



# **Modify An Application to Avoid Aries™ Network Congestion (S-0048-B)**

---

## Contents

Changes to This Document.....	3
Record of Revision.....	4
About Network Congestion Protection.....	5
Network Congestion Protection System Messages.....	6
Modify Code Communication Patterns to Avoid Network Congestion.....	7

---

## Changes to This Document

---

This update (S-0048-B) includes no new technical information; only editorial changes were made. Supports the 5.1.UP00 release of the Cray Linux Environment (CLE) operating system running on Cray systems.

---

## Record of Revision

---

S-0048-B Published April 2015 Supports the Cray Linux Environment (CLE) operating system 5.1.UP00 release running with the System Management Workstation (SMW) 7.1.UP00 release.

S-0048-A Published September 2013 Supports the Cray Linux Environment (CLE) operating system 5.1.UP00 release running with the System Management Workstation (SMW) 7.1.UP00 release.

---

## About Network Congestion Protection

---

In rare cases, there may be periods of intense communication between nodes that can cause network congestion. In extreme cases, this congestion degrades system performance and can cause system software to act to protect against such congestion.

Various conditions can cause network congestion; however, the likelihood of congestion caused by applications is highly correlated with the communication patterns used. For example, all-to-all transfers and synchronous access of shared global variables are two commonly identified communication patterns that can cause congestion, whereas MPI is less likely to do so.

The most direct way to solve problems of severe network congestion, throttling, or DMAPP API transient errors is to use the Cray performance analysis tools to find code that causes the troublesome communication patterns, and then to modify that code. For more information about using the Cray performance analysis tools, see *Cray Performance Measurement and Analysis Tools (S-2376)*.

---

## Network Congestion Protection System Messages

---

The system may start congestion protection software to throttle node bandwidth system wide. This is indicated by the following message in stderr when an application launched by aprun exits:

```
Application apid network throttled: nodecount nodes throttled, time node-seconds
```

This condition also produces a corresponding syslog message in `/var/opt/cray/log/pn-current/messages-YYYYMMDD` on the SMW:

```
date time nid aprun[pid]: apid=apid, Network throttled,user=uid, batch_id=unknown, nodes_throttled=nodecount, node_seconds=time
```

## Modify Code Communication Patterns to Avoid Network Congestion

The most direct way to solve problems of severe network congestion, throttling, or DMAPP API transient errors is to use the Cray performance analysis tools to find code that causes the troublesome communication patterns, and then to modify that code. The following examples address the most commonly identified communication patterns that can cause network congestion.

### All-to-all transfers

One way of implementing an all-to-all transfer pattern is by writing code similar to the following loop, in which the images of a Fortran 2008 program start by putting data on image 1, then on image 2, and so on. This loop concentrates all puts on one node, then on the next, and so on, in sequence. This can cause congestion and produce undesirable performance.

```
ti=this_image()
do ri=1, num_images()
  a((ti-1)*blksize+1:ti*blksize)[ri]=c((ri-1)*blksize+1:ri*blksize)enddo
```

A better method is to spread the targets by having each image begin with itself and wrap around. For example:

```
ti = this_image()
do rri=1,num_images()
  ri=mod((rri+ti,num_images()))+1
  a((ti-1)*blksize+1:ti*blksize)[ri]=c((ri-1)*blksize+1:ri*blksize)
enddo
```

When using a one-sided programming model, it is easy for a many-to-one transfer pattern to have the same effect as all-to-all. In particular, if many PEs all begin their many-to-one with the same initial target, then congestion can occur.

### Shared global variables

Another way to induce congestion is by writing code that enables all processing elements (PEs) to check the status of a global variable as quickly as possible. For example, this is UPC code in which all threads are loading the same globally shared variable.

```
volatile strict shared int spin = 0;
int main() {
  if ( MYTHREAD == 0 ) {
    printf("Waiting...\n");
    sleep( 50 );
    spin = 1; }
  else {
    unsigned long cnt;
    for( cnt = 0; spin != 1; ++cnt );
```

---

```
    printf("%d %lu\n",MYTHREAD,cnt); }
    return 0;}
```

In this case, it is better for each PE to spin-wait on a local variable and wait for that local variable to be changed. Alternatively, you can use UPC or Fortran 2008 locks, which are available in the respective languages and implemented using scalable algorithms that do not cause contention issues. Another option is to use the locks implemented with the DMAPP API. For more information on locking mechanisms found in DMAPP, see *GNI and DMAPP APIs (S-2446)*.

### SHMEM atomic compare-and-swap

Another example of code that may cause network congestion is the following loop, written using the SHMEM atomic compare-and-swap function, and executed simultaneously by all PEs in the job.

```
#include <stdio.h>
#include <mpp/shmem.h>
#include <unistd.h>

long spin = 0;

int main() {
    long spin_res = 0;
    unsigned long cnt = 0;
    int mype;

    shmem_init();
    mype = shmem_my_pe();

    if ( mype == 0 ) {
        printf("Waiting...\n");
        sleep( 50 );
        spin = 1;
    } else {
        while (spin_res == 0) {
            spin_res = shmem_long_cswap(&spin, 1L, 1L, 0);
        }
    }
    shmem_finalize();
    return 0;
}
```

It would be better for each PE to spin-wait on a local variable, and best to use UPC, Fortran 2008, or SHMEM locks.