Warning: *This paper was written for an MIT area exam and the material is intentionally inflammatory. Suggestions are welcome.*

A Critique of FLASH, TYPHOON, and SHRIMP

Frederic T. Chong*
ftchong@ai.mit.edu

September 7, 1994

1 Introduction

I critique three papers which describe FLASH [KOH+94], TYPHOON [RLW94], and SHRIMP [BLA+94], three recent multiprocessor designs. Although I focus on the scalability and performance of these systems,[1] I also examine protection and usability issues central to these systems. In examining these four issues, I focus on communications requirements.

All three systems provide separate communications hardware to minimize processor overhead during communication. None of the papers provide substantial motivation for this decision, but I present applications data that supports this decision.

TYPHOON demonstrates the benefits of user-level shared-memory with EM3D, an irregular electromagnetics application. However, close examination of EM3D reveals poor support for the TYPHOON conclusions. In fact, a message-passing implementation of EM3D would be no harder to write, higher in performance, and more scalable through dramatically lower memory requirements.

FLASH and TYPHOON both present evidence that software caching achieves good performance which scales to large working sets. However, FLASH does not provide adequate performance justification for their use of a custom processor in their network interface. TYPHOON achieves nearly the same performance with a stock processor, saving substantial processor design and software effort, as well as providing more robust performance.

To ensure protection in a multiprogrammed environment, all three systems are based upon the following premise: either a system must restrict user access to messaging hardware or the messaging hardware and network must be context switched. They miss the important alternative of unrestricted user access with timeouts that ensure network resources are available to the next user. The *T [PBGB93] system presents this alternative.

Additionally, the `map` operation provided by SHRIMP, for establishing protected communication between pairs of processors, is inadequate for irregular or cooperative applications. I propose extensions to `map` which provide scalable support for efficient group communication. I also note that `map` may need to operate on buffers smaller than a page in order to make memory requirements manageable.

The discussion begins in Section 2 with some brief system overviews to familiarize the reader with the three systems. Section 3 provides evidence supporting communications hardware, as well as arguments against the use of a custom processor in the FLASH hardware. Section 4 discusses protection in a multiprogrammed environment. It also presents my extensions to the `map` operation. Section 5 describes the benefits of software caching. Finally, Section 6 discusses shared-memory and message-passing implementations of EM3D and related irregular applications.

*Fred Chong is supported in part by an Office of Naval Research Graduate Fellowship. This paper is available via anonymous ftp from transit.ai.mit.edu or through the web from http://www.ai.mit.edu/people/ftchong/ftchong.html

[1] This paper is the result of an MIT area exam, in which doctoral candidates are given a three-week period to examine a broad question outside of their dissertation area. The original paper was submitted April 7, 1994 to the exam committee of Professors Bill Weihl, Arvind, and Jerry Saltzer.

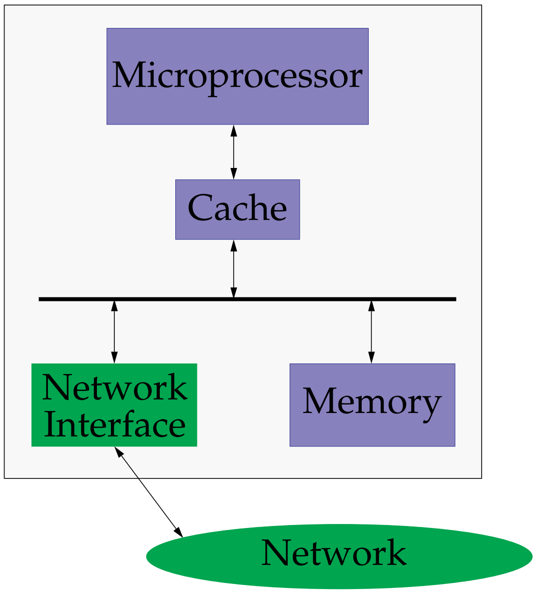

Figure 1: A typical multiprocessor node.

2 System Overviews

This section provides a brief overview of the essential features of each system. All three systems follow the current trend of tying commercial microprocessors together with an interconnection network to form a multiprocessor. Figure 1 shows the general structure of such systems. The network interface is generally the only custom part on the node.

Each system also addresses the need to reduce communications overhead to the microprocessor by providing extra communications hardware within the network interface on each node. The systems differ, however, in whether this hardware supports shared-memory protocols and whether users are allowed to access the hardware.

2.1 FLASH

The Stanford FLASH (FLexible Architecture for SHared memory) provides a good example of a multiprocessor with integrated shared-memory and message-passing support.[2] The most distinguishing feature of the machine is its custom VLSI protocol processor, called MAGIC, which is optimized to provide both message-passing and shared-memory operations. MAGIC takes the place of the network interface in a conventional multiprocessor node. For protection reasons discussed in Section 4, MAGIC only runs trusted, system-level code.

2.2 TYPHOON

In contrast to FLASH, Wisconsin's TYPHOON provides user-level access to a network interface processor (NP) through a software interface called *Tempest*. As with MAGIC on FLASH, the NP takes the place of the network interface on a conventional multiprocessor. User-level access includes executing user code on the NP. User code on the NP enables their key result, "user-level shared memory," where the user implements custom shared-memory protocols tailored to their specific applications.

Unlike the custom MAGIC chip in FLASH, the NP in TYPHOON is based upon a SPARC integer processor. This decision will be examined in Section 3.

[2] Such integration was first proposed by another ongoing project, the MIT ALEWIFE multiprocessor [KJA+93].

2.3 SHRIMP

While FLASH and TYPHOON are designed to support coherent shared-memory, the Princeton SHRIMP multiprocessor adopts a simpler model of memory-to-memory communication. As a result, SHRIMP only requires a DMA (Direct Memory Access) engine in their network interface, instead of an entire processor as in FLASH and TYPHOON.

The primary contribution of SHRIMP is low-overhead, protected communication through the separation of protection and data transfer operations. Such separation was first proposed by Wilkes for the *Hamlyn* interface [Wil92]. Protection is checked using a `map` operation, which uses a kernel call to map a send buffer on one processor to a receive buffer on another processor. Once this mapping is performed, data can be efficiently and safely communicated between buffers at user level without kernel intervention. `map` is a coarse-grain mechanism and requires that buffers must be at least the size of a page. Additionally, buffers are allocated in physical memory to ensure that communication does not incur page faults. The important advantage of this allocation is that messages will not be stuck in the network waiting for page faults. The big disadvantage is that memory requirements can be quite large with buffers at least as large as a page.

3 Message and Protocol Hardware

Based on a study of the three systems, this section presents evidence in favor of hardware support for message passing, as well as arguments against designing a custom protocol processor.

3.1 Communications Overhead

All three systems provide message-passing hardware to minimize overhead on compute processors. Processor overhead generally includes composing messages, moving messages to or from the network interface, and checking the progress of messages.

A separate messaging unit allows the compute processor to initiate a message send in a few cycles and then return to its computation. The mechanics of breaking up a large transfer into packets, reading the actual data from memory, checking message status, and resending failed messages are all handled by the messaging unit. The messaging unit can receive messages from the network and write the received data to memory.

None of the papers, however, quantify the impact of this reduced overhead on applications. We studied the overhead in sparse triangular solves on the CM-5 [CSBS94]. Having no separate messaging unit, the CM-5 [Thi93a] requires the compute processor to perform all the mechanics of sending and receiving a message. Our study found that such overhead accounted for 15 to 40 percent of execution time. This data supports the use of separate messaging hardware in FLASH, TYPHOON, and SHRIMP.

3.2 Custom Protocol Processor

In addition to supporting basic send and receive functions in their network interfaces, both FLASH and TYPHOON include a processor to execute *message handlers*. A message handler is code which is executed upon receiving a message. A message may specify the handler to be executed and may contain arguments for the handler. Message handlers have been shown to efficiently support a variety of communication and computation models [D+92] [vCGS92] [CDG+93]. FLASH and TYPHOON use handlers to support shared-memory coherence operations.

Both systems have been influenced by the work of Henry and Joerg [HJ92]. Henry and Joerg dramatically increased handler performance by adding fast *dispatch hardware* to a commercial processor. Dispatch hardware decodes the type of an incoming message and calculates the address of the handler to be executed. TYPHOON directly applies this lesson by using a SPARC integer core augmented with fast dispatch hardware. Interestingly, FLASH opted for a custom processor augmented with dispatch hardware.

The use of a custom processor in FLASH is inadequately justified. They claim four major advantages of the MAGIC chip:

1. Fast message dispatch via hardware support.
2. Speculative memory operations, initiated during dispatch, which attempt to fetch data before it is used by the processor.
3. Special instructions for common protocol operations.
4. Separate hardware for the mechanics of outgoing sends.

Three out of four of these advantages can be attained without a custom processor. TYPHOON attains 1 and 4, and could be extended to attain 2. Only the use of special instructions requires a custom processor.

The use of a custom processor results in three major disadvantages. First, the processor must be designed, built, and tested. Second, assemblers, debuggers, compilers, and other software must be written for the processor. Third, MAGIC will be less robust than a commercial microprocessor because it lacks the ability to detect resource conflicts and to enforce pipeline interlocks. This lack of error hardware is really a result of the first disadvantage. The error hardware was left out to save design and test time. The FLASH paper cites the lack of error hardware as a factor in not allowing user-code to run on MAGIC. Only system-level code, which is guaranteed not to cause unsupported errors, can run on MAGIC.

TYPHOON avoids all of these disadvantages with its SPARC core. The benefits of the special instructions in MAGIC have not been quantified and are unlikely to offset the disadvantages of a custom processor [3].

4 Protection

All three systems emphasize the need for protection to support a multiprogrammed environment. This section discusses two major protection issues. First, a user process must relinquish network resources in a timely manner so that the other processes may use the network. Second, different processes may need to efficiently communicate with each other.

4.1 Network Resources

The three systems ensure availability of network resources in one of two ways: either the messaging hardware only performs trusted functions or the messaging hardware and network are context switched along with the compute processor.

FLASH only allows system-level code to run on MAGIC. System code is written so that any functions MAGIC performs are guaranteed to relinquish network resources in a timely manner. SHRIMP does not run any code at all on its network interface. SHRIMP network functions are simple, hardwired, and designed to guarantee availability.

TYPHOON, however, allows user code to run on its protocol processor. If the user writes code which continuously sends data without terminating, then other processes may not be able to access the network. More importantly, if user handlers fail to consume messages from the network, the network will become clogged with unreceived messages.

To address this, TYPHOON adopts the solution used on the CM-5, which involves context switching the messaging hardware and the network when the compute processor switches to another process. All messages in the network are evacuated to temporary destinations and re-injected into the network when the user process is switched back in.

[3] At ISCA 94, Mark Horowitz pointed out that interfacing dispatch and DMA units with a SPARC core on TYPHOON is a substantial engineering effort. He argued that this effort was comparable to the design of MAGIC, especially given that MAGIC uses some base processor designs from the TORCH project.

Context switching of the network requires *strict gang scheduling*, where the same user process is executing on every processor of the machine. FLASH and SHRIMP are designed to allow arbitrary scheduling to support a *multiprogrammed* environment. Multiprogramming implies that different users and processes are executing simultaneously on different nodes of a multiprocessor. Neither paper motivates the need for multiprogramming, but some motivation can be found in the modeling community [ZB91] [MEB88] [LV90] [ZM90], the operating systems community [TG89], and the theory community [FSSHT91] [SWW91]. None of these studies is conclusive. As a whole, however, they suggest that there are better alternatives to strict gang scheduling.

All three systems miss an important alternative: the protocol processor can run user code as long as there are timeout mechanisms that ensure message consumption and availability of the protocol processor. If a user handler has not consumed a message in a specified amount of time, the *T multiprocessor [PBGB93] interrupts the handler, receives the next message, and either invokes the new handler or queues the message for later handling. The errant handler is restarted at a later time. *T does not have a separate protocol processor. On systems with separate protocol processors, this processor must be context switched to ensure availability. The switch can either occur along with every context switch of the compute processor, or only if a user handler executes longer than a specified time limit. Care must be taken to make sure the message consumption timeouts can keep up with the rate a user might be injecting messages into the network. The FUGU [MKAK94] system, however, suggests that message consumption need only allow the network to make enough progress to deschedule any process that is clogging the network. For example, an operating system on processor 0 may recognize that a particular process on processor 1 is sending too many messages to node 0. The system must be able to make enough progress for processor 0 to send a message to processor 1 which causes the offending process to be suspended for a while.

Among the mechanisms discussed, ensuring message consumption has the greatest impact. It provides protection for user-level handlers without requiring strict gang scheduling.[4]

4.2 Inter-Process Communication

Neither TYPHOON nor FLASH allow inter-process communication. Inter-process communication is important for remote procedure calls, multi-threaded applications, communicating with device drivers, and many other applications. SHRIMP addresses this problem directly with its `map` operation. However, as described later, `map` should be extended to operate on groups of processors rather than just pairs. Motivation for group communication is given by Kaashoek [Kaa92].

The SHRIMP paper presents the `map` operation with the following syntax:

```
map(send_buffer, destination, receive_buffer)
```

The types of the arguments are unclear. `send_buffer` and `receive_buffer` must include both starting address and length information. For clarity, let us specify `map` as follows:

```
map(char* send_buffer_address, int send_buffer_length,
    int destination_processor,
    char* receive_buffer_address, int receive_buffer_length)
```

`map` checks protection and establishes a mapping between the send buffer and the receive buffer. Data written into a send buffer is communicated to the receive buffer either automatically or at the user's request. However, this kind of pairwise mapping is inefficient for global or irregular applications. A global operation, such as sorting, would require mappings which provide all-to-all communication. Many irregular applications, such as adaptive multigrid, would also require all-to-all communication. To establish such links for n processors, n^2 map operations, n per processor, would be required. This would be very expensive, especially given that it requires n kernel crossings per processor. All-to-all mapping could be established for all processors, in one kernel call per processor, with a `global_map` function:

[4] Note that the protection mechanisms discussed only ensure availability, not fairness. Leiserson's Virtual Networks [Lei93] are a proposal for ensuring a fair distribution of network resources.

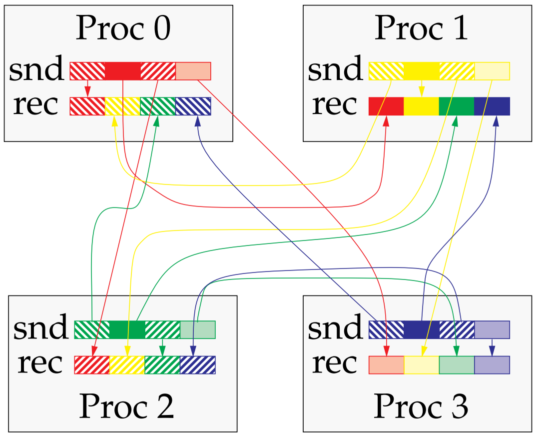

Figure 2: Global mapping for four processors.

```
global_map(char* send_buffer_address, int send_buffer_length,
           char* receive_buffer_address, int receive_buffer_length)
```

Where `send_buffer_address` is a pointer to n buffers of length `send_buffer_length` and n is the number of processors on the machine. The receive buffer is also a set of n buffers. The j-th send buffer on processor i is mapped to the i-th receive buffer on processor j. Figure 2 illustrates a global mapping of a set of send buffers to receive buffers for four processors. A more general function, `subset_map` could be specified as follows:

```
subset_map(char* send_buffer_address, int send_buffer_length,
           int start_processor, int end_processor
           char* receive_buffer_address, int receive_buffer_length)
```

Where there are n send buffers and n receive buffers and $n =$ `end_processor` - `start_processor`. The j-th send buffer on processor i is mapped to the receive buffer (i - `start_processor`) on processor (j + `start_processor`).

Figure 3 gives an example of `global_map` used in part of a sort. The routine sorts keys into buckets. There is one bucket per processor, but this code uses a receive buffer per processor for each bucket. In contrast, Figure 4 shows the sequence of `map` operations required to replace the `global_map` in Figure 3. The difference in kernel calls is clear.

This simple sorting example uses a lot of memory. Memory could be saved by overlapping all the receive buffers (discussed in alternatives below) and providing a synchronized method of calculating offsets into the buckets. However, SHRIMP may need to support mapping of smaller size buffers to keep memory requirements manageable for fine-grained group communication.

SHRIMP also uses `map` for intra-process communication. This is in contrast to the single shared-address space of both FLASH and TYPHOON. In a sense, those two machines already have a global mapping within processes. Intra-process communication makes it even more important for SHRIMP to support group mappings.

Extended maps are readily implemented on the SHRIMP architecture. `global_map` could be a cooperative operation where every processor must make the same call. The operation consists of a kernel call per processor to check protection, a global reduce operation to determine if protection passed on all processors, and then entering all the mappings into the page table on the network interface. The most significant savings is from reducing the number of kernel calls on each processor from n using `map` to one. Additionally, the reduce can be implemented with a tree which takes $O(\lg n)$ steps as opposed to n steps using `map`. `subset_map` could be implemented similarly, except on a subset of the processors. Other implementations of both functions could involve: a list of processors in a group rather than a consecutive range; mapping send buffers to overlapping receive

```
void nonlocal_sort(int *keys, int numkeys)
{
  int i, bucket;
  int send_buckets[num_procs][max_bucket_size];
  int recv_buckets[num_procs][max_bucket_size];
  int offset[num_procs];

  /* this only performs one kernel call */
  global_map(&send_buckets, max_bucket_size,
             &recv_buckets, max_bucket_size);

  /* initialize offsets into buckets */
  for (i = 0; i < numprocs; i++) offset[i] = 0;

  /* send appropriate keys to buckets */
  for (i = 0; i < numkeys; i++) {
    /* determine bucket number for this key */
    bucket = sort_bits(keys[i]);

    /* write to send buffer of bucket */
    send_buckets[bucket][offset[bucket]++] = keys[i];
  }
  /* send buffer data is either automatically or
    explicitly sent to receive buffers */
}
```

Figure 3: Using global mappings in a part of a sort.

```
  /* this performs numprocs kernel calls */
  for (dest_proc = 0; dest_proc < numprocs; dest_proc++)
    map(&send_buckets[dest_proc], max_bucket_size,
          dest_proc,
          &recv_buckets[proc_self_address], max_bucket_size);
```

Figure 4: Performing a global map using `map`.

buffers; an interrupting operation initiated by one processor instead of all processors; or storage savings by modifying the page table in the network interface to represent a group mapping with a single entry.

The SHRIMP project has not studied applications extensively. As a result, they fail to support applications which require well-connected group communication. Such applications are common and include sorting and irregular codes. Fortunately, the SHRIMP mechanisms are readily extended to support group communication.

5 Software Caching

To their credit, both FLASH and TYPHOON present convincing data that *software caching* strategies can increase the performance of shared-memory applications.[5] Software caching uses the main memory of a processor as a cache. Both studies show that software caching can increase performance when the working sets of applications will not fit in the hardware cache of a conventional protocol. They examined applications from the NAS [BBLS91] and SPLASH [SWG92] benchmark suites, as well as Cholesky factorization [RG93] and EM3D (described in Section 6).

The TYPHOON study compared their software caching protocol, *Stache*, to conventional $DIR_N NB$ hardware cache protocols. They found that *Stache* performed up to 25 percent better for data sets that do not fit in the hardware cache. The FLASH study found similar results and argues for even greater gains as processor speed increases relative to network latency.

These results are not surprising. Given the high cost of communication, it is better to manage a large working set in a software cache than to incur remote misses on a smaller hardware cache. The study of applications is reassuring, but the comparisons are limited to shared-memory implementations. The next section discusses alternative implementations.

6 Irregular Applications

The TYPHOON paper presents a study of EM3D, an irregular electromagnetics application originally used by Culler and others [CDG+93]. EM3D models electromagnetic wave propagation through three-dimensional objects. The computation involves an irregular bipartite graph. The graph consists of E nodes, representing electric field intensities, and H nodes, representing magnetic field intensities.

The TYPHOON study of EM3D makes three questionable conclusions. First, they claim high performance without costly run-time preprocessing. Second, they claim substantial gains through the use of a user-level, hybrid shared-memory protocol rather than a standard update protocol. Third, they claim that a user-level, hybrid protocol is easier to write than a message-passing program. The next three section deny each of these three claims.

6.1 Run-Time Preprocessing

Run-time preprocessing can result in good message-passing implementations of irregular applications [SCMB90]. In such applications, dependencies are generally not known at compile-time. Instead, dependencies are computed at run-time before the real computation begins. This run-time preprocessing step can also optimize the mapping of data and computation onto processors. Many applications have dependencies that do not change for long periods of computation. A good example of such applications are iterative computations. Because of this unchanging structure, preprocessing is worthwhile. In other words, the cost of preprocessing is amortized over the entire computation. In the remainder of this section, the discussion assumes an *owner-computes* model, where computations occur on the processor where the result is mapped. In other words, only data mappings are discussed, but these imply computation mappings.

Based on EM3D, TYPHOON claims high performance on irregular applications without preprocessing costs. There are, however, actually two aspects to preprocessing, *dependency preprocessing*

[5] The FLASH data is given in a separate technical report [CGSG93].

and *data remapping*. In dependency preprocessing, irregular dependencies may be examined so that data may be *sent*, requiring one message, rather than *fetched*, requiring two messages. In data remapping, data layout may be optimized to minimize communication resulting from non-local dependencies. The TYPHOON conclusion only applies to the dependency preprocessing.

The original description of EM3D in Culler's paper, however, reveals that the bipartite graph used in the TYPHOON study requires a *preprocessing* step to construct. This preprocessing step could easily include dependency preprocessing with little increase in cost.

When compared to *oblivious* distributions, the benefits of data remapping can be substantial. Oblivious distributions are the best that can be done at compile-time with no dependency information, and generally involve block or cyclic mappings. On irregular mesh computations similar to EM3D, preprocessing performed by the PARTI [DMS+92] software libraries resulted in factors of 2.5 to 3 in performance gains on the Intel iPSC/860 and Delta [Dun92] multiprocessors. On an architecture with more efficient communication, we found that preprocessing improved sparse triangular solve performance up to a factor of 1.5 on the Thinking Machines CM5 [CSBS94].

Note that caching strategies are not a substitute for data remapping. Cache strategies can move data to the processor which uses it. The data, however, is used wherever the relevant computation is mapped. Most shared-memory systems use owner-computes, which maps the computation to where the data is first placed by the compiler. Even without owner-computes, the compiler does not have the dependency information with which to map the computations intelligently. How can the compiler optimize for communications locality or minimize critical path times when communication and critical paths are entirely dependent upon run-time data? Such optimizations must be done at run-time.

6.2 Protocols

Figure 5 contains the code[6] used in the TYPHOON study of EM3D. EM3D is an iterated computation which results in repeated calls to the function `compute_E()`. In a conventional invalidate protocol, every call to `compute_E()` requires that every `e_node` be fetched, cached, and invalidated, resulting in at least four messages (request, response, invalidate and acknowledge).

The main result of TYPHOON is to use user-level shared memory to implement a delayed-update protocol which only requires one message per `e_node` per iteration. While conventional protocols can be implemented at system-level, application-specific protocols need user-level code to run on the protocol processor. The delayed-update protocol allows copies to become inconsistent within an iteration and only updates copies when the user makes an explicit update call. This update call replaces the `barrier()` in the code. The new protocol maintains presence counters which provide synchronization, removing the need for an explicit barrier. In fact, the new protocol is extremely similar in both operation and coding to an optimized message-passing implementation. This relationship is examined more closely in Section 6.3.

First, however, we examine EM3D under a conventional update protocol implemented at system-level, an alternative not presented by the TYPHOON study. Under an update protocol, the code in Figure 5 would result in an update for every iteration of the inner loop of `compute_E()`. However, the code in Figure 6 uses a temporary local variable to delay the update until the end of every iteration. Depending upon the implementation of the protocol, this code would require one or two messages per `e_node` per iteration (update and possibly acknowledge messages). The barrier, however, would still be required because no special presence counters are present in the standard update protocol. However, given a fast barrier implementation, we can get similar performance on EM3D without writing new protocols.

6.3 Message-Passing

The delayed-update protocol is extremely similar to an optimized message-passing implementation of EM3D. This is somewhat by design, since the protocol is meant to attain the same performance.

[6] An error in the dereferencing of `h_nodes` in the TYPHOON paper has been corrected in Figure 5.

```
typedef struct e_node {
   double value;
   int    edge_count;
   double *weights;
   double **h_node_values;
   struct e_node *next;
} e_node_t;

void compute_E()
{
   e_node_t *n;
   int i;

   for (n = e_nodes; n ≠ NULL; n = n→next)
      for (i = 0; i < n→edge_count; i++)
         n→value −= *(n→h_node_values[i]) * n→weights[i];

   barrier();
}
```

Figure 5: EM3D code used by TYPHOON

```
void compute_E()
{
   e_node_t *n;
   int i;
   double tmp;

   for (n = e_nodes; n ≠ NULL; n = n→next)
      tmp = n→value;
      for (i = 0; i < n→edge_count; i++)
         tmp −= *(n→h_node_values[i]) * n→weights[i];
      n→value = tmp;

   barrier();
}
```

Figure 6: Intelligent EM3D code for standard update protocols

Why not, however, just write a message-passing code? The TYPHOON study claims that the message-passing code is complex. However, this section argues that writing message-passing code can actually be simpler than writing both the shared-memory code and the delayed-update protocol. They also claim that preprocessing to set up the dependencies is expensive. These dependencies must be explicitly known to perform appropriate message passing, but Section 6.1 showed that EM3D already contains a preprocessing step which could calculate dependencies with minimal modification and expense. Finally, this section shows that the message-passing implementation uses an order of magnitude less memory and performs better than the delayed-update implementation.

Figure 7 contains the message-passing code for EM3D. The computation is producer, rather than consumer driven, so `compute_E()` now works on `h_nodes` rather than `e_nodes`. There is an extra field, `e_node_procs`, which will be filled in by preprocessing code and specifies the processor on which the `e_node_values` reside. The computation has been moved to a handler, `compute_handler`. The counter, `rec_count` provides synchronization and is similar to what the delay-update protocol must use. `CMAML_poll_while_lt` replaces the global barrier, synchronizing by polling for messages until the expected number have been received.[7] `CMAML_rpc` sends a message which invokes the handler on the remote processor.

Overall, this code is not much more complex than the shared-memory code in Figure 5. More importantly, the counters and handlers are simpler than the code which must be written for the delayed-update protocol. Writing a customized shared-memory protocol may not be difficult for the authors of the original TYPHOON system-level protocols, but getting such a protocol correct would be extremely difficult for the end user.

The encapsulation of message-passing mechanisms into a "user-level protocol" implies that the protocol is reusable for other applications. However, it is not clear that such protocols are reusable for multiple applications. If a protocol is broadly applicable, it should be added at system-level rather than coded by the user. The applicablility of user-level protocols needs more exploration. If users are expected to modify protocol code, a clear, simple interface must be provided which hides the complexity of coherence operations and which helps the users write correct protocols.

Consequently, simplicity is not a strong argument for the delayed-update implementation. Storage requirements, however, are a strong argument against it. All the shared-memory implementations must allocate memory to cache all the `e_node` and `h_node` copies. With no preprocessing, most of the dependencies will be to nodes on remote processors, requiring cached copies. The TYPHOON runs of EM3D used graphs with a degree of 15. This would result in nearly 15 copies of every node. The message-passing implementation only requires one copy of every node. The handlers compute on message data immediately and require no extra copies. Message-passing is the clear winner on applications where data can be immediately processed and consumed.

Additionally, all the shared-memory implementations require the `e_node` values to be computed in a strict order dictated by the inner `for` loop. This ordering is arbitrary and incurs unnecessary synchronization cost. The message-passing implementation performs the computations in the message handlers, avoiding unnecessary synchronization. My experience with sparse-matrix applications has shown that these synchronization costs can affect performance by as much as a factor of two.

A further source of performance is the interleaving of communication with computation in the message-passing code. In contrast, the delayed-update code performs all the communication in each iteration at once. Sending too much data at once fills up the buffering in the network and causes congestion. Brewer and Kuszmaul [BK94] have shown that the CM-5 network can buffer no more than about 10 messages per processor and that performance degrades severely if each processor has more than about 5 messages in the network. Such limits apply to any network.

The delayed-update code makes no attempt to aggregate messages going to the same processor. Even if aggregation were used, however, there would be no performance gain. Modern packet-based networks break long messages into a series of short messages. User-level message sends make sending a several short messages just as efficient as sending one long message. Given limited network buffering, sending short messages interleaved with computation is better than sending long messages of aggregated data.

[7] The CMAML functions are low level communications functions used by the CMMD message-passing library [Thi93b] on the CM-5.

```
typedef struct h_node {
   double value;
   int    edge_count;
   double *weights;
   double **e_node_values;
   int *e_node_procs;
   struct h_node *next;
} h_node_t;

int rec_count, rec_limit;

void compute_handler(double *enode_value, double value)
{
  rec_count++;
  *(enode_value) -= value;
}


void compute_E()
{
   h_node_t *h;
   int i;

   for (h = h_nodes; h ≠ NULL; h = h→next)
      for (i = 0; i < h→edge_count; i++)
         CMAML_rpc( h→e_node_procs[i], compute_handler,
                    h→e_node_values[i], h→value*h→weights[i]);

   CMAML_poll_while_lt(&rec_count, rec_limit);
}
```

Figure 7: Message-passing version of EM3D

Consequently, EM3D is a poor example[8] to support the TYPHOON conclusions. The message-passing implementation is no more complex than delayed-update, it uses an order of magnitude less memory, and it performs better.

7 Summary

This paper has examined performance, scalability, protection, and usability issues in the FLASH, TYPHOON, and SHRIMP multiprocessor designs.

A major conclusion is that the EM3D application presented by TYPHOON gives poor support for their conclusions. A message-passing implementation is no harder to write, performs better, and uses dramatically less memory than the delayed-update shared-memory implementation presented by TYPHOON. Although they do not emphasize it, TYPHOON would support such message-passing implementations well.

A contribution of this paper is the extension of the SHRIMP `map` operation to efficiently support protected group communication. These extensions are readily supported by the existing SHRIMP system. For fine-grained group communication, however, SHRIMP may need to support mapping of smaller buffers to keep memory requirements manageable.

Additionally, I refuted the premise that user-level handlers could not be run without strict gang-scheduling. The *T system is a counterexample to this premise.

On the positive side, FLASH and TYPHOON present convincing evidence that software caching can have significant benefits when applications have working sets too large to fit in hardware caches. This paper presented evidence that the separate messaging hardware of all three systems should result in significantly better applications performance through reduced processor overhead. However, I found inadequate justification for the use of a custom processor in the messaging hardware on FLASH.

8 Acknowledgments

The following people provided useful references or discussions relating to this paper: David Chaiken, Kirk Johnson, Steve Keckler, Kathy Knobe, Ken Mackenzie, Tom Simon, Carl Waldspurger, and Deborah Wallach. The critiques in this paper, however, are all my own ideas except where explicitly cited (e.g. the FUGU and *T work).

References

[BBLS91] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Technical Report RNR-91-002, Numerical Aerodynamic Simulation (NAS) Systems Division, NASA Ames Research Center, Moffett Field, CA 94035, 1991.

[BK94] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In *International Parallel Processing Symposium*, Cancun, Mexico, April 1994. To Appear.

[BLA+94] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, and Edward W. Felton. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, Chicago, Illinois, April 1994. To appear.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in split-c. In *Supercomputing '93*, pages 262–273. IEEE, 1993.

[CGSG93] Rohit Chandra, Kourosh Gharachorloo, Vijayaraghavan Soundararajan, and Anoop Gupta. Performance evaluation of hybrid hardware and software distributed shared memory protocols. Technical Report CSL-TR-93-597, Computer Systems Laboratory, Stanford University, December 1993.

[8] Better examples might be found in N-body applications [SHG92], but it is unclear that a worthwhile user-level shared-memory protocol exists for such applications.

[CSBS94] Frederic T. Chong, Shamik D. Sharma, Eric A. Brewer, and Joel Saltz. Multiprocessor runtime support for fine-grained, irregular DAGs. Submitted for publication, 1994.

[D+92] William J. Dally et al. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, pages 23–39, April 1992.

[DMS+92] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. Technical Report 92-12, ICASE, Hampton, Virginia, March 1992.

[Dun92] Thomas H. Dunigan. Performance of the Intel touchstone delta mesh. Technical Report ORNL/TM-11983, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, January 1992.

[FSSHT91] Anja Feldmann, Jiri Sgall, and Shang-Hua-Teng. Dynamic scheduling on parallel machines. In *IEEE 32nd Annual Symposium on Foundations of Computer Science*, pages 111–120, 1991.

[HJ92] Dana S. Henry and Christopher F. Joerg. A tightly-coupled processor-network interface. In *Proceedings of the fifth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, Boston, Massachusetts, October 1992.

[Kaa92] M. F. Kaashoek. *Group Communication in Distributed Systems.* PhD thesis, Vrije University, Amsterdam, 1992.

[KJA+93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 54–63, San Diego, California, May 1993.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta adn Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, Chicago, Illinois, April 1994. To appear.

[Lei93] Charles Leiserson. Talk given at Thinking Machines Corporation, Cambridge, Massachusetts., August 1993.

[LV90] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 226–236, May 1990.

[MEB88] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 104–112, May 1988.

[MKAK94] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. FUGU: Implementing protection and virtual memory in a multimodel, multiuser multiprocessor. Submitted for publication, 1994.

[PBGB93] Gregory M. Papadopoulos, G. Andy Boughton, Rober Greiner, and Michael J. Beckerle. *T: Integrated building blocks for parallel computing. In *Supercomputing '93*. IEEE, 1993.

[RG93] Edward Rothberg and Anoop Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing '93*, pages 503–512. IEEE, 1993.

[RLW94] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, Chicago, Illinois, April 1994. To appear.

[SCMB90] Joel H. Saltz, Kay Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, pages 303–312, 1990.

[SHG92] Jaswinder Pal Singh, John L. Hennessey, and Anoop Gupta. Implications of hierarchical n-body methods for multiprocessor architecture. Technical Report CSL-TR-92-506, Computer Systems Laboratory, Stanford University, 1992.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, 1992.

[SWW91] D. B. Schmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on-line. In *IEEE 32nd Annual Symposium on Foundations of Computer Science*, pages 131–140, 1991.

[TG89] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159–166, December 1989.

[Thi93a] Thinking Machines Corporation, Cambridge, MA. *CM-5 Technical Summary*, November 1993.

[Thi93b] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual (Version 3.0)*, May 1993.

[vCGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Eric Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, Queensland, Australia, May 1992.

[Wil92] John Wilkes. Hamlyn – an interface for sender-based communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, Operating Systems Research Department, November 1992.

[ZB91] Songnian Zhou and Timothy Brecht. Processor pool-based scheduling for large-scale NUMA multiprocessors. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 133–142, May 1991.

[ZM90] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 214–224, May 1990.