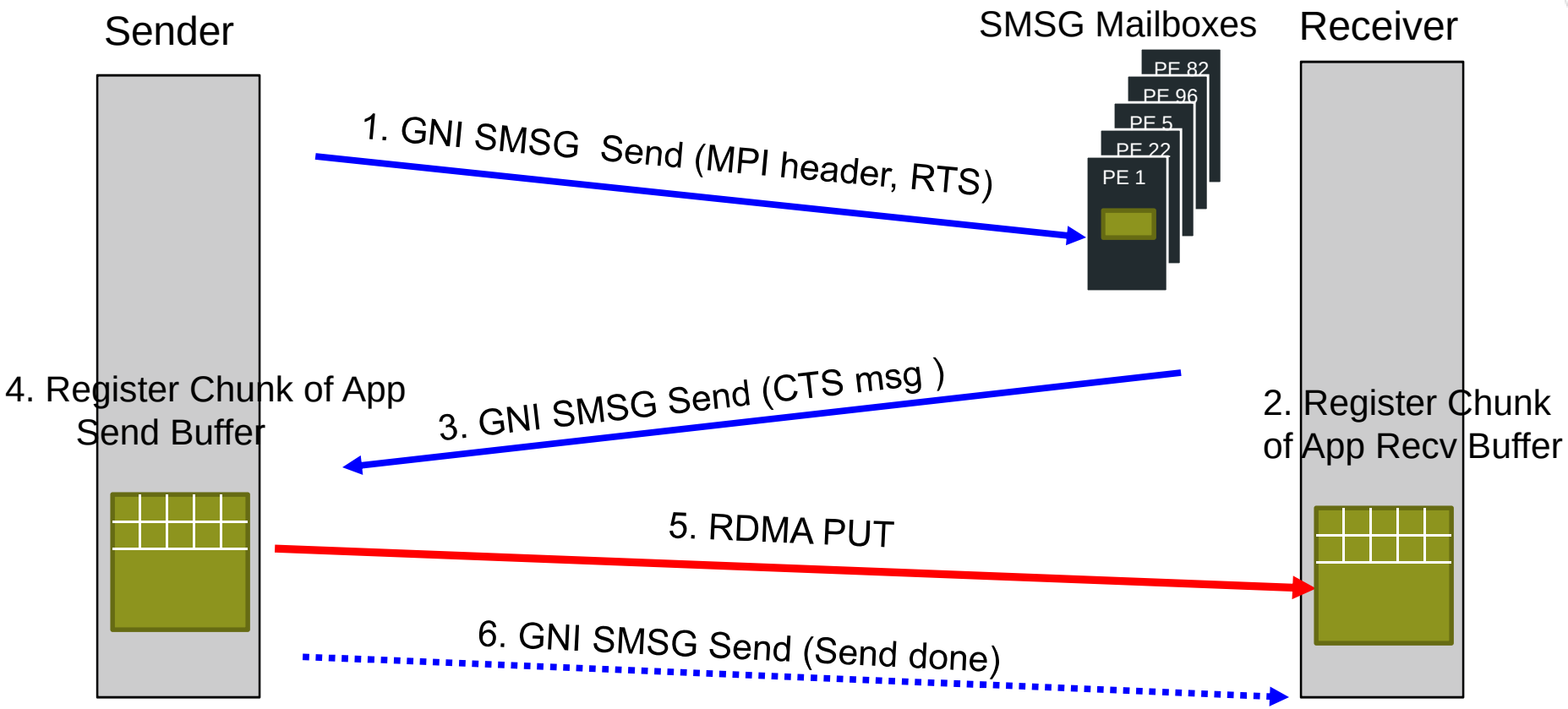
- Keep complexity/new software to a minimum
- Minimize impact on short message latency/message rate
- Don’t focus on a single message transfer protocol path for providing MPI progress (e.g. don’t focus just on RDMA read protocol)
- First phase focuses on better progression of longer message sizes (32 – 64 KB or longer)

MPICH2 - Progress Thread Model

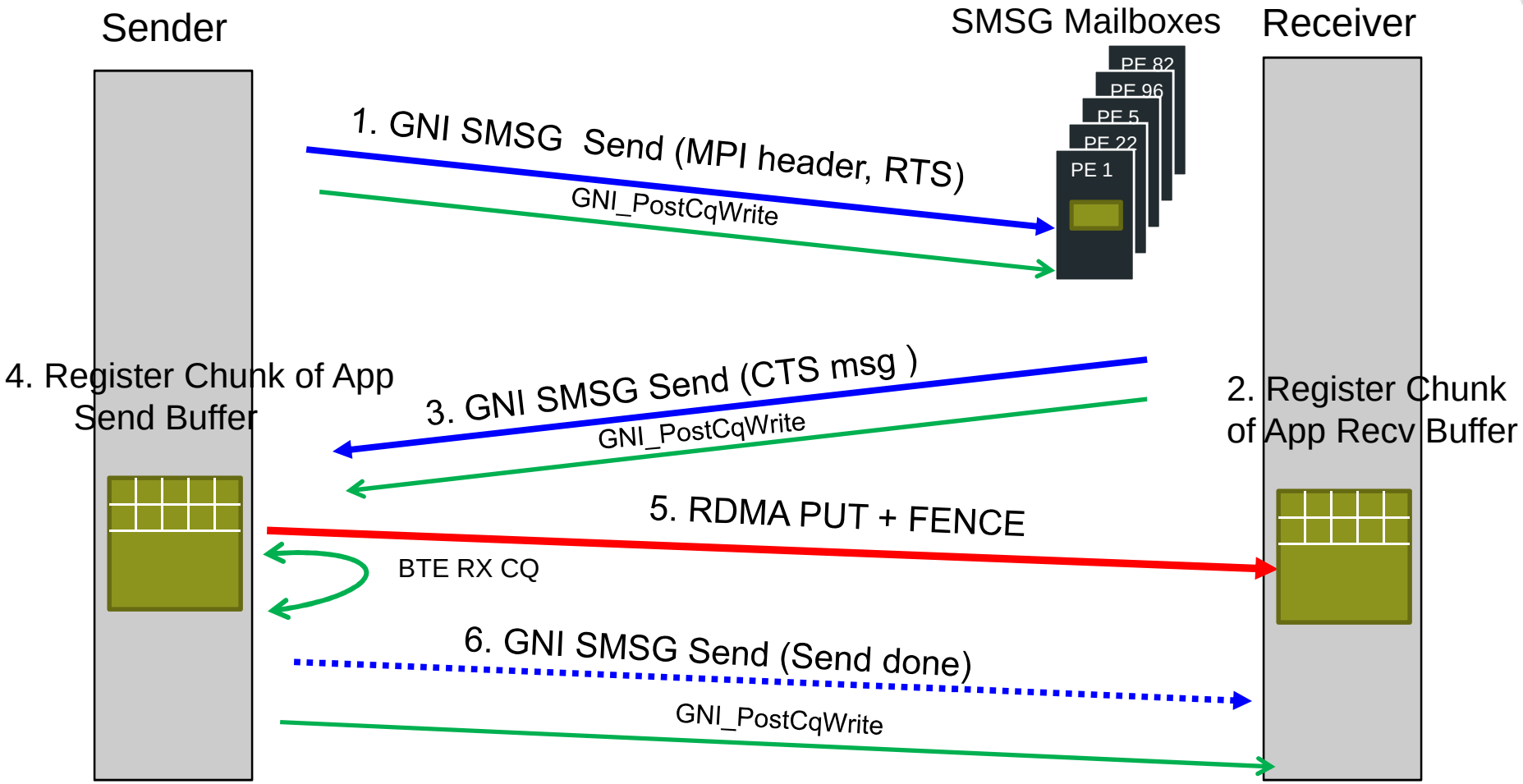
- Each MPI rank on a node starts an extra progress thread during MPI_Init.
- An extra completion queue (CQ) is created with blocking attribute during MPI_Init
- The progress threads call *GNI_CqWait* and remain blocked in the kernel until an interrupt is delivered by the Gemini NIC indicating progress on an MPI message receive or send
- Interrupts are only generated when progressing inter-node LMT (rendezvous) messages



LMT RDMA Write Path – No Progress



LMT RDMA Write Path – With Progress



Core Specialization and MPI Progress

Core Specialization

- Modifications to the Linux kernel to allow for dynamic partitioning of the cores on a node between application threads and OS kernel threads and system daemons
- Provides an interface to allow application libraries to specify whether threads/processes it creates are to be scheduled on the application partition or the OS partition
- User of a parallel application specifies at launch time how many cores, if any, per node to reserve for the OS partition

Core Specialization and MPI progress

- Typical HPC application threads tend to run **hot**, i.e. they don't typically make calls that result in yielding of the core on which they are scheduled
- Because of this, MPI progress threads need to have at least one core of a *compute unit* available per node for efficient handling of interrupts received from Gemini
- Core Specialization provides a convenient way to partition cores on a node between **hot** application threads, and **cool** system service daemon threads as well as MPI progress threads.

Job Container/Corespec Example

Tell job container package that the next thread/process created by this process should not be placed like an application thread/process.

Tell CoreSpec package that this task (thread) should be scheduled on OS partition if one is available, otherwise use default Linux scheduler policy.

```
int disable_job_aff_algo(void) {
    int rc;
    int fd = open("/proc/job", O_WRONLY);
    if (fd == -1){
        return -1;
    }
    rc = write(fd, "disable_affinity_apply",
     strlen("disable_affinity_apply"));
    if (rc < 0){
        close(fd);
        return -1;
   }
   close(fd);
   return 0;
}

int task_is_not_app(void) {
    size_t count;
    FILE *f;
    char zero[]="0";
    char   filename[PATH_MAX];

    snprintf(filename, sizeof(filename),"/proc/self/task/%ld/task_is_app",
             syscall(SYS_gettid));
    f = fopen(filename, "w");
    if (!f) {
        return -1;
    }

    count = fwrite(zero,sizeof(zero),1,f);
    if (count != 1) {
        fclose(f);
        return -1;
    }
    fclose(f);
    return 0;
}
```

MPI Asynchronous Progress - enabling

- export MPICH_NEMESIS_ASYNC_PROGRESS=1
- export MPICH_MAX_THREAD_SAFETY=multiple

Results

Sandia MPI Benchmark (mpi_overhead)

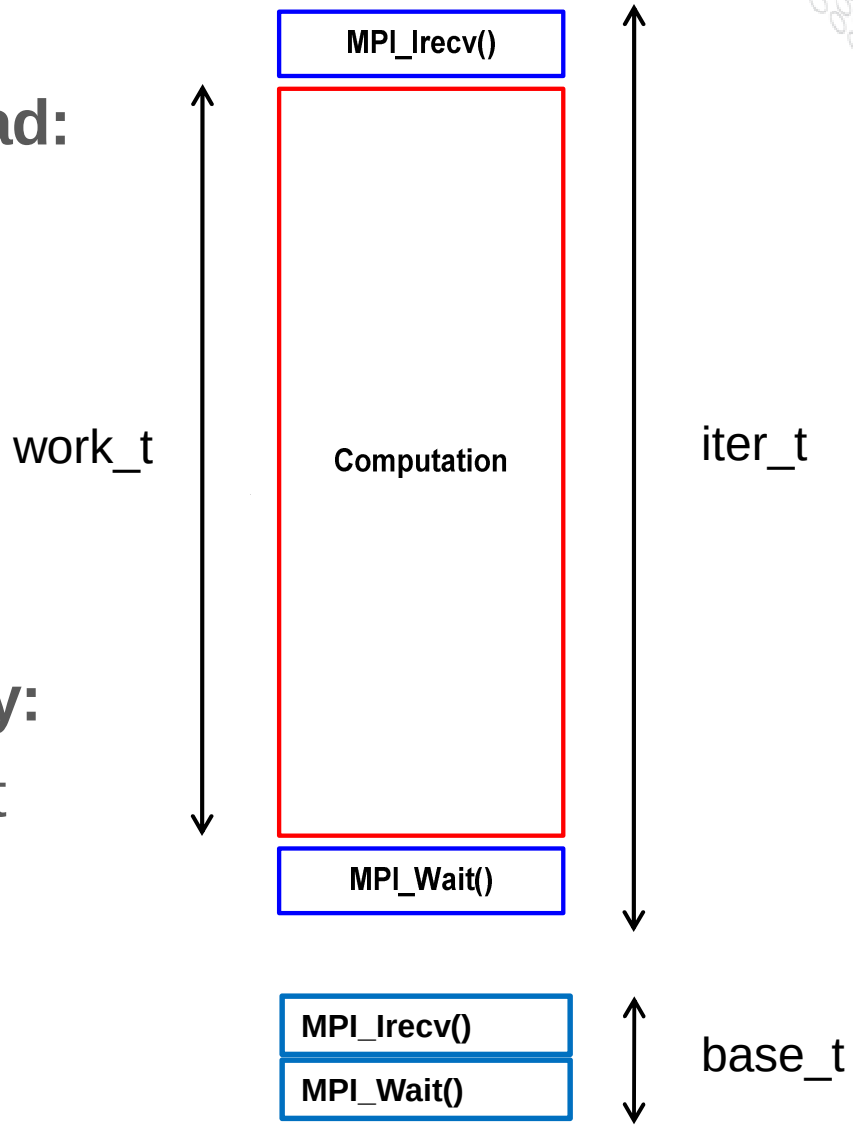
Measures MPI message overhead:

overhead = iter_t – work_t

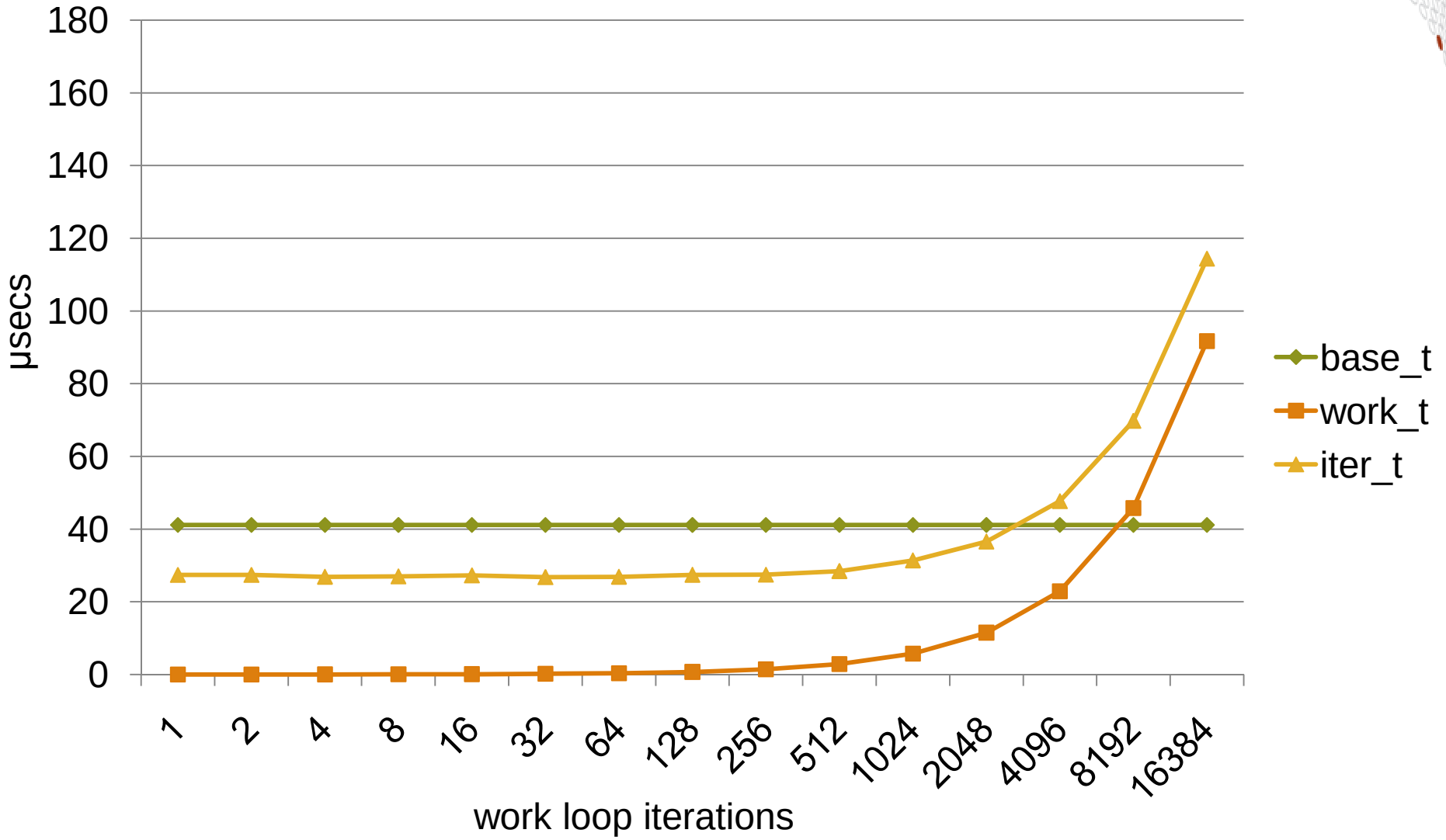
Measures application availability:

avail = 1 – (iter_t – work_t)/base_t

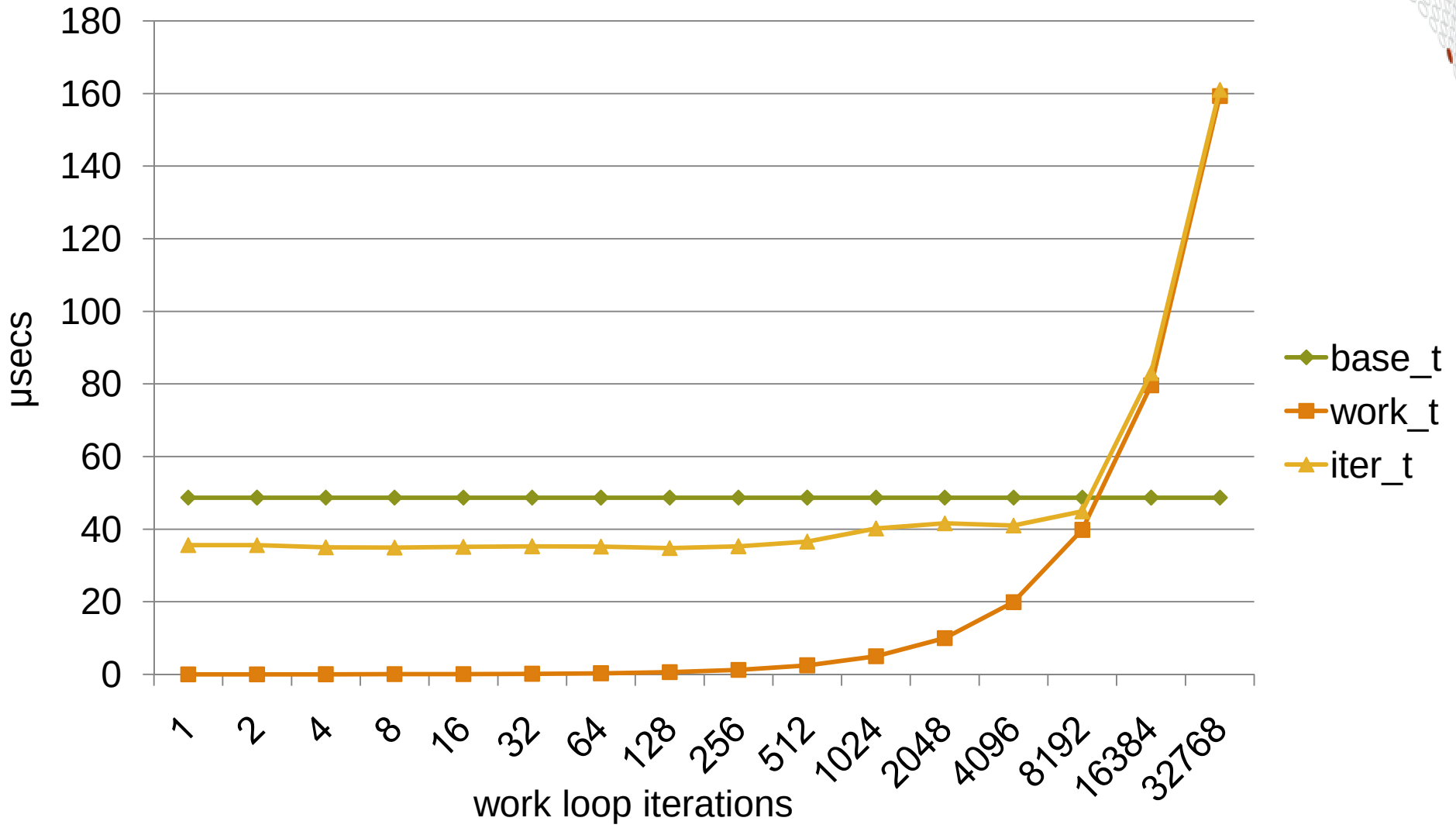
http://www.cs.sandia.gov/smb/index.html



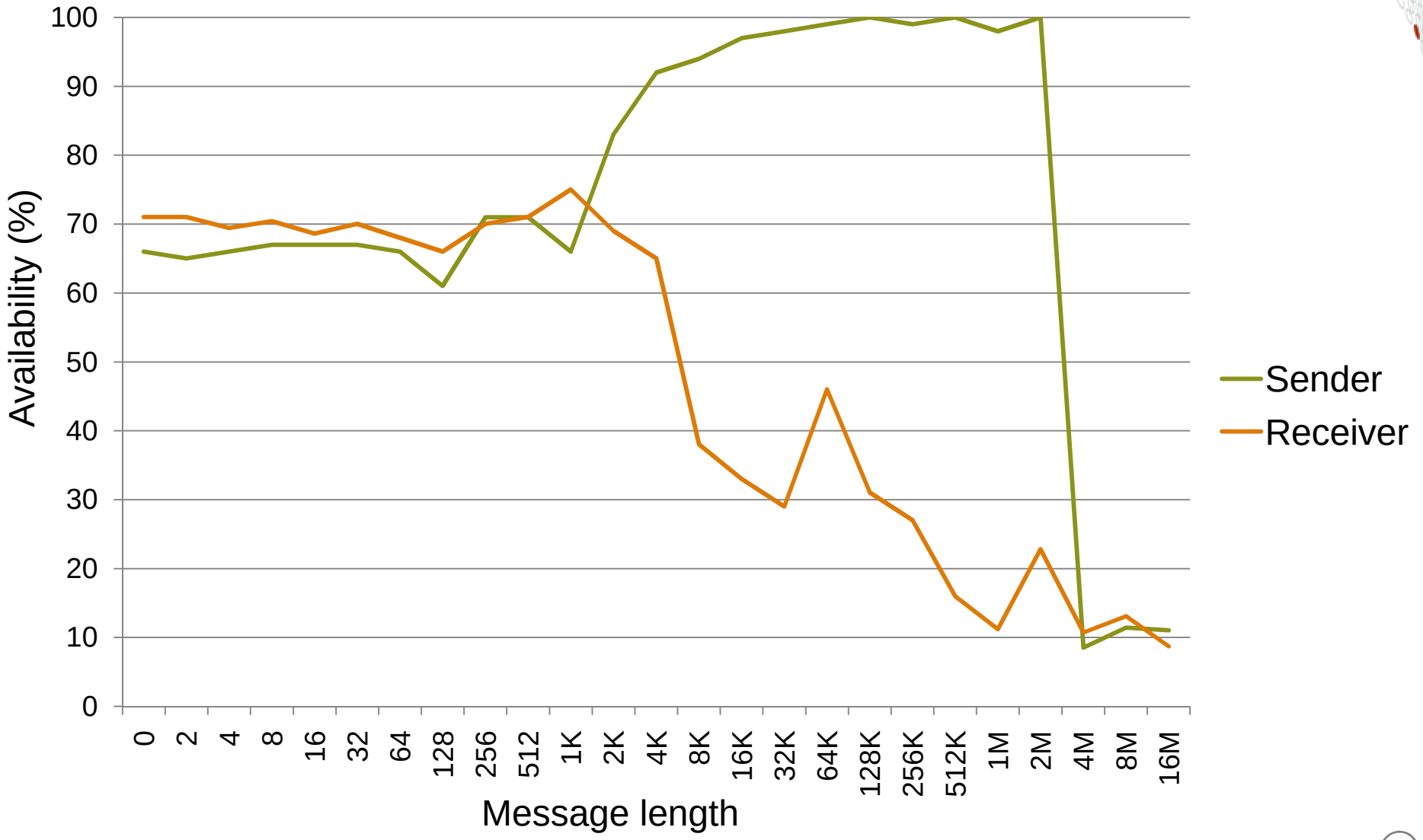
SMB – Receive side 64 KB message size (no progress)



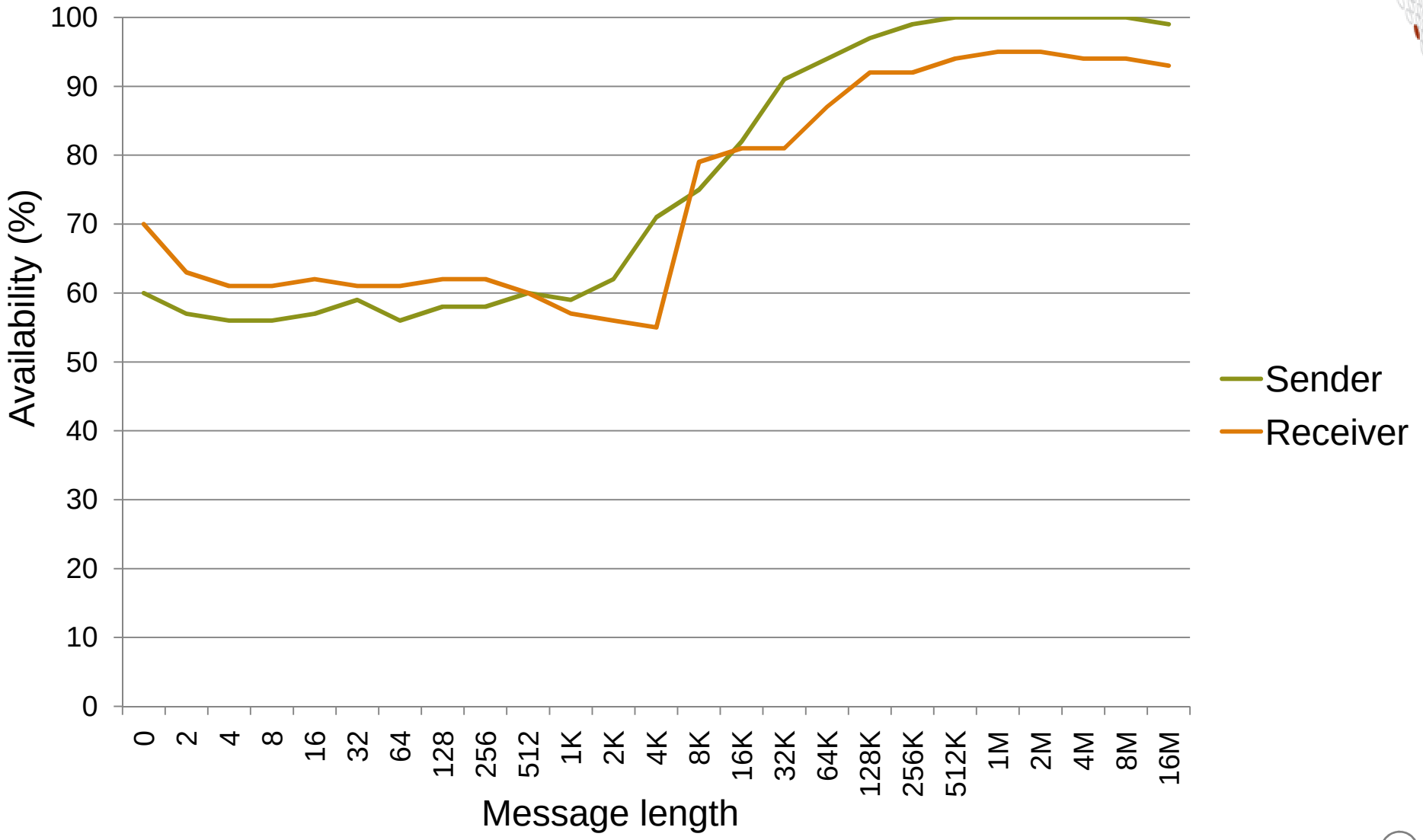
SMB – Receive side 64 KB message size (progress enabled)



Availability – no progress



Availability – progression enabled



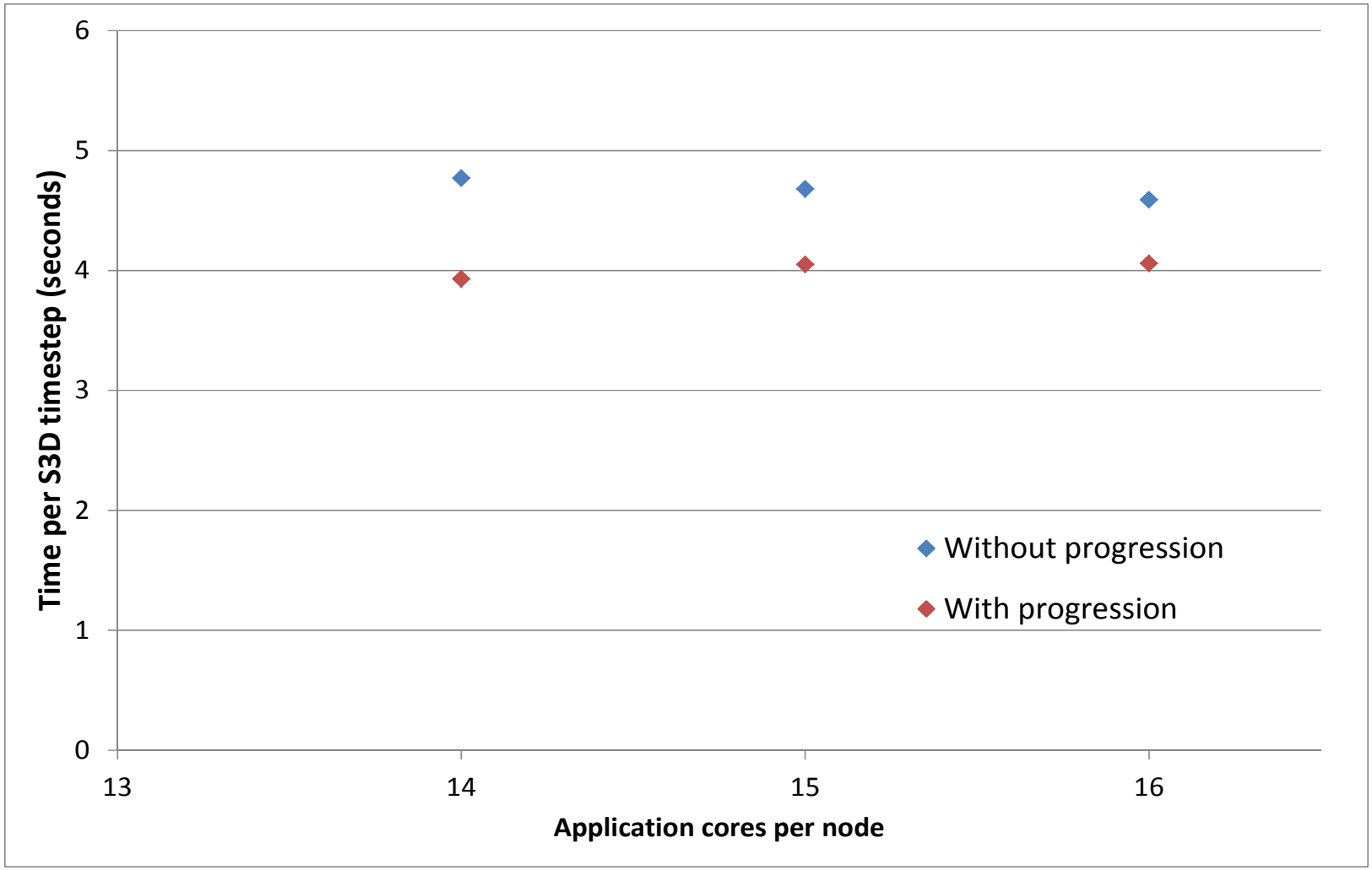
S3D Application

- Hybrid MPI/OpenMP
- Joint work by ORNL and Cray to prepare app for either using accelerators or OpenMP only on node
- Large messages for the data set considered

```
|-----------------------------------------------------
| Time%   | Time       | --  |   Calls |Total
|-----------------------------------------------------
100.0%    | 333.554284 | --  | 2022296 |Total
|-----------------------------------------------------
|  56.7%  | 189.059471 | --  |  605408 |USER
||====================================================
|  38.3%  | 127.905679 | --  |  869027 |MPI
||----------------------------------------------------
||  23.7% |  78.906595 | --  |   61200 |mpi_waitall_
||  11.8% |  39.426609 | --  |    6960 |mpi_wait_
||   2.4% |   7.933348 | --  |  399480 |mpi_isend_
||====================================================
|   4.1%  |  13.729663 | --  |  546009 |OMP
|…
```

MPI Msg Bytes	MPI Msg Count	< 16 Bytes	16 – 256B Bytes	256 – 4K Bytes	4K – 64K Bytes	64 K – 1M Bytes
22238303636	401311	1795	27	100806	2	298681

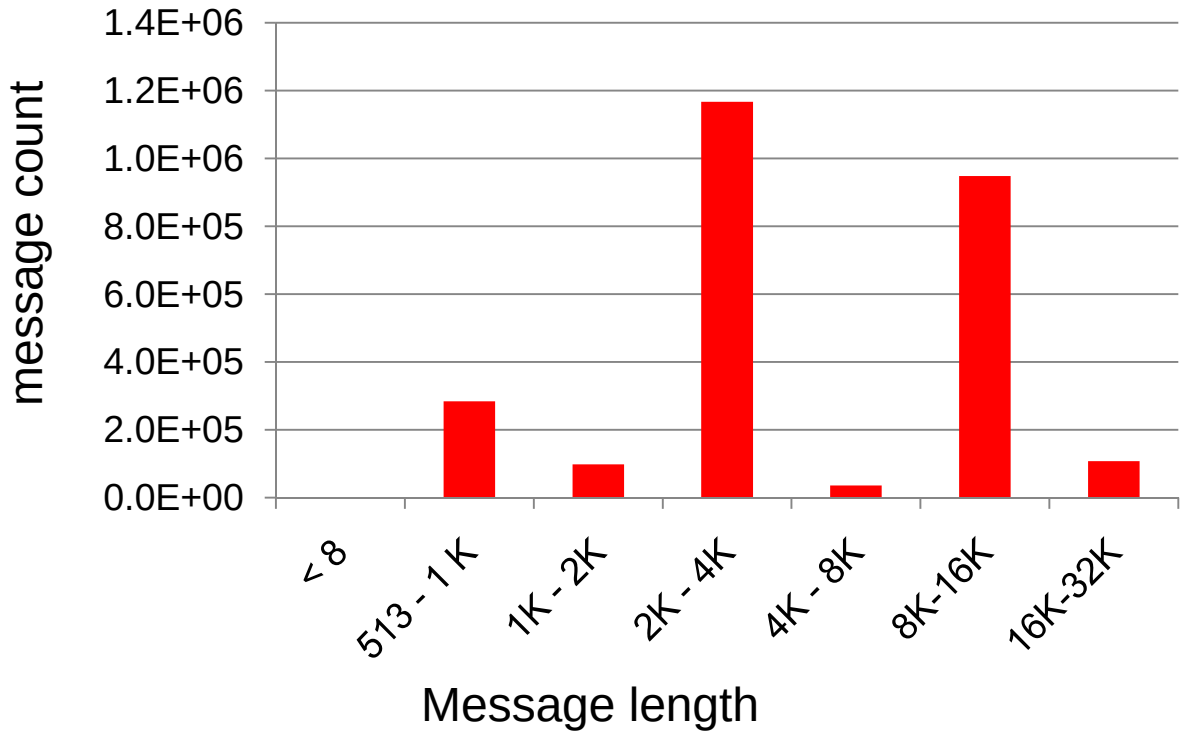
S3D Timings with and without Progression



Runs done on 64 XK nodes

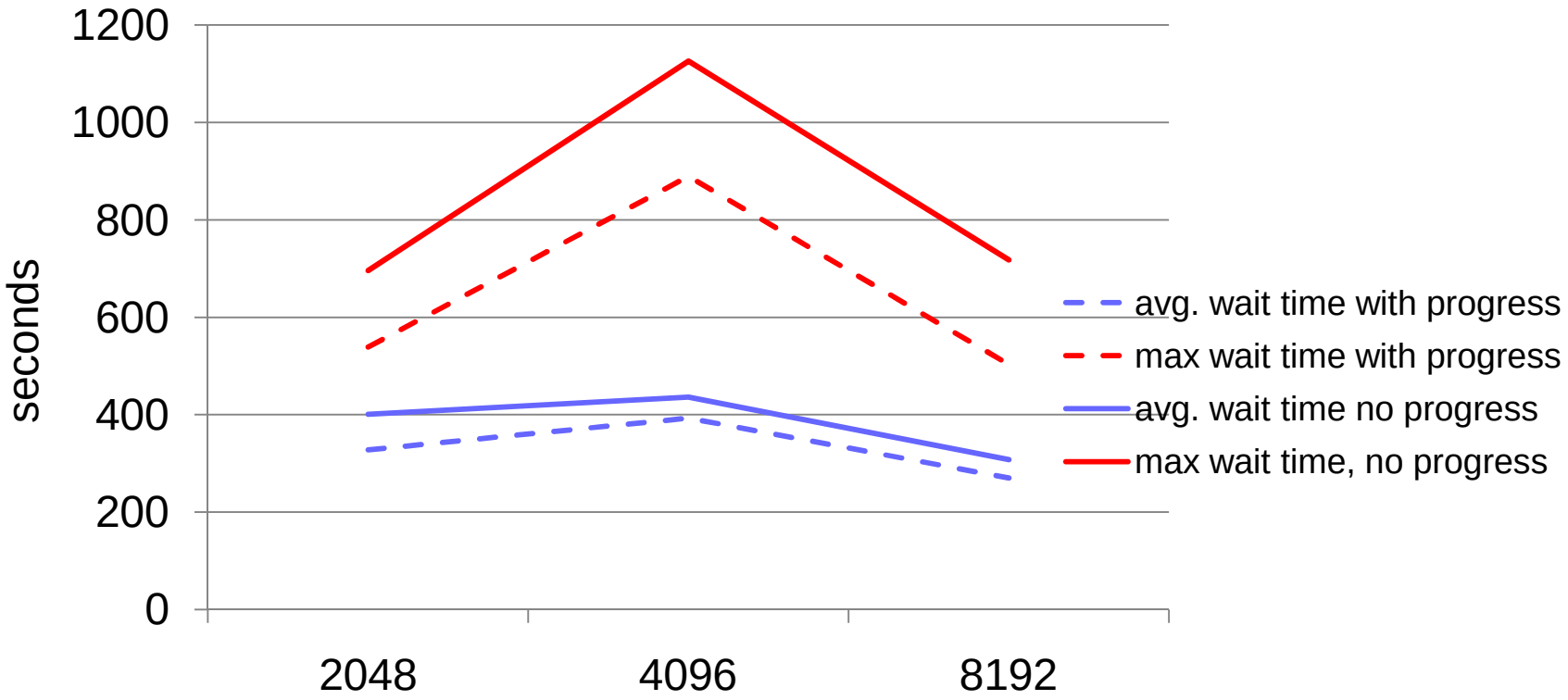
MILC Application

- Flat MPI application
- Message sizes more varied, both eager and rendezvous protocols are being used
- Significant amount of intra-node messaging



MILC Application – MPI_Wait time

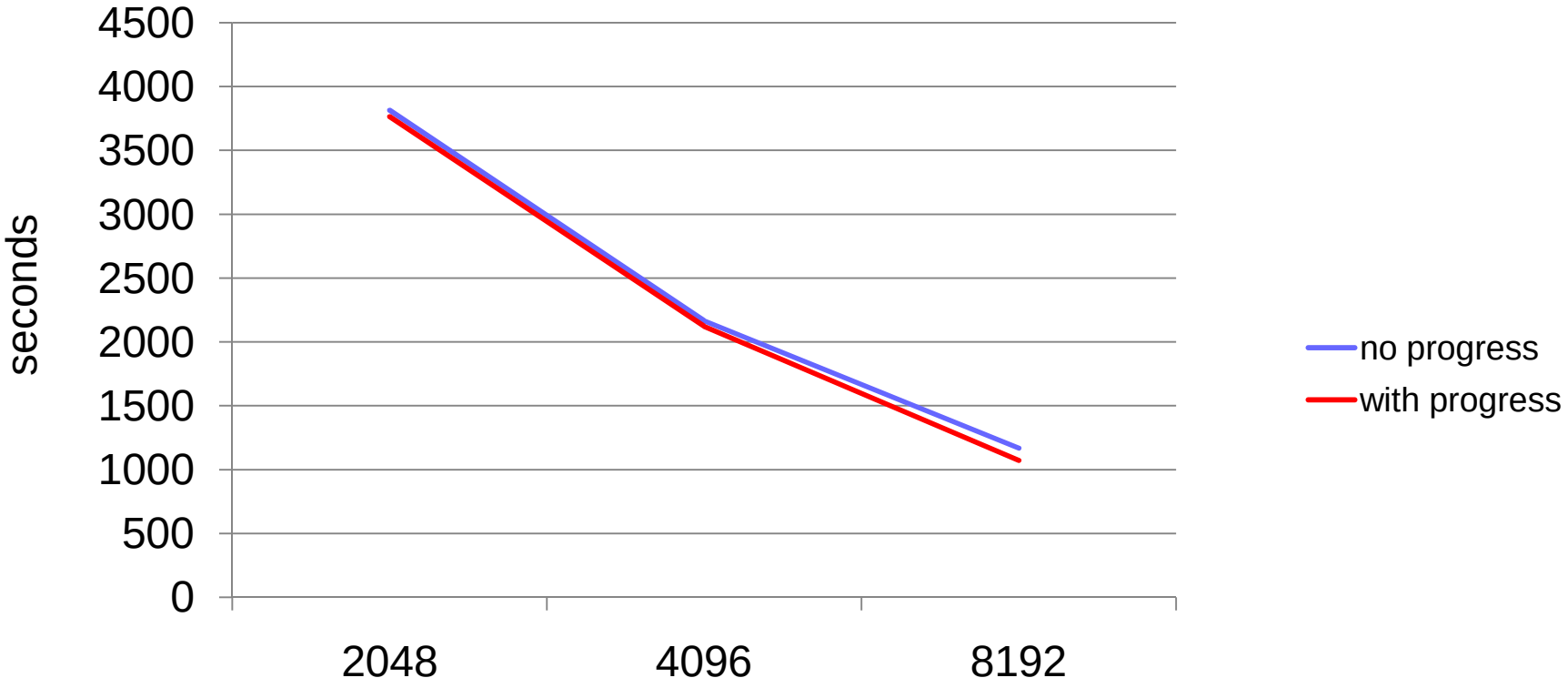
- App was modified to measure time spent in core gather/scatter communication scheme
- Non-blocking send/recvs used with some intervening work
- Max MPI_Wait time reduced significantly



2048 runs had to be run 16 ranks/per nod due to memory size

MILC Application – actual runtime

- The actual runtime changed much more modestly
- Load imbalance and MPI_Allreduce sync points may limit the actual speedup for this particular job type for MILC
  - ~8 % improvement at 8192 ranks
  - ~2 % improvement at 4096 ranks



Conclusions

- Thread-based method to progress does show some promise based on micro-benchmarks
- The current approach can be effective for applications with larger messages (at least 32 KB), but a mixture of messages using eager and rendezvous path limits the usefulness of the progress threads
- Not shown in the data here, but in the paper, that for flat MPI codes, CoreSpec is highly recommended when trying to use this thread-based progress mechanism

Future Work

- Enhance GNI interface to allow for more efficient use of the DMA (BTE) engine, allowing for better handling of shorter MPI messages (< 32KB)
- Use extensions to the CoreSpec/job package to schedule the progress threads on unused hyperthreads for next generation of Gemini interconnect
- Enhancements to GNI to more efficiently wake up progress threads
- Integrate the improved progress thread framework back into mainline Nemesis MPICH2 code

Acknowledgements

- **John Levesque and Steve Whalen for assistance with MILC and S3D.**

This material is based upon work supportedby the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357