*Final Report by Kelly Johnson, Bernard Lee, and Peter Leong*

## Section 0: Table of Contents

Section 1. ***Introduction*** — ***Page 2***

Section 2. ***Overall Design*** — ***Page 3***

Section 3. ***Case Studies*** — ***Page 9***

Section 4. ***Implementation Technicalities*** — ***Page 25***

Section 5. ***Performanace Analysis*** — ***Page 39***

Section 6. ***Conclusion*** — ***Page 54***

*Copyright 1992 Stanford University.*

# Section 1. Introduction

## Abstract

Parallel computing has been evolving toward two major programming paradigms. In the future, there will be an inevitable marriage between shared–memory and message–passing schemes. One may advance toward that goal via two different means: first, through the development of a message–passing mechanism on an existing shared–memory architecture; or, second, through the emulation of shared–memory ‘‘behavior’’ on an existing message–passing architecture. Such a goal is attained when both paradigms can be blended together and effectively supported, when it is possible to apply either paradigm on the same architecture, and when there are no longer any immediate distinctions between the two paradigms. This project attempts to advance toward that goal via the first approach –– i.e., by implementing the iPSC/860 message–passing library on the Stanford DASH prototype.

## Objectives

The authors' intention has been to implement the essence of the iPSC/860 message–passing library on the Stanford DASH prototype, such that the majority of programs that run on the iPSC/860 can be exported to the Stanford DASH prototype with a minimum amount of modifications. Such an effort serves to illustrate the feasibility of implementing a message–passing scheme on a shared–memory architecture, so that:

**1)** *There is a quantitative means for researchers to compare the effectiveness of both architectures;*

**2)** *Researchers can readily identify and integrate the merits of either architecture.*

In addition, the capability of executing the same pieces of code on two different architectures allows scientific researchers to access the combined set of code no matter which architecture the codes were originally developed for.

# Section 2. Overall Design

## Design Philosophy

The fundamental philosophy in the current design is to implement the library at such a level of transparency that the user should not notice any major difference between programming the iPSC/860 and the DASH machine. Unfortunately, total transparency can only be achieved at the compiler level. The current interface does require the users to make a minimum amount of modifications in their iPSC/860 applications. The host modules should begin in the following manner:

```
/* This is the host */
    #include "ipsc_lib.h"
    main()
    {
          host_init();
          ....
    }
```

and each of the node modules should begin in the following manner:

```
/* This is a node */
    #include "ipsc_lib.h"
    main()
    {
          node_init();
          ....
    }
```

Finally, the compiled modules should be linked to the `ipsc_lib.a` library as in:

```
 # This is the Makefile for an iPSC/860 application
    IPSCLIB = -lipsc_lib.a
    IPSCLIBPATH = -L/usr/local/lib
    ${CC} ${CFLAGS} -o ${PROG} ${PROG}.o ${IPSCLIBPATH} ${IPSCLIB}
```

## System Architecture

The Client–Server model, similar to the one adopted by the X–Window system initiated by MIT, is used by the iPSC/860 emulation library. The MIT experience suggests to the authors that this model is desirable due to the following reasons:

**1)** *Despite its level of complexity, the client–server model has proven to be highly successful when it is necessary to insulate the user module completely from the system module, and when the system module can be reasonably expected to absorb a certain amount of "abuse" from innocent users with little experience in system programming;*

**2)** *From a development point of view, this model provides an efficient mechanism for literally hundreds of programmers to contribute to future versions of the library. Moreover, if the current project has proven to be successful, there is every reason to believe that vendors of non–SGI based shared–memory architectures will try to follow the paradigm. In other words, this model prevents front–end users from being burdened by future changes made by system programmers in the back–end, thereby giving the back–end designers plenty of freedom for maneuvering toward a future upgrade.*

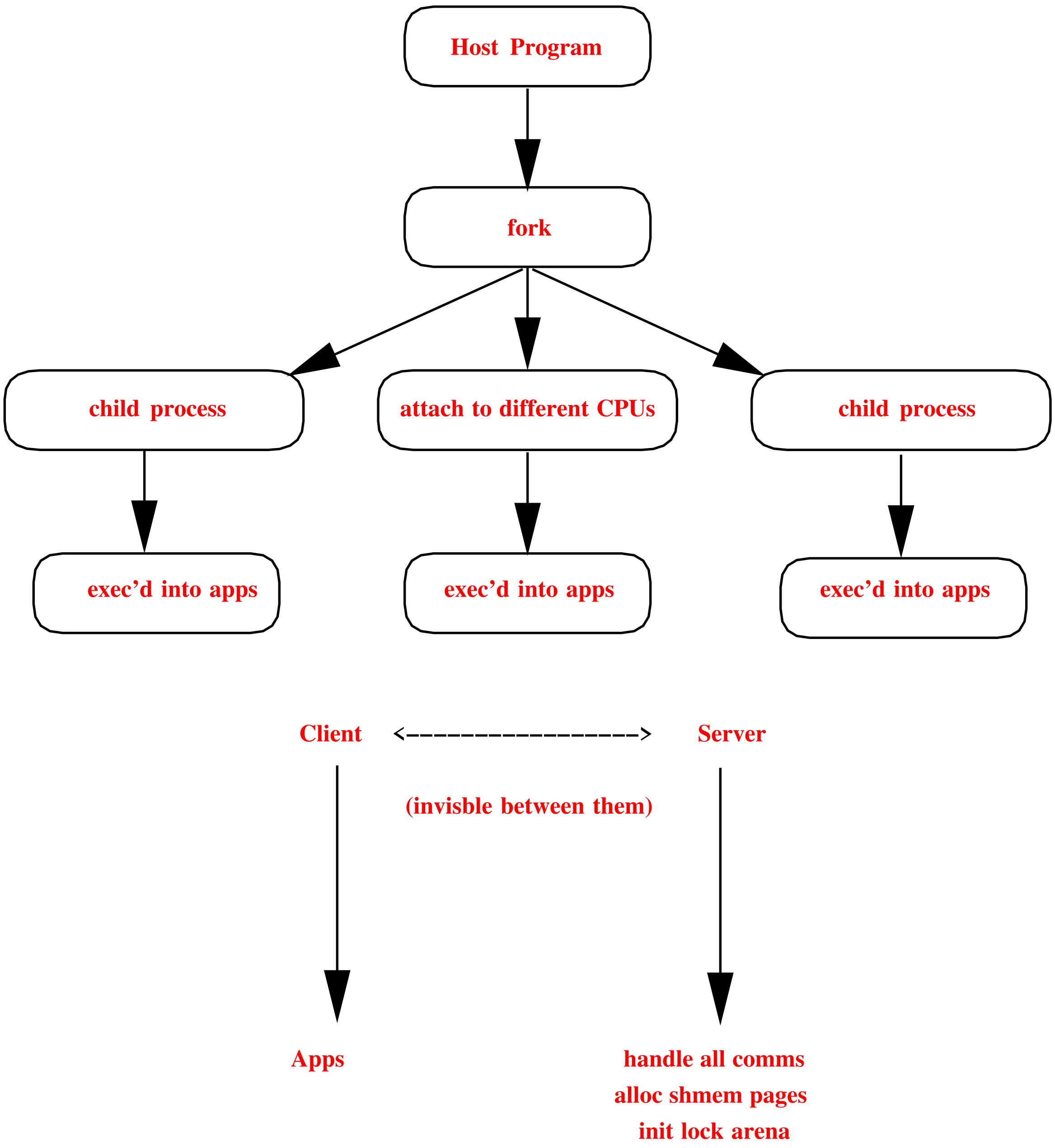
**Figure 1**

Figure 1 presents a brief overview on the implementation of this client–server model. The host program forks into the specified number of children processes, each of which is attached to a different CPU. Upon successful completion, the children processes `exec()` into a node program. Notice that the children processes become entirely independent of their parent process upon the `exec()` call. No visibility whatsoever is allowed between the client (the node programs) and the server (the communication primitives). Upon the completion of each service request, the client will only receive responses as specified by the arguments.

**Figure 2**

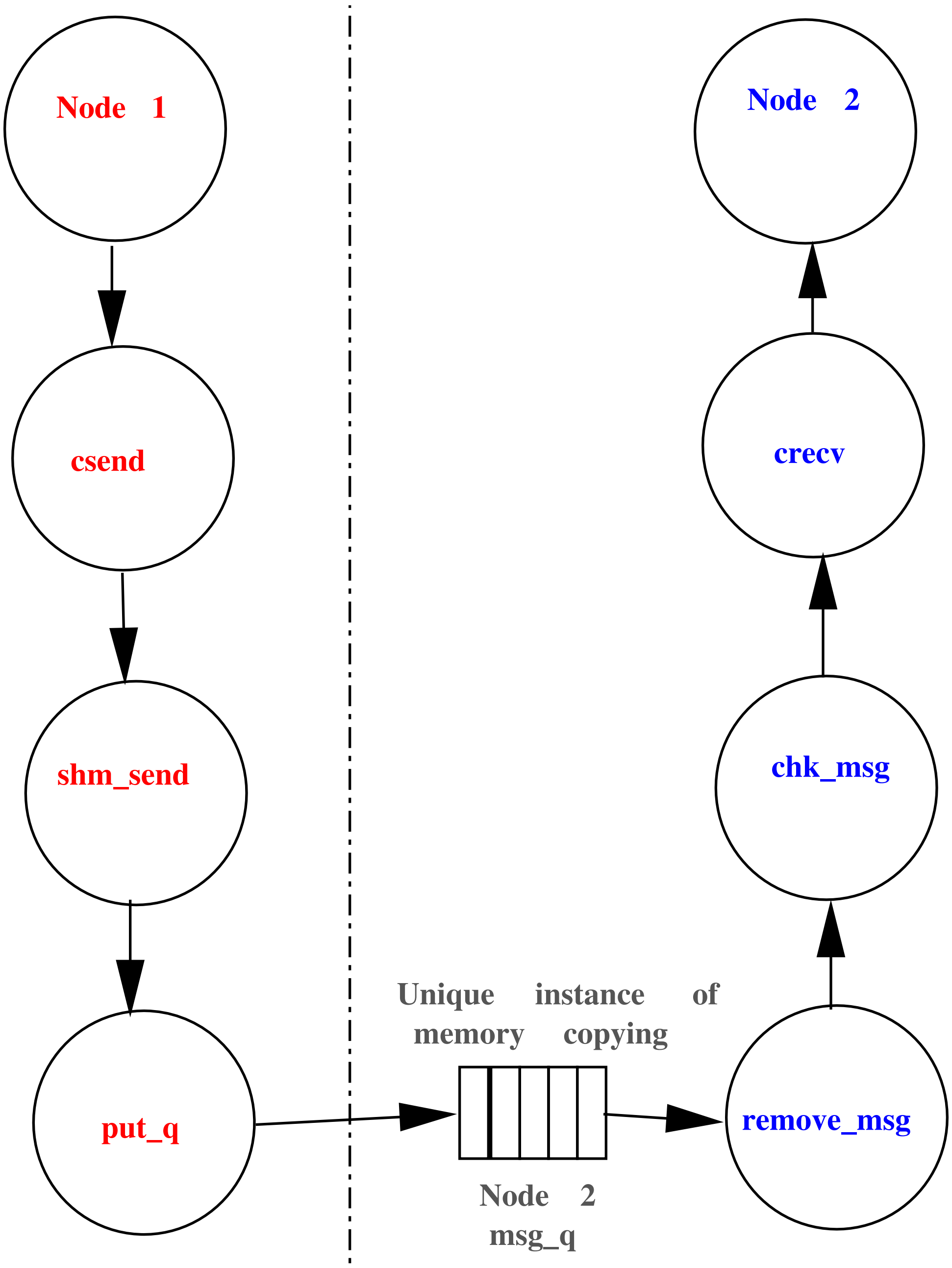
Figure 2 presents a brief overview of the data–flow hierarchies between two nodes. Notice that the memory is copied twice throughout the sequence despite the number of hierarchies that each message must travel through.

## Figure 1: Client–Server Model




Copyright 1992 Stanford University.

## Figure 2: csend and crecv Data–Flow Diagram




*Copyright 1992 Stanford University.*

## Pragmatic Design Adjustments

Due to salient time constraints, the authors have chosen to modify several minor details in their original design to accommodate the pragmatic goals of bringing a functional version of the iPSC/860 simulation library to DASH. It is hoped that these *ad hoc* measures can be replaced by permanent and more efficient implementations as the DASH hardware and operating system mature and stabilize. Moreover, many of the current implementations can be made more efficient if it is not necessary to conform strictly to the iPSC/860 protocol. It is hoped that these implementations can eventually be made in the simulation library.

### BSD Signals

The current implementations of `hsend()`, `hrecv()` rely on BSD signals to facilitate inter–processor communication. For instance, if processor `0` wishes to communicate with processor 1, then processor `0` would fill in a wait–buffer previously allocated by processor `1`, and then signal processor `1` to service that buffer. The net effect is that the UNIX kernel must be swapped in three times before such a transaction can be completed –– once to send out the signal for processor `0`, once to receive the signal for processor `1`, and, finally, once to restore processor `1` from the exception handling context. Since context–switching in UNIX is very expensive, such a scheme is undesirable from an efficiency point of view. On the other hand, a socket–based implementation can complete the transaction with a single UNIX context switch –– the operating system can copy the message for the sending processor and generates the signal simultaneously, and then the receiving processor will perform a `select()` on its socket at its earliest convenience. This scheme will be more efficient as long as the operating system is capable of localizing the socket buffers on each cluster. However, that cannot be accomplished without possibly rebuilding part of the system kernel.

### Pre–fetching

The necessary `gm` data structures can be pre–fetched at the beginning of each routine. At the present time, however, extensive usage of pre–fetching is causing numerous kernel panics on the DASH machine. As a result, this possibility has not been fully explored.

### Static Buffers, Garbage Collection and Malloc() Locks

By using statically pre–allocated message buffers, the iPSC/860 exhibits significant performance gains in communicating messages that are shorter than 100 bytes in length. Such an advantage has yet to be exploited by the simulation library. Moreover, the current version of the simulation library does not perform garbage collection on the shared memory heaps, which is expected to become a serious problem once the testing of the library extends to "industrial" applications which typically consumes voluminous amount of memories. Finally, the current implementation only supports one global lock for the different free lists in all the clusters. That is an obvious flaw in this implemenation and it can be made more efficient as the `gm` data structure is gradually broken down into cluster–specific data structures where possible.

### Micro–kernel

The authors had hoped to implement the simulation library within a micro–kernel environment that is invisible to the user. However, that plan was abandoned when it was clear that such a scheme could only be implemented fully at the compiler level. The major problem is that the authors have experienced persistent difficulties in making the data segments on each cluster visible to all processes without running the high risk of accidental overwrites. On the other hand, the compiler can award visibility privileges exclusively to the micro–kernel. Unfortunately, such a level of complexity is not realistic within the time frame of this project.

# Section 3. Case Studies

## Acknowledgements

The authors are grateful to the following organizations or individuals for supplying test cases that are crucial to the testing of the simulation library:

**BB -- A Combinatoric Optimization Problem Solver**
Although the present version of this program is written by two of the authors, Johnson and Lee, Eirik Wilson of LSI Logic has made significant contributions to earlier versions of this program.

**PFLOW -- A Computational Fluid Dynamics Package**
This package is supplied by Andrew Anagnost of the Stanford Aeronautics and Astronautics Department and also of the NASA Ames Research Center.

**PSRS -- A Parallel Sorting by Random Sampling Algorithm**
This algorithm is supplied by a Canadian group of computer scientists, namely, Xiaobo Li, Pual Lu, Jonathan Schaeffer, John Schillington and Pok Sze Wong of the University of Alberta Computer Science Department, as well as Hanmao Shi of the Unviersity of Waterloo Computer Science Department.

**RENDER -- A Volume Rendering Program**
This program is supplied by Jason Nieh of the Stanford Electrical Engineering Department.

## BB –– A Combinatoric Optimization Problem Solver

A program that solves the traveling salesman problem using the branch and bound strategy was chosen for the initial testing of the `iPSC/860` emulation library. The decision to use the traveling salesman problem was made because the working version of the code already exists for both the iPSC/860 and a shared memory machine (Encore Multimax), with speed–up data available for comparison. To modify the iPSC/860 code to run on DASH, it is only necessary to call the `host_init()` and `node_init()` subroutines, include the proper header files, and then rebuild the executables. These initial results show that the iPSC/860 library on DASH is capable of achieving speedups similar to a real iPSC/860 machine.

## Chart 1: Total Speed–ups for the 11–City Case

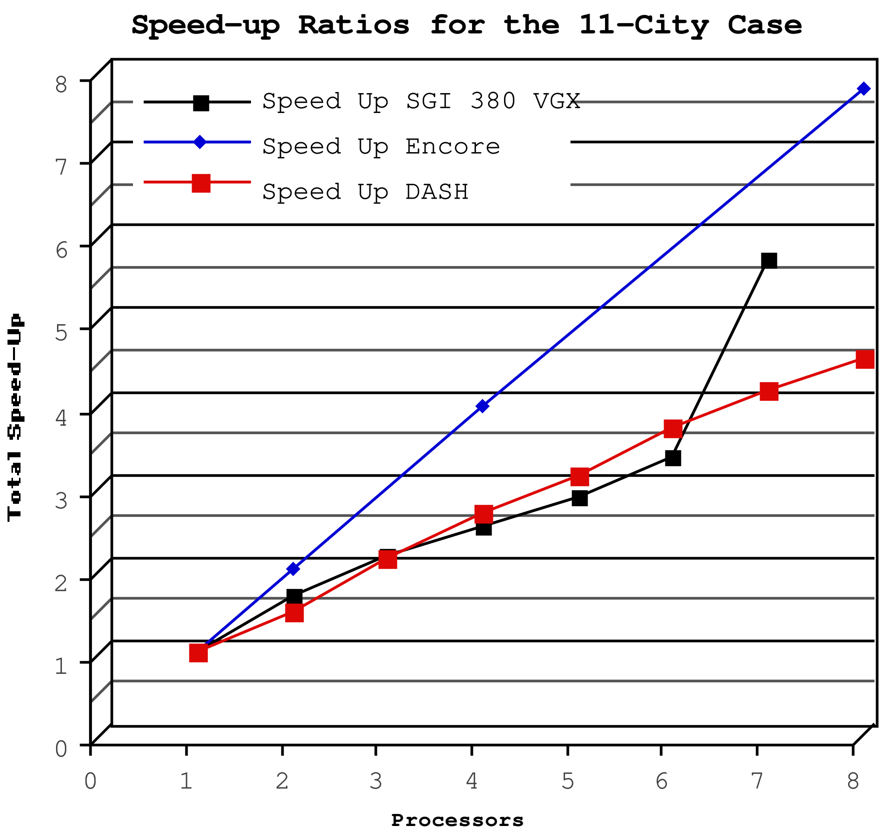
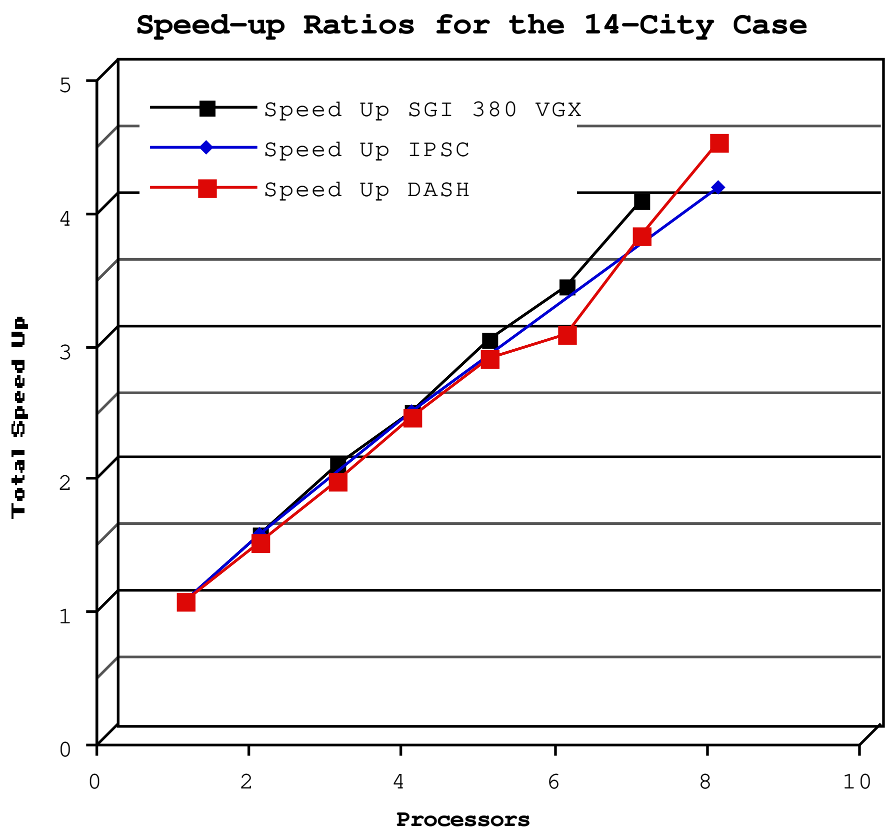



Chart 1 shows the speedup comparison between the DASH version of the iPSC code and the code running on the Encore Multimax for the 11–city problem. While the Encore speedup is almost linear, the DASH speed–up is less than ideal. This can partly be attributed to the differences between the shared–memory and message passing code. The message–passing code uses a slightly outdated bound during its search while the shared–memory code always uses the most current bound. Because of this difference in the algorithms, proper caution should be exercised in interpreting the implications of these results. Nevertheless, it is somewhat helpful to compare similar algorithms implemented on different architectures.

*Copyright 1992 Stanford University.*

## Chart 2: Total Speed–ups for the 14–City Case




The most interesting thing in Chart 2 is the comparison between the same code running on the iPSC/860 and on the DASH machine. Chart 2 shows the speedup comparison between the iPSC/860 and DASH for the 14–city problem. The speedup data on the DASH is very close to the results from the iPSC/860, and, as the number of processors is increased, even slightly better speedup ratios have been achieved on the DASH machine. Also included is the speed–up data for the SGI 380 VGX, where the initial testing of the library was performed.

*Copyright 1992 Stanford University.*

## Chart 3  Total Run Time for the 14–City Case

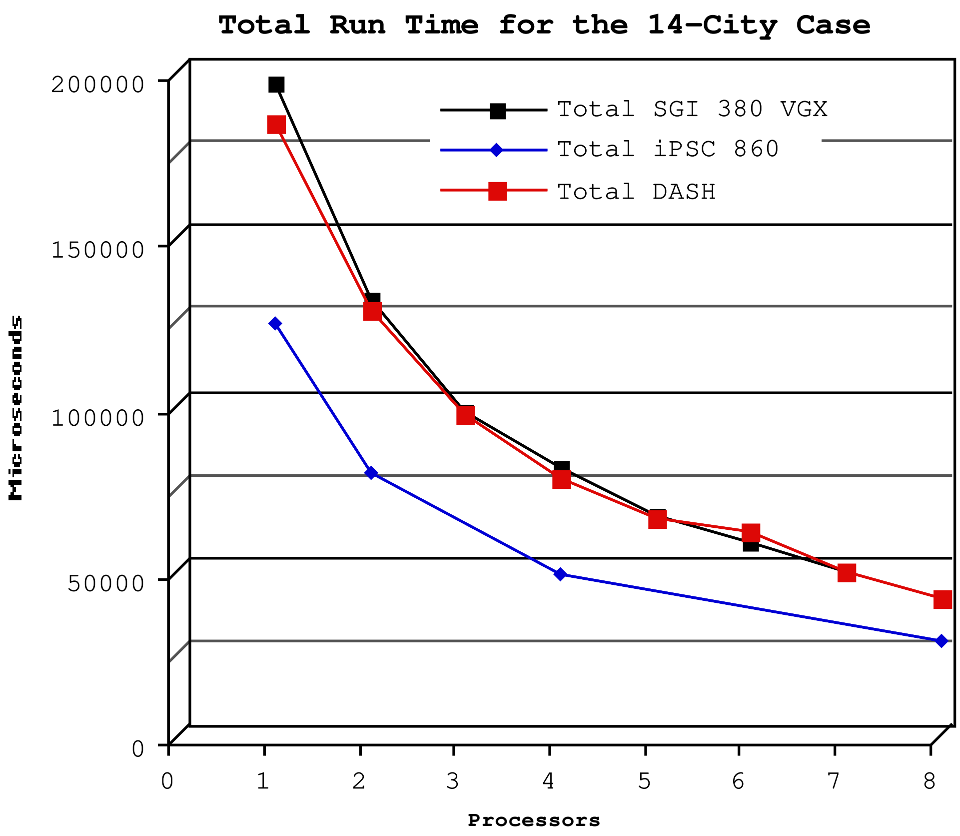
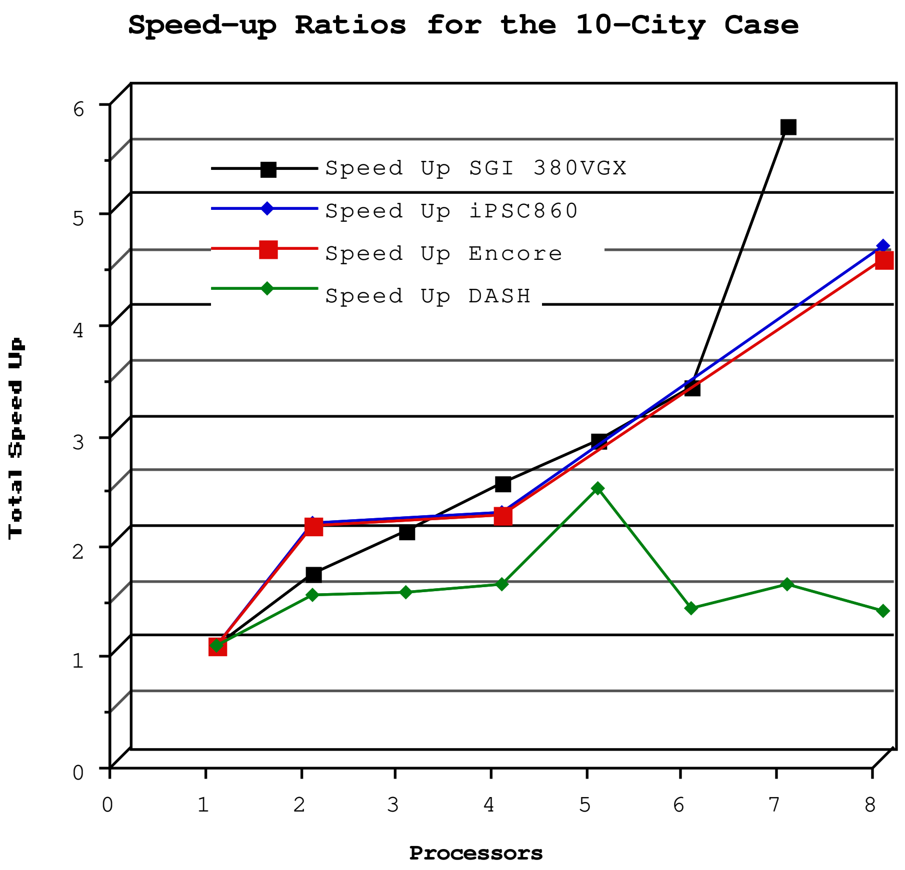



Chart 3 shows the comparison of the absolute runtime between the same code running on iPSC 860, the ipsc library running on DASH, and the ipsc library running on SGI 380 VGX. For a large problem such as the 14–city case, more time is spent in computation, and the greater computing power of the i860 processor may account for the better runtime.

## *Chart 4: Total Speed–ups for the 10–City Case*




The same observation can be made for Chart 3, which shows the speed–up comparisons among the message–passing code running on the `iPSC/860`, `DASH`, `SGI 380 VGX`, and the shared memory version running on the Encore Multimax for the 10–city problem. The message–passing code achieves the same speed–up as the Encore shared memory code in this case, contrary to the conclusion that one might draw from Chart 1. The large difference between `SGI 380 VGX` and `DASH` is troubling and warrants further investigation.

## Chart 5: Total Run Time for the 10–City Case

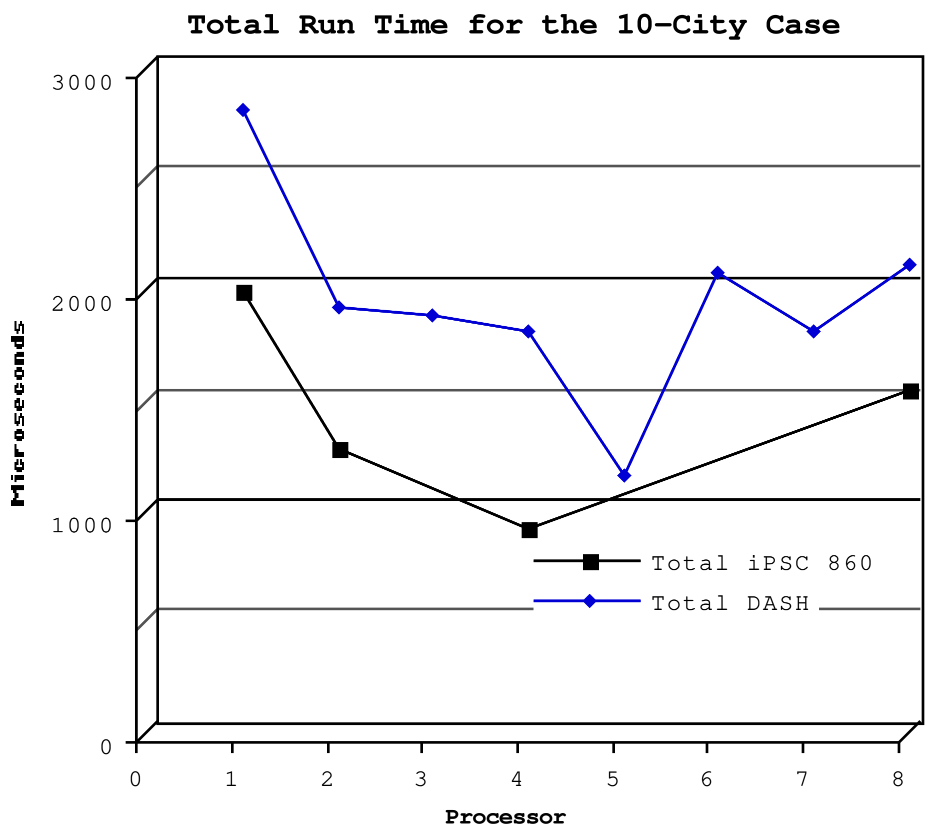



Chart 5 shows the absolute runtime comparison of the iPSC/860 and DASH running the traveling salesman for the 10–city case. Again, the performance of the library running on DASH for more than 4 processors needs further improvement.

## PFLOW –– A Computational Fluid Dynamics Package

`PFLOW` is a fairly straight–forward `CFD` program. It starts by reading in a data grid and simulates the dynamics of fluid flow around the grid object for a fixed number of time–steps.

`PFLOW` has been instrumental in helping the authors to debug those library functions designed for asynchronous messages, such as `isend()`, `irecv()`, `msgwait()`, and `msgdone()`. Limited performance data was collected on `PFLOW` due to time constraints.

*Copyright 1992 Stanford University.*

## Chart 6: PFLOW Total Speed–ups

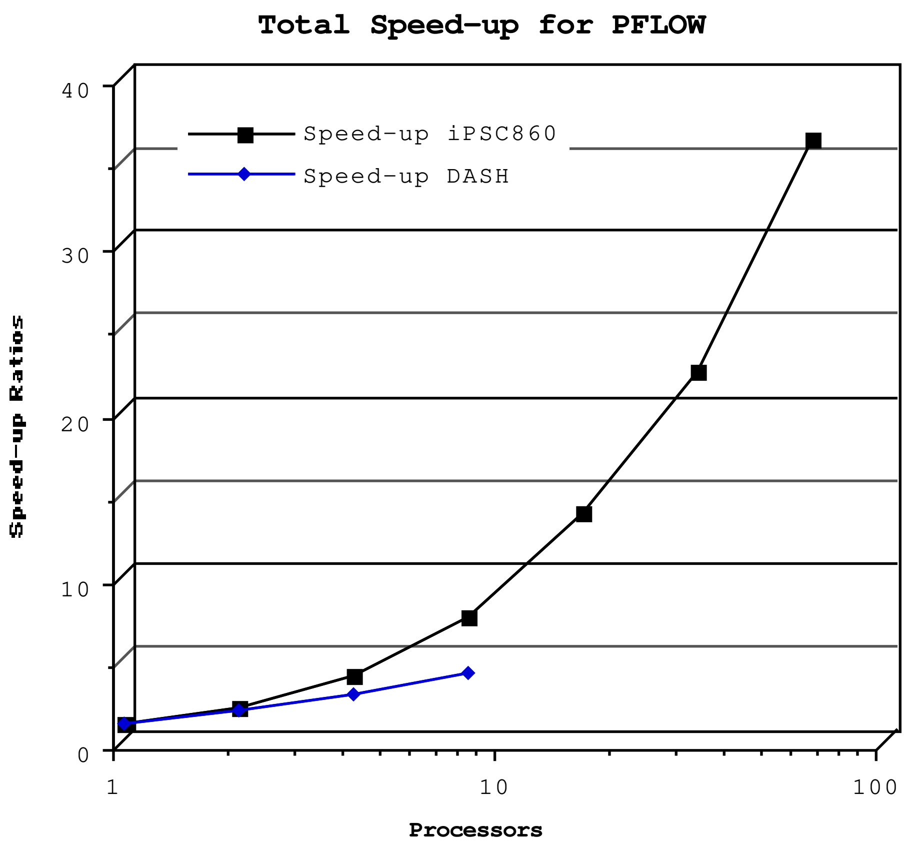



Chart 6 shows the speed–up comparison for PFLOW running on the iPSC/860 and on DASH. For lower number of processors, DASH's speed–up tracks the iPSC/860 speed–up reasonably well. Data for DASH with more than 8 processors is not available at this time.

## Chart 7: PFLOW Total Run Time

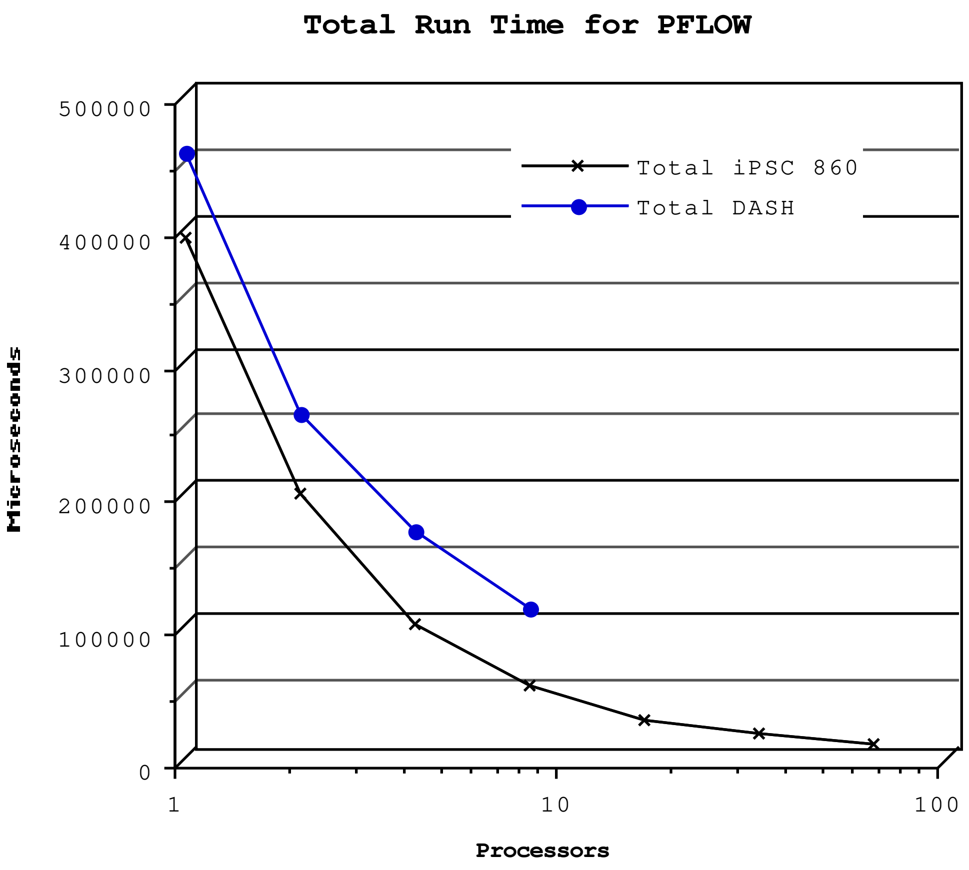



Chart 7 shows the absolute running times of PFLOW. Again, the results are reasonably close except that current results for more than 8 CPUs on DASH are not available.

## *PSRS –– Parallel Sorting by Regular Sampling Algorithm*

`PSRS` is a partition–based parallel sorting algorithm. It attempts to partition the data evenly among all processors by using a regular sampling of data to achieve ideal load balancing. Initially, the data is partitioned evenly among all processors and sorted within each processor using a sequential quicksort algorithm. On a machine with `p` processors, `(p–1)` regularly spaced samples are chosen from each processor. After this set of `p*(p–1)` items are sorted, `(p–1)` regularly space items are picked as the pivot points for the partitioning of the entire data set. These pivot points in turn divide the data into `p` sets, one for each processor, and every processor sends a sublist to every other processor and performs a merge on the `p` sorted sublists that it received. After the receive is complete, all processors synchronize and the entire data set is sorted.

In practice, regular sampling enables `PSRS` to achieve near ideal load balancing most of the time. The complexity of this algorithm has been shown to be `O((n * log n)/ p)` for `n` data items and `p` processors, making it one of the faster parallel sorting algorithms in existence.

The performance data for `PSRS` is obtained by generating a random data set, sorting it, and repeating the process five times.

*Copyright 1992 Stanford University.*

## Chart 8: PSRS Total Speed–ups

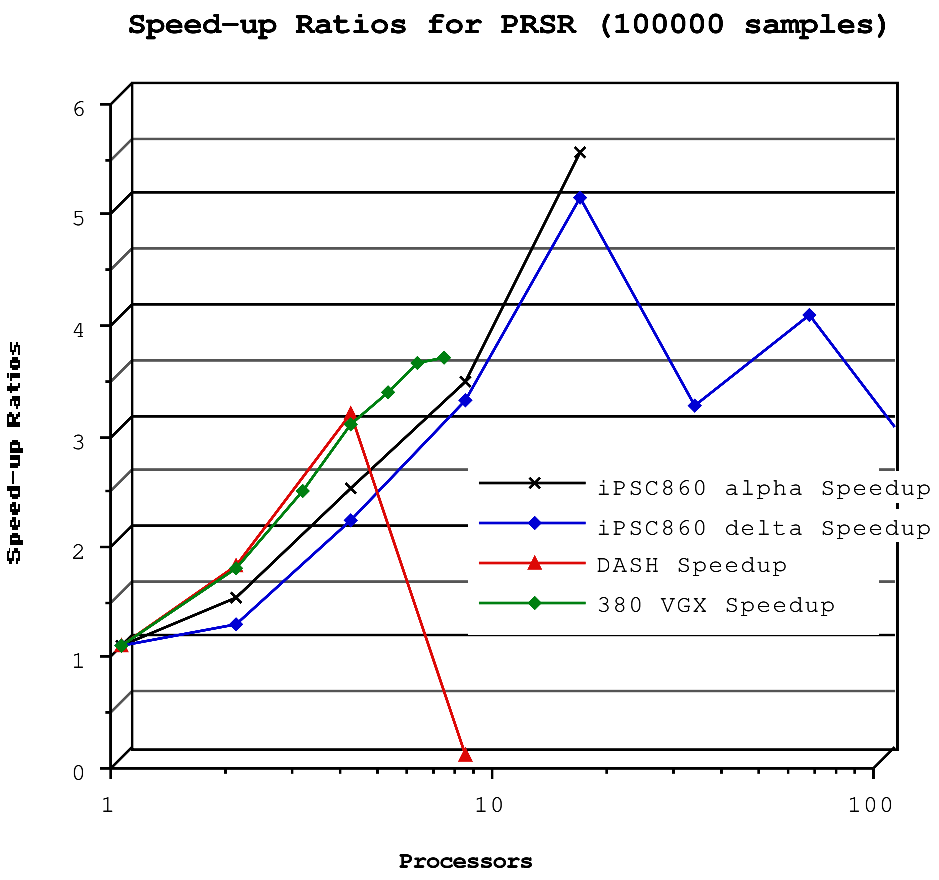



Chart 8 gives the speedup comparisons of PSRS for a sample size of 100,000. The poor performance of DASH is troubling, since it is expected to be at least as good as SGI 380 VGX. This is probably due to the initial difficulties of bring up the library on DASH and will hopefully be solved when the rest of the performance bugs are worked out. In all cases, the performance drops after 16 processors, reaffirming the belief that the initial data splitting limits the speedup to a maximum of about 5 or 6 regardless of how many CPU's are used.

## Chart 9: PSRS Total Run Time

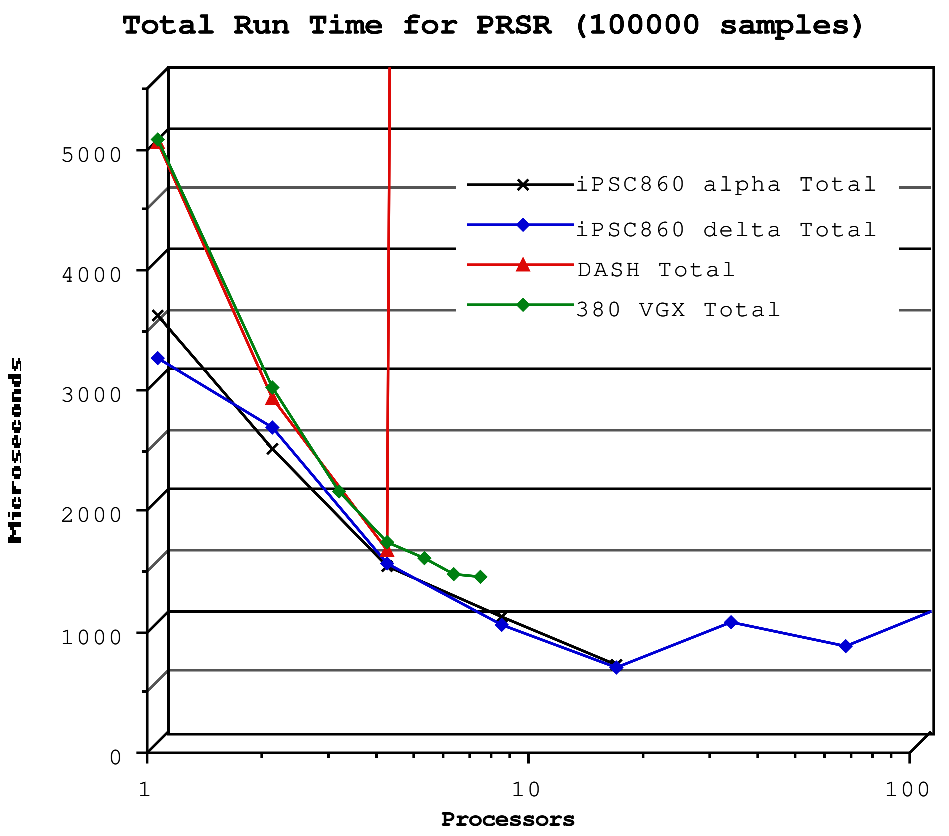



Chart 9 shows the total runtime of `PSRS`. Here we notice that the runtime for `DASH` is on–par with the rest of the machines, suggesting that there are bottlenecks in the current implementation of the library that limits scalability.

## RENDER –– A Volume Rendering Program

RENDER is a parallel volume rendering program for visualizing three–dimensional sampled scalar fields. This program partitions the data set among all the processors, and then performs adaptive ray–tracing along with a number of other optimizations. The algorithm has been written for both the iPSC 860 and the DASH multiprocessor. At the time of this report the performance results for the code on DASH are not yet available. However, it will be interesting to compare these results to the results of iPSC code running on DASH.

Render first reads in the data descriptions and computes the normal, opacity, and octree for each voxel once it is used by all nodes. Normally, each processor starts to render its assigned region by ray–tracing. A series of frames is generated. After the completion of each frame, one of the nodes sends the image back to the host processor.

Aside from the initial computation, RENDER spends most of its time in either ray–tracing or waiting for the send to the host to complete. The next two charts will characterize the performances in terms of these two categories. In all cases performance results are based on generating a 209x209 image using a 128x128x113 CAT–SCAN data–set of a skull viewed from above.

## *Chart 10: RENDER Total Speed–ups*

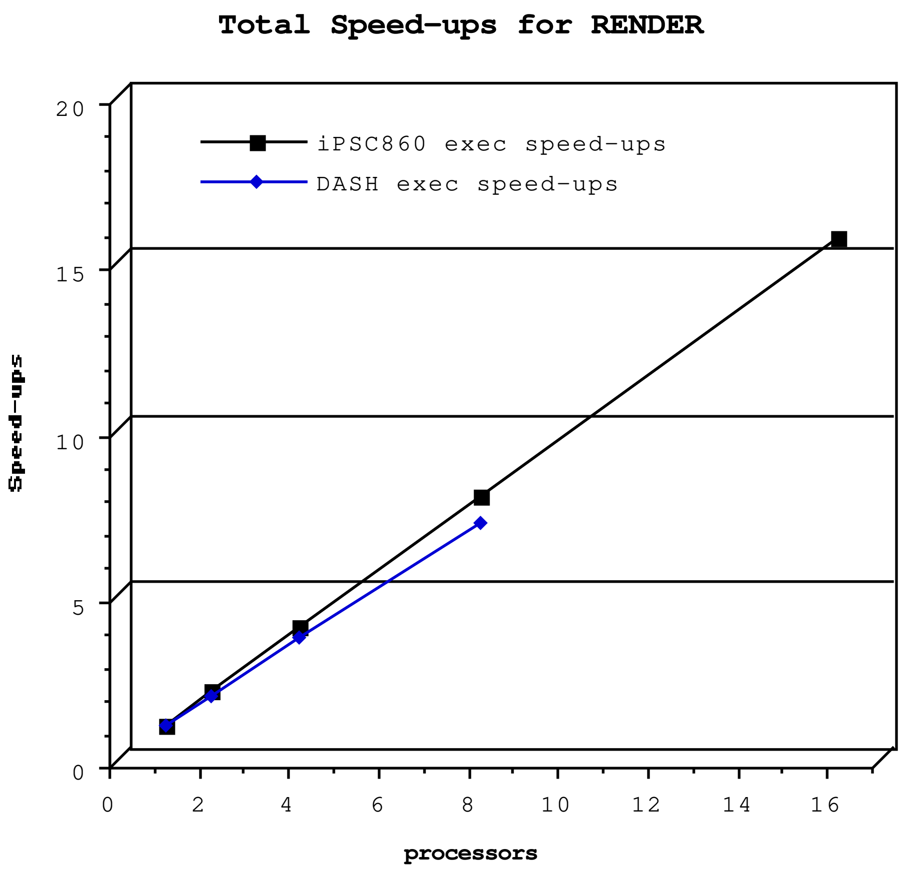



Chart 10 shows the speedup comparison of `RENDER` on `iPSC/860` and `DASH`. The two speedup results are fairly close for up to 8 processors, after which point the `DASH` results are currently unavailable. However, the speedup for `DASH` is expected to continue to scale well since communication is not the bottleneck in this program.

## Chart 11: RENDER Total Run Time

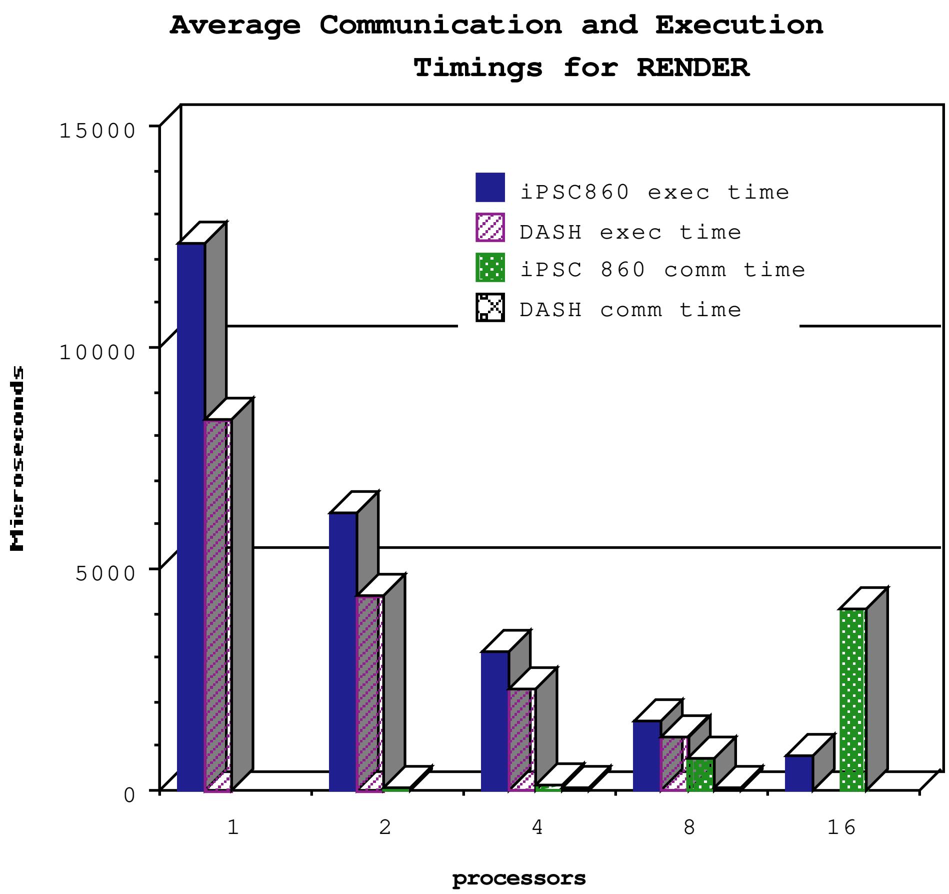



Chart 11 shows the average communication and ray–tracing time per frame for DASH and iPSC/860. DASH has lower CPU time, but the most surprising fact is that the communication cost for DASH remains very low and constant while the communication cost for the iPSC/860 increases dramatically. This suggests that the iPSC/860 library on DASH is more efficient for large messages, as expected.

# Section 4. Implementation Technicalities

## *Interprocess Communications and Key Data Structures*

Emulation of the iPSC/860 is an exercise in implementating efficient interprocess communications. The proper use of operating system primitives such as shared memory segments is crucial to the effective emulation of the iPSC/860 message–passing library. Message–passing is emulated by using standard queue mechanisms, without imposing the requirement that queue elements must be referenced in strictly FIFO or LIFO order. Each item in the queue is associated with a message "type". A specific message can be received by specifying the corresponding message "type". The receive calls in the library scans the queue for the first message of the associated "type".

### **msg_struct**

In C syntax, the structure of a "queued element" is as follows :

```
struct msg_struct {
   long type;              /* message type */
   long msglen;            /* length of message */
   long sender_node;       /* node ID of the sender */
   long sender_pid;        /* process ID of the sender */
   unsigned msg_offset;    /* offset from shmem base */
} MSG_STRUCT;
```

The message queue contains a message offset instead of the message itself. This offset is the offset for the message contents from the shared heap base address. An offset was used because `exec()'ed` processes are not guaranteed to attach the shared memory segments at the same virtual address. The contents of the message reside in shared memory segments that are allocated by the `d_share_malloc()` function. The `d_share_malloc()` function is an extension to the `GMALLOC()` macro that allows allocation from a specific cluster.

### msg_queue_struct

Each node maintains its own message queue for incoming messages. The structure of the message queue is as follows :

```
struct msg_queue_struct {
   int head, tail;          /* Head and Tail of the queue */
   LOCKDEC(lock);           /* Mutex for this queue */
   MSG_STRUCT msg[MAX_MSG]; /* list of messages */
}
```

### host_init

Shared memory segments for the message queues and all other synchronization variables are allocated by the "host" process when it calls the `host_init()` function. This function does the following :

**1)** *Creates a shared memory heap (or* `malloc()` *region) for* `G_MALLOC` *calls;*
**2)** *Creates a lock arena from which spin-locks are allocated;*
**3)** *Allocates shared memory pages for a shared read/write data structure (such as "*`gm`*");*
**4)** *Allocates shared memory pages for a read-only data structure (such as "*`unix_info`*").*

### gm

The message queues reside in the "`gm`" data structure along with various locks and barriers which provide mutual exclusion on data structures and synchronization among the nodes. The "`unix_info`" structure contains UNIX process ID's and other information used by both the host and node processes. The "`unix_info`" structure is created by the host, and it can only be read by the nodes. The structure of the "`gm`" data structure is as follows :

```
struct gm_t {
  MSG_Q    msg_q[MAXPROC+1]   /* The message queues */
  ASYNC_Q async_q[MAXPROC+1] /* hrecv() wait queue */
  INFO     info[MAXPROC+1]    /* for doing info()   */
  long     msgid;             /* Message ID for async */
  BARDEC(syncnohost)          /* for gsync() */
  LOCKDEC(printlock)          /* printf() resource lock  */
  LOCKDEC(memlock)            /* GMALLOC()/G_FREE() lock */
  unsigned Gxxx_offset;       /* For doing gxxx() calls */
  char     *pGxxx;
  long     Gxxx_sz, Gxxx_malloc_req, Gxxx_cnt;
  char     Gxxx_space[MAX_GXXX_SPACE];
  LOCKDEC(Gxxx_lock)
} GLOBAL;
```

**msg_q**

The "`msg_q`" field is also embedded in this global structure. Nothing in the current implementation allows each "`msg_q`" field to reside in the neighborhood of its owner node. A future investigation will determine if throughput can be improved by moving the actual queue to the vicinity of its owning node.

**async_q**

The "`async_q`" is used during `hrecv()` and `hsend()` asynchronous message–passing calls. This list contains a finite number of slots with information pertaining to messages that a node has been "registered" for. When a sending process forwards a message that a receiving node is already waiting for, a "method" pointer in the "`async_q`" is used to invoke the corresponding user defined function. In reality, the "`async_q`" is really a list of "registered for" messages. The structure of items in this queue is as follows :

```
struct async_wait_struct {
  char    *buf;       /* Pointer to user buffer */
  long     length;    /* Length of the user buffer */
  long     typesel;   /* Message type being waited for */
  void     (*proc)(); /* "Method" to be invoked */
  long     msgid;     /* Message ID for isend/irecv */
} ASYNC_Q;
```

*Copyright 1992 Stanford University.*

### info

The "info" field contains information related to the last message dequeued by the receiving node. There is one such "info" structure for each node. The "info" structure is used by the various "infoXXXX()" iPSC/860 routines. Every time a receiving function such as crecv() is called, this "info" field is updated. The structure of "info" is as follows :

```
struct info_struct {
   long pid;         /* Sender's logical process ID */
   long node;        /* Sender's node ID */
   long typesel;     /* Message type     */
   long count;       /* Message length   */
}
```

### NODE_INFO

The next key data structure is the "NODE_INFO" data structure, which contains information unique to each node. The most important field in this structure is the node id. A unique "NODE_INFO" data structure is passed to each node process from the host process via shared memory. This data structure is created using the standard UNIX System V IPC primitives shmget() and shmat(). "NODE_INFO" is created when the host calls attachcube(). The node processes "attach" themselves to these memory segments when they call the node_init() function in the emulation library. The structure of "NODE_INFO" is as follows :

```
struct node_info {
   unsigned        cube_id;     /* My cube ID */
   unsigned        node_id;     /* My node ID */
   unsigned        ipsc_pid;    /* My logical process ID */
   unsigned        no_nodes;    /* Number of nodes in cube */
   struct timeval  tp;          /* Time since cube boot   */
   struct timezone tzp;         /* Time zone adjustment   */
   int             sc_myid;     /* Used by ANL Macros     */
   struct timeval  sc_tp;       /* Used by ANL Macros     */
} NODE_INFO;
```

## UNIX_INFO

The next key data structure is the "UNIX_INFO" data structure. This data structure contains information that is needed in making certain UNIX system calls. The most important fields in this structure are the pid, ipsc_pid, shmid, and GMshmid fields. The pid field contains the UNIX process ID of each process (including the host). The ipsc_pid field contains the logical process ID of each process. This logical process ID is the process ID referred to by the iPSC/860 calls. The shmid field contains the UNIX shared memory handle for the "UNIX_INFO" structure, and the GMshmid field contains the shared memory handle for the "gm" structure. The structure of "UNIX_INFO" is as follows:

```
struct unix_info {
   long   pid[MAXPROC+1];       /* UNIX process ID's */
   long   ipsc_pid[MAXPROC+1]   /* Logical PID's     */
   int    curr_cube;            /* Active Cube ID    */
   int    total_cube;           /* Total Cubes allocated */
   unsigned shmid;              /* UNIX_INFO shmid   */
   unsigned GMshmid;            /* gm shmid          */
}
```

## CUBE_INFO

The final key data structure is the "CUBE_INFO" data structure. This structure is used by the host to maintain information related to the currently executing cube. At the present time only one cube exist through the course of a run, but this structure is present to allow for future expansion. The "CUBE_INFO" data structure is resident in the host's local memory and its structure is as follows :

```
struct cube_info {
   char cubename[MAX_CUBENAME]; /* Cube Name      */
   char cubetype[MAX_TYPENAME]; /* Cube Type Name */
   char srmname[MAX_HOSTNAME];  /* Host name      */
   long  keep;                  /* Keep cube at release */
   long  no_nodes;              /* No of nodes in cube  */
} CUBE_INFO;
```

## Completeness of the Library

The following iPSC/860 library calls have been tested on the 8–processor SGI 380VGX "hookipa", the 8–processor DASH, and the 48–processor DASH, and they are available for distribution on the DASH system:

```
– getcube(), attachcube(), load()
– killproc(), killcube(), relcube()
– waitall()
– cprob(), iprobe()
– csend(), csendrecv(), crecv()
– hsend(), hsendrecv(), hrecv()
– isend(), isendrecv(), irecv()
– msgdone(), msgcancel(), msgwait()
– mynode(), myhost(), mypid()
– setpid()
– gsync()
– masktrap()
– numnodes()
– dclock(), mclock()
– infonode(), infocount(), infopid(), infotype()
– gsync()
– giand(), gland(), gihigh(), gshigh(), gdhigh(), gilow(),
  gslow(), gdlow(), gior(), glor(), giprod(), gsprod(), gdprod(),
  gisum(), gssum(), gdsum(),
– gcol()
```

## High Level Pseudo–Code for Inter–Process Communication Routines

```
host_init()
     Call share_malloc_init to create a shared heap
     Call usinit to create a spin-lock arena

     Set up signal handlers

     Call shmget() and shmat() to create shared memory for UNIX_INFO
     Call shmget() and shmat() to create shared memory for "gm"

     Initialize all per process arrays in UNIX_INFO
     Initialize all per process arrays in gm

     Initialize all BARRIERS and LOCKS in gm
     Initialize remaining fields of UNIX_INFO and gm

     Read the pdebug_mask from an environmental variable
       and store in "gm"

     Initialize remaining "static" data
```

```
node_init (unixshmid, nodeshmid)
     Call share_malloc_init_child to attach to shared heap
     Call usinit to attach to the shared spin-lock arena

     Call shmat() to attach to UNIX_INFO, "gm", and NODE_INFO

     Exit with an error if memory didn't attach to the same virtual
       address as in the host

     Initialize sc_myid and sc_tp (for use in ANL Macros)
     Initialize my_node to my node_id given to me from the host
```

*Copyright 1992 Stanford University.*

```
getcube(cubename, cubetype, srmname, keep)
    Check for argument errors and for cube allocation limitations
    Allocate local memory for the CUBE_INFO structure
    Allocate local memory for a NODE_INFO structure for the host
    Initialize all fields in the CUBE_INFO structure
```

```
attachcube(cubename)
    Verify a cube by the name of "cubename" has been created
    Parse cube information to determine type and no of nodes
      to create
    Fill in the local NODE_INFO structure for the host only
    Fill in the host portions of the UNIX_INFO structure

    for each node to be created
     Do a fork()
      If I am the child then
       Attach the child to a specific node using sysmp()
       return
      Else
       Increment sc_myid (for ANL)
       Store the child's process ID in the UNIX_INFO structure
      Endif
     End
```

```
load(filename, node, pid)
    Check for argument errors
    Call shmget() and shmat() to allocate shared memory for
      NODE_INFO
    Copy the local node info to this shared memory (initialize)
    Initialize NODE_INFO fields not properly initialized by the copy
    Call execlp() to execute the desired filename
```

**csend(type, buf, length, node, pid)**

```
Check for argument errors
Call shm_send() to send data to desired node(s)
```

**csendrecv(type, sbuf, slen, tonode, topid, typesel, rbuf, rlen)**

```
Check for argument errors
Call shm_send() to send data to desired node(s)
Wait for the desired message to be received by calling chk_msg
When the message is received copy it to the local buffer
and call ipsc_free() to free the shared space
```

**crecv(type, buf, length)**

```
Check for argument errors
Wait for the desired message to arrive
Copy the message to buf
Call ipsc_free() to free the shared space
Fill in the info field of "gm" for use in subsequent info()
  calls
```

**hsend(type, buf, length, node, pid, proc)**

```
Check for argument errors
Call shm_send to send the message and a "SIGNAL" to the
  receiver
If proc is not NULL
Call proc to inform caller that the message has been sent
```

**hrecv(type, buf, length, proc)**

```
LOCK the async_q of my node
Insert the type, buf, length, proc into the async_q list
UNLOCK the async_q

Remove any messages of "type" that are still the message queue
Free up shared memory allocated to those messages
Call the handler "proc" for each of those messages
```

**isend()**

```
Check for argument errors
Set msgid = gm->msgid++
Call shm_send
return msgid
```

**irecv()**

```
Check for argument errors
Set msgid = gm->msgid++
lock out interrupts
If the desired message has not already arrived then
 insert the buffer pointer into the asynchronous receive queue
Else
 copy the message into the caller's buffer
 remove the message from the queue
 Call ipsc_free to free the message
unlock interrupts
return msgid
```

**gsync()**

```
BARRIER(syncnohost)
```

**killproc (node, proc)**

```
Check for argument errors
If all nodes are to be killed
Send SIGKILL to all nodes except mine
Free up all dynamically allocated memory
Send SIGKILL to my node
Else
Send SIGKILL to the desired node
```

*Copyright 1992 Stanford University.*

**gxand(), gxhigh(), gxlow(), gxor(), gxprod(), gxsum(), gxxor(), gxcol()**

```
Call gxxx_get_mem to allocate memory if the vector is huge
If this is the first node to get here
 allocate space for all node's vectors plus the result vector
Store this node's input vector into the proper location
Synchronize the nodes
Loop through the vector and perform the vector operation
 on the vector elements (my_node % num_nodes) so each
 node does 1/num_nodes of the total work
Call gxxx_rel_mem to release dynamically allocated memory
```

**gxxx_get_mem()**

```
Use a lock to force sequential access
If this is the first node to get here
  If the vector is huge
    allocate shared memory to hold input and answer vectors
  else
    store a pointer to statically allocated space
else if this node attempts to use a different vector size
  Unlock
  return an error to the caller
else
  Unlock and return OK status
```

**gxxx_rel_mem()**

```
Synchronize all nodes with gsync()
Copy the answer vector to this nodes buffer
Synchronize all nodes with gsync ()
LOCK
If this is the first node to get her
  If dynamic memory was allocated for the huge vector
    release the memory
UNLOCK and return OK status
```

## Challenges Encountered

The major implementation challenges to–date can be classified into three major areas :

**1)** *The use of* `exec()` *to create the children processes.* This problem was the most difficult to overcome. The reason for using `exec()` after a `fork()` instead of just a `fork()` stemmed from the goal to emulate the iPSC/860 as much as possible. The iPSC/860 allows a `filename` to be specified in the `load` command to load one or multiple node program(s). All of the group's previous experience with the ANL Macros was limited to working in an environment with `fork()`'ed children only, thereby creating the need to reverse engineer the functionalities of the ANL Macros. It was also necessary to investigate and in some cases to circumvent `shmem.c` –– the shared memory module written by Goldshmit at Stanford –– which creates a shared heap and lock arena. Since the children created via `exec()` do not inherit the parent's memory map, it becomes necessary to attach the shared memory segments explicitly. The first clue that pointed to this problem was given by locks that did not seem to be secure, and by a heap that was being corrupted. Thoughtful investigation led to the conclusion that the children were creating independent lock arenas and were re–initializing the heap. The next problem was how to get the children to attach the segments at the virtual addresses used by the parent. The solution is to pass "pointers" as offsets from the shared base rather than absolute pointers.

**2)** *The lack of convenient debugging utilities to debug children processes.* The second problem was a valuable learning experience. Older versions of `dbx` do not allow debugging of children processes. Once the heuristics were known, debugging proceeded smoothly.

**3)** *Mutual exclusion problems.* The third problem could not be corrected until the first problem with lock arena was solved. Until the locks worked correctly, the processes violated mutual exclusion requirements and corrupted data. This problem was resolved as soon as the lock arena problem was corrected, but it still appears as an occasional bug in the library. Minor race conditions, mutual exclusion problems, and memory corruption problems recur despite continuous debugging efforts.

*Copyright 1992 Stanford University.*

**4)** The authors would have used some of the performance debugging tools (like `MemSpy` and `MTOOL`), but eventually decided against that since those tools were heavily geared toward standard shared memory models. `MemSpy` runs in the `Tango Lite` environment using `sproc()`'ed processes, which could expose the test cases to a high risk of accidental overwrites.

**5)** Overcoming UNIX limitations proved to be one of the most challenging aspects of this project. It was necessary to find the most efficient way to move data between two processes while still maintaining the iPSC programming model. If it was not necessary to maintain complete compatibility to the iPSC/860, it would be possible to less constrained models which would have given higher performance.

In the attempts to achieve reasonable performance under UNIX the authors had explored the use of sockets instead of signals, the use of `mmap()` to move data between processors, the possibility of using light–weight threads, and many other interesting feature that could be found on the DASH machine.

The conclusion is that standard UNIX is severely lacking in the ability to provide a programming environment that is well–suited to the development of multi–processing applications. The memory protection model is useful for protecting novice users from interfering with each other, but it becomes a hindrance to the experienced programmer who is attempting to exploit any and all concurrencies available on a multi–processor machine.

The UNIX context switch time and the lack of good inter–processor communication primitives (like kernel managed–message queues, event flags, and mailboxes) severely limits DASH's ability to outperform the iPSC when running under the same strict message–passing paradigm. Relaxation of the strict message–passing paradigm seems to be an interesting future direction; performance gains are also expected from the use of light–weight threads, assuming that they are managed by the kernel instead of the application.

**6)** It was a challenge to learn about the several applications being used as test cases as a means to debug the library. That required learning much more about the applications than the authors would have expected.

**7)** Taking full advantage of the special features of DASH was a challenge for several reasons. For one, some of the DASH specific system calls like `sysmp()`, for instance, cause programs to "hang" when invalid arguments are passed to them. It is also difficult to determine how much performance gain can be achieved through the use of various system calls. A final challenge lies in the use of the prefetching capability which seems to cause kernel panics.

**8)** Half way through the project, the authors decided that it was necessary to either write a special dynamic memory allocation routine or modify the routines currently available from the `ANL` Macros. The later method was chosen since it presented the challenge of learning about how shared heap management was performed, as well as how heap management can be modified to allow for multiple heaps (or one heap per cluster.)

**9)** The greatest challenge of all was to get everyone on the development team to agree on how to attack the obstacles before us. Each individual brings different strengths and weaknesses to the table and a plan of attack must be formulated. Getting three people to work together toward a common goal, while each operates on a different schedule and style was both a challenge and a blessing.

# Section 5. Performance Analysis

## Comparative Evaluation

Several benchmarking tests were performed to determine the effectiveness of key routines in the library. The key routines chosen for benchmarking tests were: `csend()`, `crecv()`, `gsync()`, `giand()`, `gdsum()`, and `gdprod()`. `Csend()` and `crecv()` are obvious choices since they are the most elementary message–passing primitives. The `gsync()` function was chosen due to its simplicity and the fact that it requires global synchronization. The `giand()`, `gdsum()`, and `gdprod()` functions were chosen as "representative" of other the "global operation" functions. The `giand()` function is typical of all the "integer" operations. The `gdsum()` function was selected to compare floating point arithmetic performance and the `gdprod()` function was selected to compare the floating point multiplication performance, which is one of the most CPU–intensive operations.

Three different message–passing tests were performed. The first was a simple test designed to measure the amount of time required for a single `csend()` or `crecv()` call to complete. The test employs a single node, which sends messages back to itself. The second test was designed to measure performance and its degradation when messages are sent between separate nodes (either on the same DASH cluster or on a different cluster.) The third test was a broadcasting test. This is an $O(n^2)$ operation where each node broadcasts messages to and receives messages from all other nodes. This test was designed to put the system under a maximal stress–load, and to determine the performance degradation that could occur due to excessive network traffic and lock–contention collisions between processors.

Two sets of results were collected on DASH, which represent results for the non–optimized and DASH–optimized libraries. The non–optimized library is the older version which excludes the distributed `malloc()` features that allocate memory from the heap of a specific cluster. This distributed `malloc()` feature allows the user to allocate buffer space from the receiving node's physical address space, thus reducing the amount of time required for the receiver to read the data across the network into its local cache. The DASH–optimized library has also incorporated other DASH–specific features, such as invoking a single instance of copying per cluster upon a broadcast.

## Results

### Graph 1

Graph 1 shows the performance of `csend()` and `crecv()` for the single node case. Several conclusions can be drawn from the graph. First, it is obvious that the iPSC/860 performs `csend()` and `crecv()` calls more efficiently than the simulation library for message lengths of less than `100` bytes, but the emulation library is more efficient for buffer lengths greater than `100`. Sources from Intel Scientific Computing have indicated that the iPSC/860 optimizes message passing for buffer lengths less than `100` bytes by using preallocated static buffers rather than dynamically allocating buffers. It also appears that messages sent from any node in the iPSC/860 back to itself are also optimized. Second, the DASH–optimized code performed slightly worse than the non–optimized code for small buffer lengths, but for larger message lengths the completion time remained nearly constant while the non–optimized code degraded progressively as the message size was increased

### Graph 2

Graph 2 shows the performance of `csend()` and `crecv()` between two separate nodes. The non–optimized results indicate the timing for nodes on the same cluster as well as those for nodes on different clusters. No results were obtained for the DASH–optimized codes for message–passing on the same cluster; optimization should not make too much difference for the sake of this comparison. This graph appears to give results similar to graph 1. The non–optimized code did slightly better than the optimized code for small message lengths, but the optimized code performed much better for large message lengths. One explaination for this is that there is a slight increase in overhead for the optimized code that degrades performance until message–copying begins to dominate the execution time.

**Graph 3**

Graph 3 shows the performances of `csend()` and `crecv()` when the functions are used to broadcast `1`–byte messages. This is an area where there is an obvious improvement for the DASH–optimized code. One of the optimizations implemented a single instance of message copying from the sending node, thereby requiring only the last receiver of the message to free the message buffer. This scheme significantly reduces the instances of message–copying among the clusters. This graph shows that the iPSC/860 performs broadcasting better than the simulation library. One explanation is the large amount of required message–copying (each receiver must copy the message) and the overhead due to global synchronization. There was a clear improvement in the DASH–optimized library over the non–optimized version. Non–optimized performance data for 16 and 32 processors was not collected.

**Graph 4**

Graph 4 shows the performance of `csend()` and `crecv()` when used to broadcast `1000`–byte messages. Once again, the optimized code performed better than the non–optimized code. This test also shows that, for large messages, the DASH–optimized code compares much more closely to the iPSC/860, taking about twice as long to complete the functions as on the iPSC/860. One reason is probably due to the 100–byte message optimization in the iPSC/860. The non–optimized results took a large jump when the number of processors increase to 8. It is not clear whether that is turly indicative of performance degradation, which is a possibility since the non–optimized code does broadcast a significantly higher number of messages over the network.

**Graph 5**

Graph 5 shows the performance of a relatively simple "global operation" function, namely the `gsync()` function. The `gsync()` function is simply a barrier function and, therefore, it is expected to be a fast operation. The graph indicates that these barrier operations are much faster on DASH than on the iPSC for up to 16 processors, but they become increasingly expensive beyond 16 processors. The non–optimized and DASH–optimized performances are about equal, since no optimizations were conceived for these "global operations".

**Graph 6 and 7**

Graphs 6 and 7 shows the performance of the `giand()` function for 10–element and 1000–element vectors, respectively. Graph 6 indicates that, for a small number of processors (less than or equal to 8), the DASH performance is better than the iPSC/860, but for more than 8 processors the iPSC/860 performs better. Graph 7 indicates that for large vectors, the DASH library consistently outperforms the iPSC up to the 32 processor configuration. It is unclear why the optimized code performed slightly better than the non–optimized code.

**Graph 8 and 9**

Graphs 8 and 9 show the performance of the `gdsum()` function for 10–element and 1000–element vectors respectively. Both graphs indicate that the DASH library performance exceeds that of the iPSC/860 up to 32 processors for both large and small vectors. Graph 9 once again indicates that the DASH machine out–performs the iPSC/860 in "global" operations as the vector size increases. This is due to an algorithm that distributes the "global" operation processing among the processors. The non–optimized and DASH–optimized simulation library offers similar performance, as expected.

**Graph 10 and 11**

Graphs 10 and 11 show the performance of the `gdprod()` function for 10–element and 1000–element vectors respectively. Once again, both graphs indicate that the DASH library performance exceeds that of the iPSC/860 up to 32 processors for both large and small vectors. The best explainationas for why `gdsum()` and `gdprod()` do so much better on DASH than on the iPSC/860 is that the iPSC/860 is not using a floating point accelerator commonly found in SGI–based architectures. Not only is the iPSC/860 floating performance not as good, but floating point multiplications also take longer, as expected, than floating point adds. The DASH machine is surprisingly efficient as well as consistent in floating point operations. For large vector lengths, the DASH machine's performance is indeed impressive.

*Copyright 1992 Stanford University.*

## *Chart 1: Csend()/crecv() from a Node to Itself*

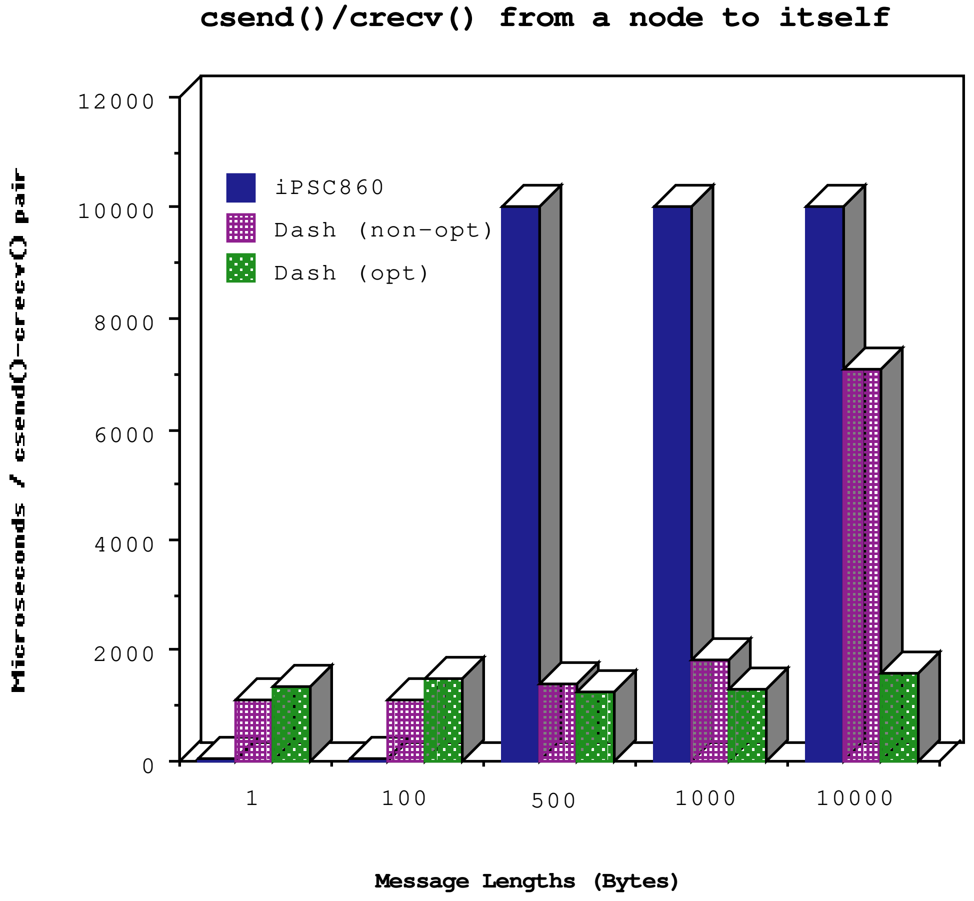



Chart 1 shows the best case performance of the `csend()` and `crecv()` functions. The test program does `10000 csend()` and `csrecv()` calls for different message lengths from a single node. The node receives each of the messages that it sends out.

## Chart 2: Csend()/crecv() between Separate Nodes

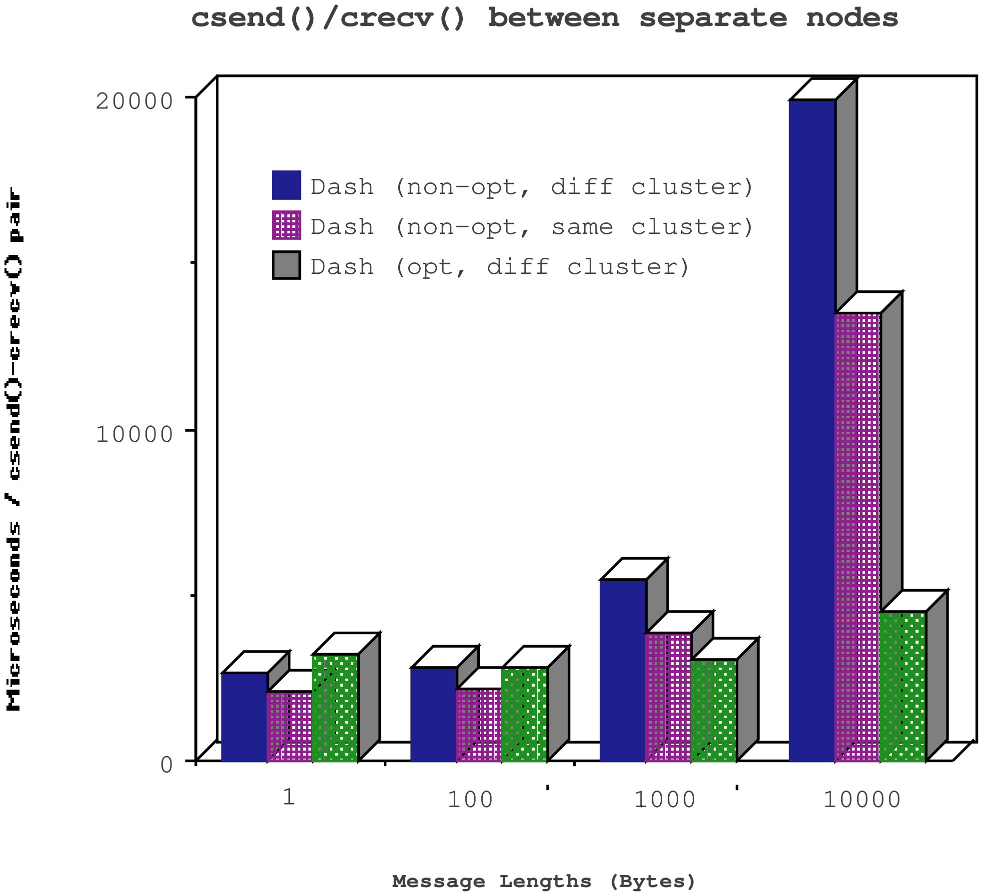



Chart 2 shows the best-case performance of the `csend()` and `crecv()` functions when they are performed between two separate node processes. The test program performs `10000 csend()/crecv()` pairs from each of the two nodes participating in the test. The timings on the graph indicate the time required for each participating node to do a `csend()` and a `crecv()`.

## *Chart 3: Csend()/crecv() Broadcast for 1–byte Messages*

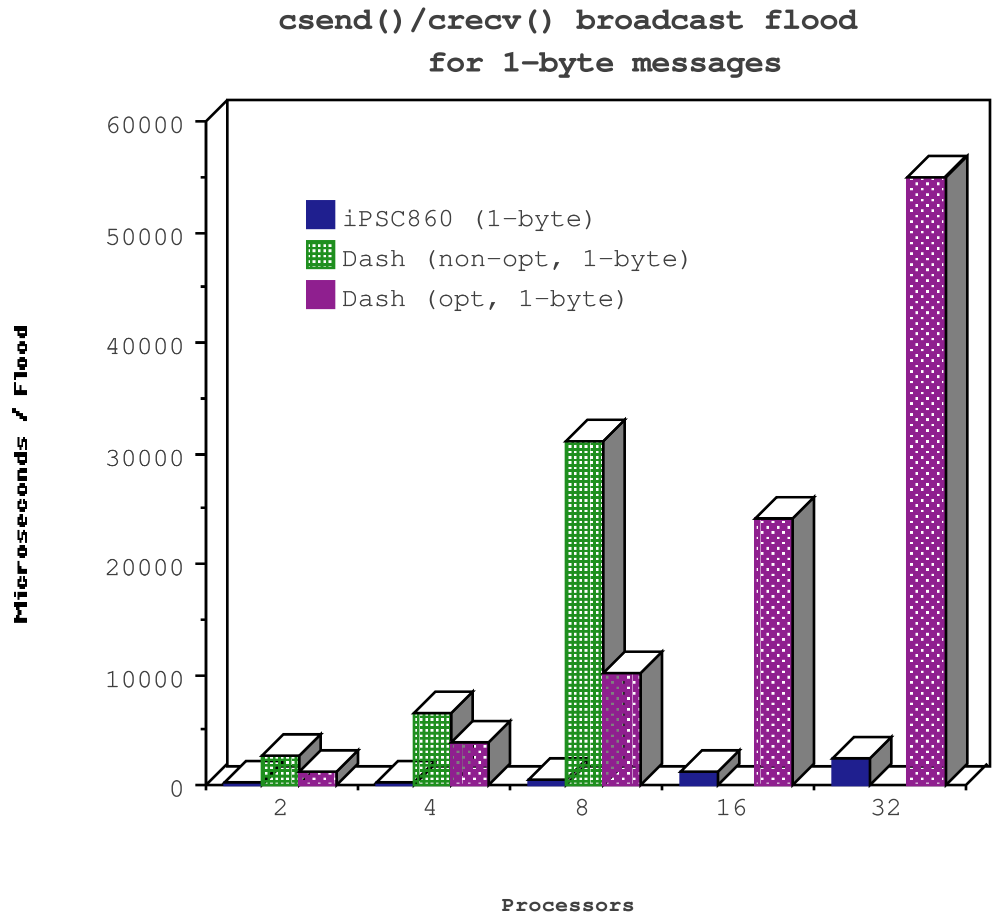



Chart 3 shows the performance of `csend()` and `crecv()` calls in a "broadcast" scenario with `1`–byte message lengths. Each iteration of the test involves broadcasting a message to all other nodes and receiving a message from all other nodes. The timings on the graph indicate the time required for each one of these iterations.

## Chart 4: Csend()/crecv() Broadcast for 1000–byte Messages

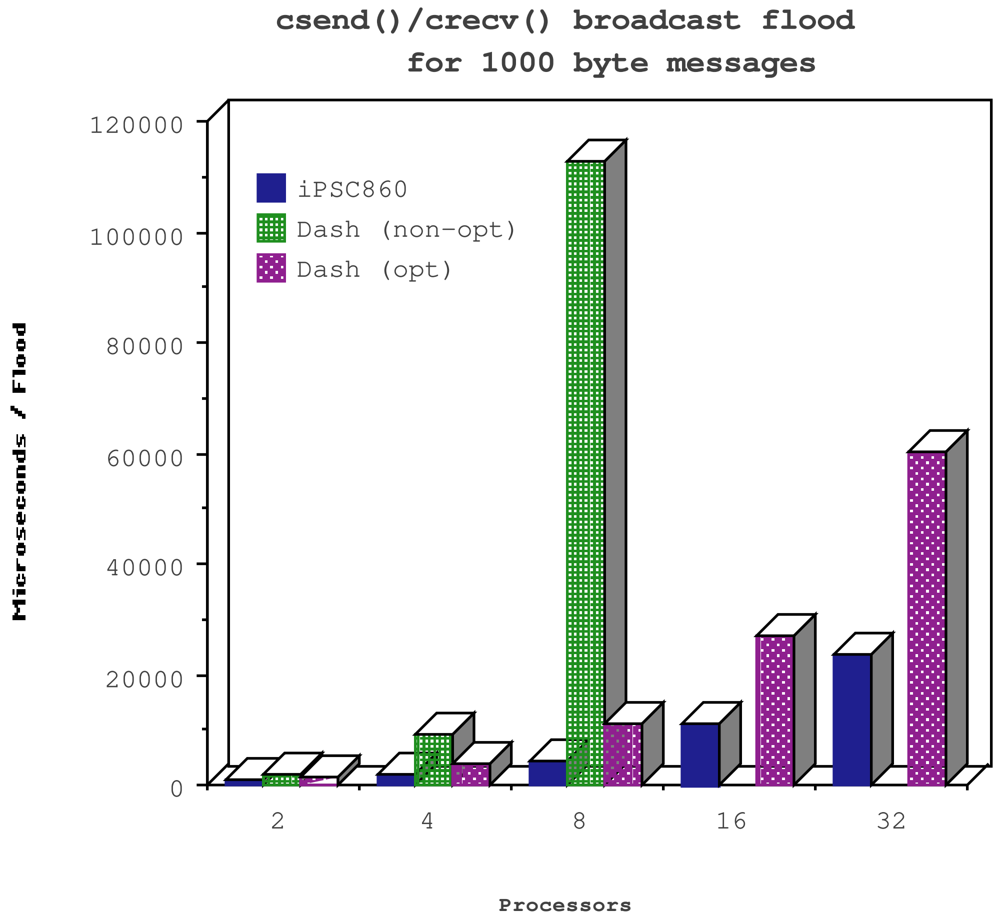



Chart 4 shows the performance of `csend()` and `crecv()` calls in a "broadcast" scenario with `1000`–byte message lengths. Each iteration of the test involves broadcasting a message to all other nodes and receiving a message from all other nodes. The timings on the graph indicate the time required for each one of these iterations.

*Copyright 1992 Stanford University.*

## *Chart 5: Gsync() Timing Comparisons*

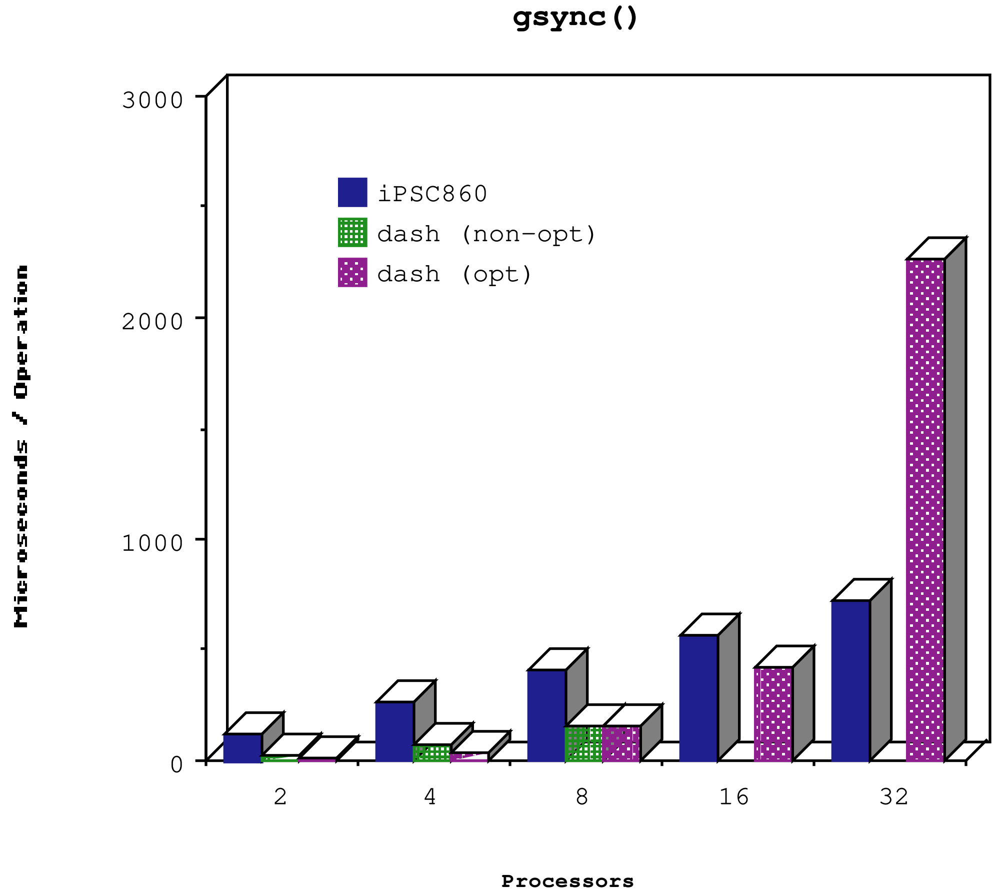



Chart 5 intends to present the performance of the `gsync()` call. The test program performs `10000 gsync()` calls and measures the time required for each call. The timings on the graph indicate the time required for a single `gsync()` call on any of the node processes participating in the test.

## *Chart 6: Giand() Timing Comparisions on 10–element Vectors*

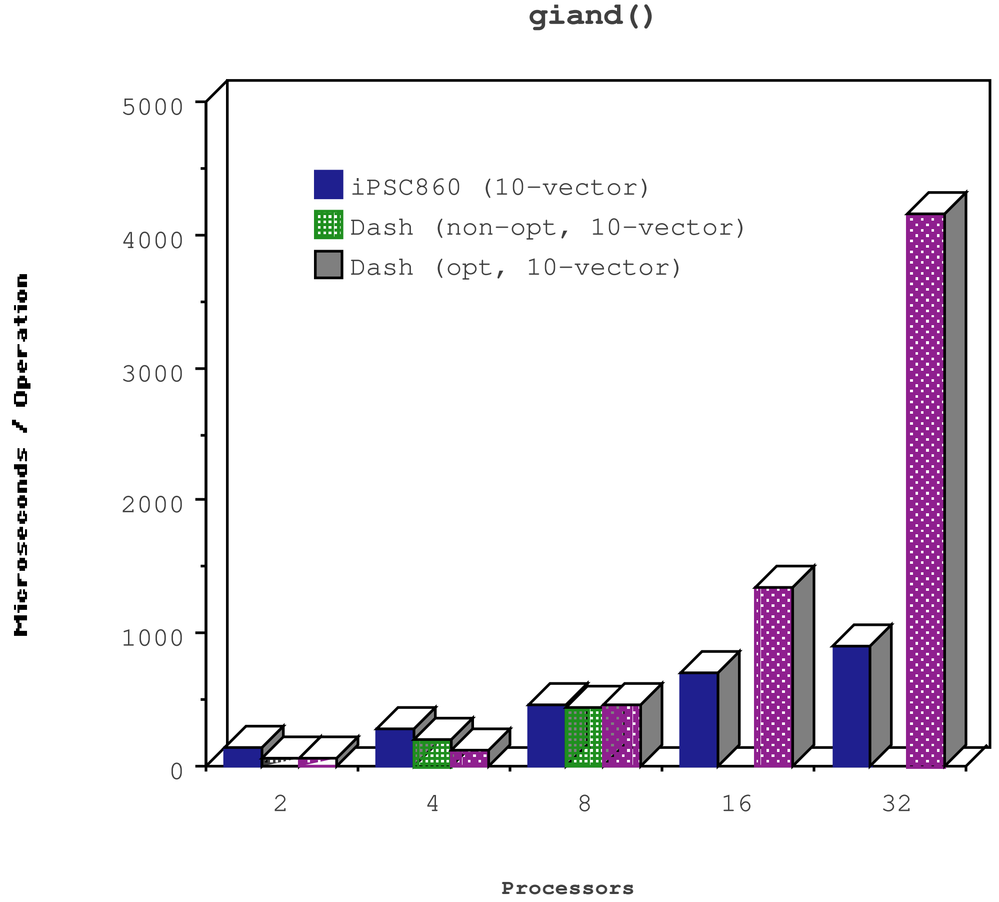



Chart 6 shows the performance of the `giand()` call for `10`–element integer vectors. The test program performs `1000 giand()` calls and measures the time required for each call. The timings on the graph indicate the time required for a single `giand()` call on any of the node processes participating in the test.

## Chart 7: Giand() Timing Comparisions on 1000–element Vectors

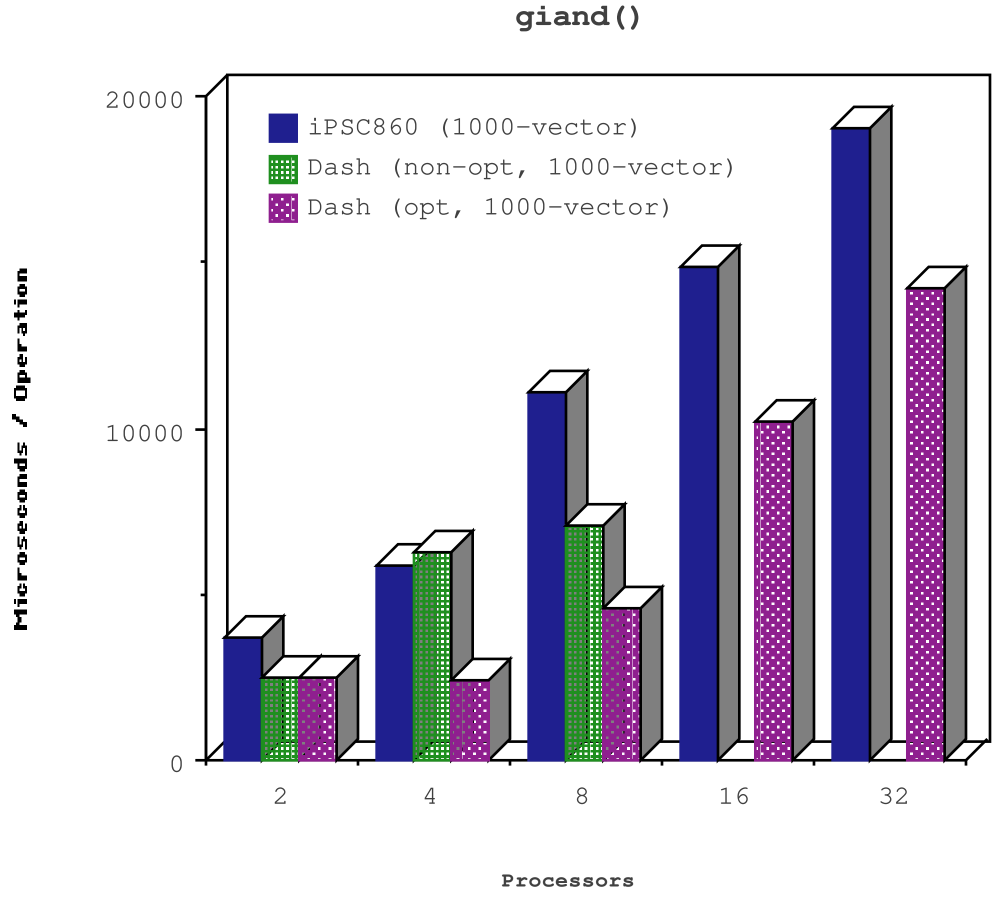



Chart 7 shows the performance of the `giand()` call for `1000`–element integer vectors. The test program performs `1000` giand() calls and measures the time required for each call. The timings on the graph indicate the time required for a single `giand()` call on any of the node processes participating in the test.

## *Chart 8: Gdsum() Timing Comparisions on 10–element Vectors*

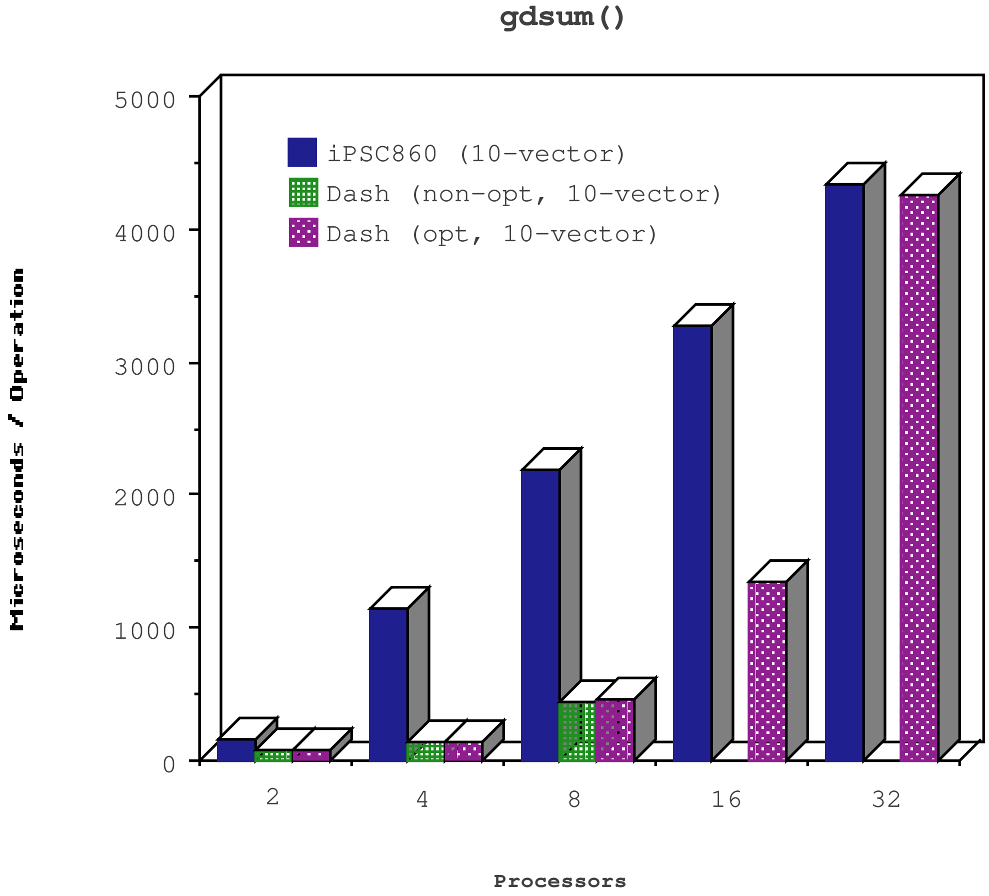



Chart 8 shows the performance of `gdsum()` calls for `10`–element double precision floating point vectors. The test program performs `1000` `gdsum()` calls and measures the time required for each call. The timings on the graph indicate the time required for a single `gdsum()` call on any of the node processes participating in the test.

*Copyright 1992 Stanford University.*

## Chart 9: Gdsum() Timing Comparisions on 1000–element Vectors

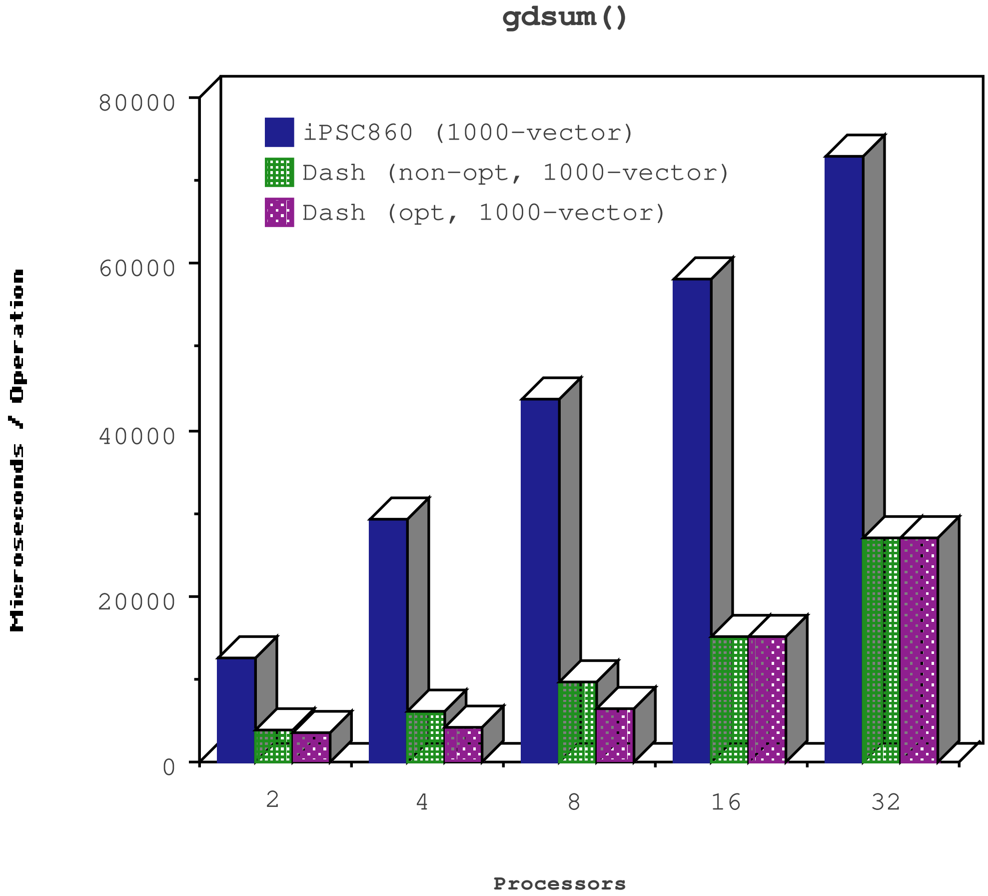



Chart 9 shows the performance of `gdsum()` calls for `1000`–element double precision floating point vectors. The test program performs `1000` `gdsum()` calls and measures the time required for each call. The timings on the graph indicate the time required for a single `gdsum()` call on any of the node processes participating in the test.

## *Chart 10: Gdprod() Timing Comparisions on 10–element Vectors*

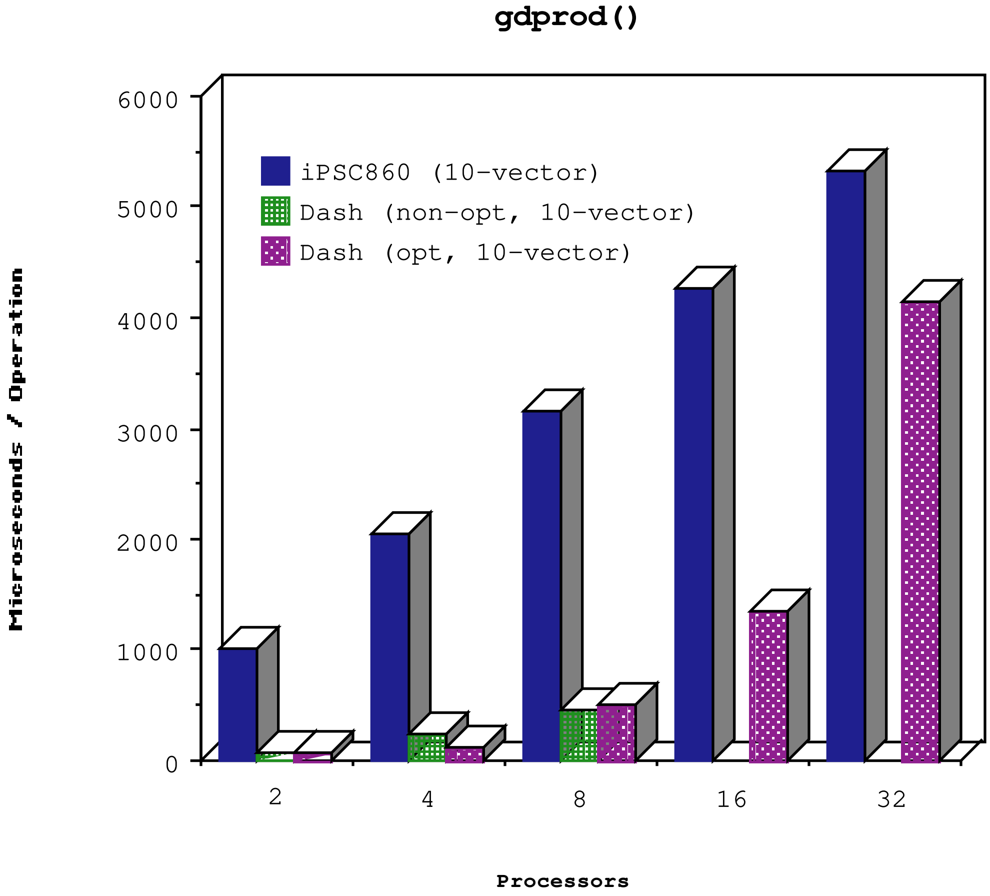



Chart 10 shows the performance of `gdprod()` calls for `1000`–element double precision floating point vectors. The test program performs `1000 gdprod()` calls and measures the time required for each call. The timings on the graph indicate the time required for a single `gdprod()` call on any of the node processes participating in the test.

## *Chart 11: Gdprod() Timing Comparisions on 1000–element Vectors*

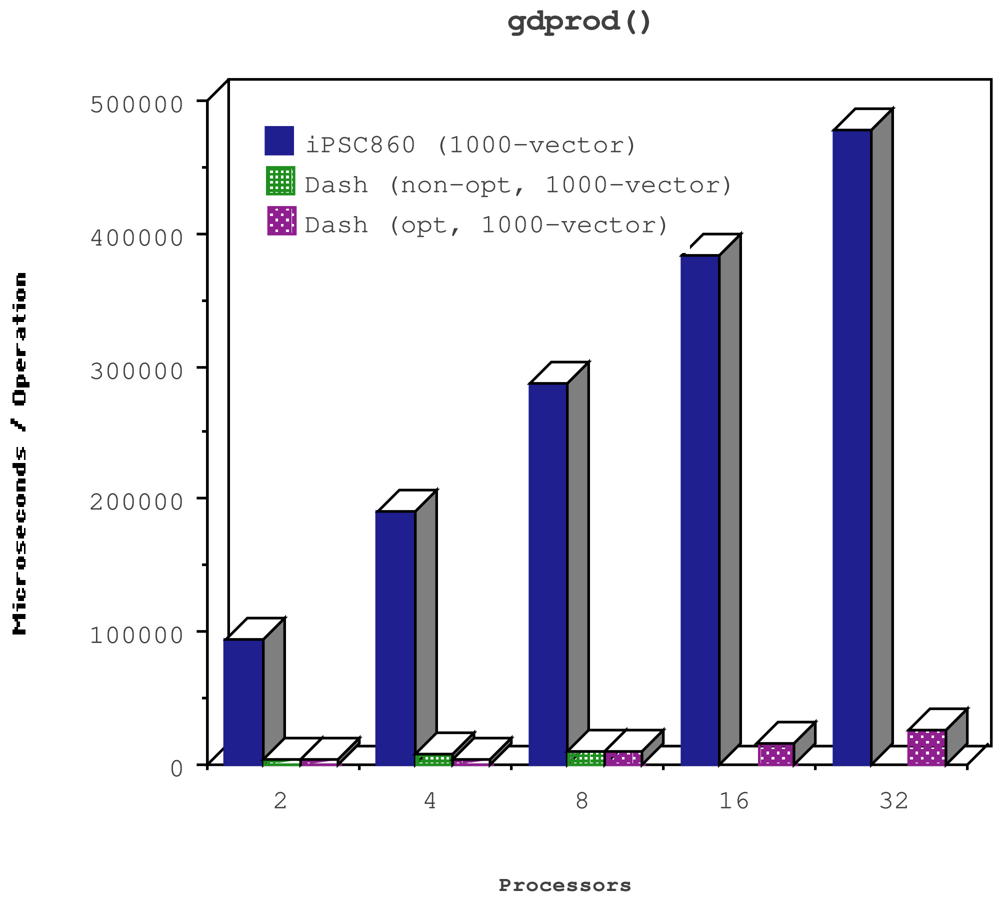



Chart 11 shows the performance of the `gdprod()` call for `1000`–element double precision floating point vectors. The test program performs `1000` `gdprod()` calls and measures the time required for each call. The timings on the graph indicate the time required for a single `gdprod()` call on any of the node processes participating in the test.

# Section 6. Conclusion

## *Lessons and Milestones*

On the practical side, the authors have learned two lessons from this undertaking:

**1)** *In many typical simulation and rendering programs, too often the programmers would prefer to write their computational algorithms under the message–passing paradigm so that they will be forced to exploit all the possible localities in their algorithms. On the other hand, many also see the advantage of being able to store their global data in shared memory, although it is often too expensive and unnecessary to maintain a completely shared single address space. In effect, this project has closed that previously unfilled technical gap. The important point here is that many future architectures will probably be a combination of both message–passing and shared–memory schemes. The user should not need to worry about those architectural details; instead, the programming paradigm should afford a uniform abstraction for attacking any computing problem.*

**2)** *From a program development point of view, this simulation library is a valuable utility to those who are developing parallel algorithms. Albeit the promise of cheap and massive computing power, parallel processors remain an impossible piece of machine for many scientists with little or no systems backgrounds. Debugging codes on a message–passing machine is an ardent task even for the most experienced systems programmers. Shared–memory provides far more visibility; accordingly, it is much easier to develop an algorithm on a shared–memory architecture using a message–passing library, so that the algorithm can be readily exported in a latter date to a message–passing architecture or other architectures which provide less visibility.*

The authors are also happy to report that a few minor milestones have been set. For instance, those who have a chance to appreciate the complexities of the library will probably refrain from subscribing to the textbook myth that it is a trivial matter to emulate message–passing on a shared–memory architecture; in fact, the authors feel that the myth should be written the other way around! In addition, plans have been made to collaborate with a number of laboratories around the world to turn the simulation library into a benchmarking standard. The authors' efforts have been well–rewarded so far.