HUNTing the Overlap

Costin Iancu, Parry Husbands, Paul Hargrove
Computational Research Division, Lawrence Berkeley National Laboratory
{cciancu,pjrhusbands,phhargrove}@lbl.gov

Abstract

Hiding communication latency is an important optimization for parallel programs. Programmers or compilers achieve this by using non-blocking communication primitives and overlapping communication with computation or other communication operations. Using non-blocking communication raises two issues: performance and programmability. In terms of performance, optimizers need to find a good communication schedule and are sometimes constrained by lack of full application knowledge. In terms of programmability, efficiently managing non-blocking communication can prove cumbersome for complex applications. In this paper we present the design principles of HUNT, a runtime system designed to search and exploit some of the available overlap present at execution time in UPC programs. Using virtual memory support, our runtime implements demand-driven synchronization for data involved in communication operations. It also employs message decomposition and scheduling heuristics to transparently improve the non-blocking behavior of applications. We provide a user level implementation of HUNT on a variety of modern high performance computing systems. Results indicate that our approach is successful in finding some of the overlap available at execution time. While system and application characteristics influence performance, perhaps the determining factor is the time taken by the CPU to execute a signal handler. Demand driven synchronization at execution time eliminates the need for the explicit management of non-blocking communication. Besides increasing programmer productivity, this feature also simplifies compiler analysis for communication optimizations.

1 Introduction

Hiding communication latency is widely accepted as one of the most important optimizations in parallel programming [10, 11, 29]. Application writers and compilers for parallel languages attempt to achieve this by overlapping communication operations with other independent work in the application. They use non-blocking communication primitives and initiate transfers as early as possible so that the transfer time is hidden by other operations.

The performance gain of this explicit scheduling of communication operations is directly related to the optimizer's (either programmer or compiler) ability to identify work that can be overlapped with data transfers. For programs that do not contain any independent work between the transfer and the computation associated with it, decomposition transformations such as message strip-mining [17, 28] have been proposed.

Current manual and compiler communication optimizations are sometimes limited by the the inherently static nature of the approach: 1) data transfer sizes are usually not known or vary dynamically for each communication operation; 2) the manner in which the application uses the transferred data might be hard or impossible to determine due to factors such as the presence of indirect accesses (a[b[i]]) or third party library functions; and 3) when scheduling communication operations the optimizer many not have a good estimate of the amount of overlap available.

In addition, there are other practical considerations that limit the overlap achieved. Manual optimizations are very cumbersome to code and programmers typically only schedule transfers but do not apply the more sophisticated message strip mining techniques. Compilers are often limited by a lack of knowledge about the communication library (MPI), lack of knowledge about the code (third party libraries), or by the quality of the optimizer itself (lack of inter-procedural analysis support). All these factors indicate that parallel programs, even optimized ones, may still contain some amount of unexploited overlap.

In this paper we introduce HUNT, a runtime system designed to find and exploit some of the overlap available in parallel programs. The design principle of HUNT is based on the insight that in order to exploit all the overlap available in an application, communication operations should be retired (waited on) only at the program point where the application uses the data. Our runtime implements this principle by ignoring the explicit calls to retire transfers (i.e. the wait calls) present in the application and using virtual memory support to determine the dynamic program point (data access) where the communication operation should instead be retired. Thus, we delay retiring communication operations until the program point where the data is actually needed and achieve a form of demand-driven synchronization. In addition, HUNT can change the schedule of the communication operations.

HUNT is implemented at the user level as a runtime for UPC [5] programs. The implementation uses GASNet [8], a portable one-sided communication layer that supports a wide variety of contemporary high speed networks. Of most interest to us are Infiniband, Myrinet and Quadrics and we present results for several computing systems. Our results indicate that the proposed approach is indeed able to find and exploit overlap in application programs. Although system characteristics (network, processor, operating system) directly influence performance, perhaps the most important factor is the time taken to execute a signal handler on the target system.

Despite the fact that HUNT provides only runtime support for UPC programs, the findings in this paper are widely applicable. In particular, the same principles apply to any one-sided communication library (ARMCI [22], MPI-2) and also to programs using two-sided communication libraries (MPI-1).

2 Programmability and Portability: Unified Parallel C

UPC is a Partitioned Global Address Space (PGAS) language designed as a small set of extensions to ISO-C99. The language implements an SPMD programming model with two distinct address spaces. Each thread has access to a local address space and in addition, all threads share a common address space. The distinction between the two address spaces is made using the language type system. Thus, at the language level there is no syntactic difference between an access to local memory and an access to shared memory.

The goal of PGAS languages is to increase developer productivity by shifting the optimization burden from the programmer to the compiler and runtime system. Besides providing the abstraction of a shared address space that encourages a fine-grained style of programming, these languages also offer control over application performance using bulk communication primitives. In the UPC case, these are one-sided blocking communication primitives that take the form `upc_memget(dest,src,size)`, `upc_memput(dest,src,size)`. In order to assist compiler optimizations the language provides a memory consistency model.

In addition, the language provides a specification [4] for interfaces to a comprehensive set of collective communication functions. These primitives are also blocking, but allow programmer hints regarding the synchronization requirements of the primitive. The goal of this design is to provide compilers and library implementors with more information about optimization opportunities.

While neither Co-Array Fortran (CAF) nor UPC provide support for non-blocking communication in the original language specification, recent efforts [9, 13] attempt to retrofit this into the languages. The intention is to offer programmers better control over communication optimizations in the cases where the compiler is not able to apply them.

Using these primitives to construct optimal execution schedules is fraught with many difficulties. In addition to the considerations discussed in the introduction, programming languages do not provide many of the tools required to express and control the concurrent nature of programs that contain non-blocking operations. Furthermore, tuning such a schedule may be very system specific because of differing network characteristics. We consider the approach proposed in this paper as offering a compromise between application performance and programmer productivity. While it may not find the optimal execution schedule, our results show that our approach can non-intrusively add all the benefits of non-blocking behavior to applications running on a wide class of machines while preserving a common code base with the implementation for systems with a tight network/CPU integration. In addition, the demand driven synchronization we propose has the potential to greatly simplify the development of compiler communication optimizations.

3 The Berkeley UPC Compiler

Figure 1 shows the overall structure of the Berkeley UPC compiler used in our experiments. The translator performs source-to-source translation of UPC programs and targets a runtime responsible for bootstrapping tasks such as thread generation and shared data allocation. The runtime delegates communication operations, such as remote memory accesses, to the GASNet [8] communication layer which provides an uniform interface for one-sided communication primitives on a large class of contemporary high speed networks.

One-sided communication APIs provide communication primitives of the form *init_transfer; sync_transfer* with "all-or-nothing" data delivery semantics, i.e. the *sync_transfer* call blocks the CPU until the entire transfer is complete and the data written to main memory.

Figure 2 shows a UPC code snippet that contains communication and the corresponding translated code. Any UPC level communication primitive gets translated into a `transfer`/`sync` pair and the placement of the `sync` operation is under compiler control.

The `upc_memget` operation in the original code has a

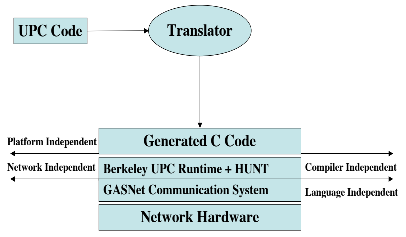

Figure 1. The Berkeley UPC Compiler

```
upc_memget(local, remote, size);
for(i=0; i < size; i++)
   ... = local[f(i)];
```

(a)

```
h = upcr_get_nb(local, remote, size);
// can't safely access [local, local+size]
upcr_wait_syncnb(h);
//can use [local, local+size]
for(i=0; i < size; i++)
   ... = local[f(i)];
```

(b)

Figure 2. **(a) UPC code sequence; (b) Translated C code**

latency that is determined by the network round trip time (RTT) and the network bandwidth. In the translated code, most of this latency can be overlapped with independent work placed between the `get/sync` communication calls. Hiding communication latency involves finding code in the original program that can be executed in between the `get/sync` pair without violating any data or control dependencies in the program. Subsequently, optimizations involve moving the initiation of a operation as far as possible from the use of data involved and explicit management of synchronization operations. Thus, any compiler analysis phase needs thus to take into account all possible execution paths.

Scheduling communication operations can exploit only the coarse grained (independent) overlap present in applications and it is often the case that in practice not enough independent work can be found and placed in between a `transfer/sync` pair of operations. However, considering a network transfer in conjuction with the computation on transferred data, uncovers an amount of fine-grained overlap present in any application. Message strip-mining [17, 28] is a program transformation that attempts to exploit such cases. This transformation decomposes transfers into a series of independent sub-transfers and modifies the structure of the associated computation to overlap a sub-transfer with computation on data previously received. Applying it requires detailed knowledge of the structure of the program and involves non-trivial source transformations. Furthermore, the transformed program is very likely to still contain parts where the processor is idle waiting for communication operations to finish.

4 Design Goals

In the ideal case, communication is performed in such a way that data items are transferred and become available in the order the application uses them rather than the sequential order imposed by the network. Furthermore, data items are delivered to the CPU for processing as soon as they arrive over the network. This functionality is clearly unattainable on most current systems and is approximated at the application level by message decomposition and pipelining techniques.

In the rest of the paper we describe the design principles behind HUNT, a runtime system that attempts to find and eliminate the idle times in a UPC program. HUNT is able to exploit both fine and coarse grained overlap by using a combination of automatic strip-mining and message scheduling. It opportunistically performs non-blocking communication operations and retires these operations only when the data involved is accessed by the CPU. For the automatic strip mining, the runtime provides a common unit of data transfer and synchronization between the two distinct subsystems: CPU-memory and NIC-memory.

The end result is that with HUNT, applications actively search for the unexploited overlap, either independent or fine-grained, available at execution time and schedule the operations found in order to hide communication latency.

The design goals are:

- Performance: 1) minimal communication management overhead and 2) preserving the performance of already well optimized applications.
- Programmability: 1) ability to transparently add non-blocking behavior to stock UPC programs and 2) a very simple set of interfaces to allow user control over runtime behavior and tuning.
- Portability: a set of simple heuristics to determine the values of the parameters that influence runtime performance combined with a minimality of the interface design.

5 Exploiting Find-Grained Overlap

The first goal of HUNT is to exploit the fine-grained overlap associated with a data transfer. This form of overlap is the hardest to exploit by programmers or compilers due the nature of the approach they are constrained to use.

We find and exploit fine-grained overlap by using a combination of two techniques: automatic strip mining and demand driven synchronization. Each data transfer is decomposed into strips and the transfer of each strip is performed in a non-blocking manner. Synchronization for each strip is demand driven. Instead of per message explicit synchronization calls, the system offers a mechanism for implicit synchronization at a smaller data granularity. Following a communication call, the first access to a data item triggers a synchronization (wait on) operation if the data has not been transfered.

The demand driven synchronization is achieved using virtual memory support. Operating systems allow programs to control the access rights to virtual memory pages using the `mprotect` system call. The access rights for a given page are either `NONE` or a combination of `READ, WRITE, EXEC`. Upon executing an instruction that accesses any memory location within a page with non-matching access rights, programs receive a `SIGSEGV`[1] signal. Thus, assuming that access to a page is restricted to `NONE` whenever a data transfer is initiated, the program is guaranteed to receive a `SIGSEGV` signal whenever it tries to access the data within the page. Whenever a program receives a signal, the operating system asynchronously executes the associated signal handler code (if any). In HUNT, receiving a `SIGSEGV` indicates that the program demands the data and we use the execution of the signal handler code to ensure that the corresponding transfer has indeed finished.

In the most general case, for the code in Figure 2(a), HUNT performs the following operations. The `upc_memget` call decomposes the transfer into strips, initiates non-blocking communication for the strips, records the decomposition and protects (`mprotect PROT_NONE`) the pages involved in the original transfer. The program will continue executing any instructions that do not access memory locations involved in the network transfer in parallel with the network transfers. Upon executing an instruction that accesses a memory location involved in the transfer, the program will receive a `SIGSEGV` and start executing the associated signal handler. The handler associates the faulting address with the sub-transfer that contains it, restores access rights and blocks execution until all the data has arrived. Then, program execution continues until references to data still outstanding are encountered and the process is repeated. By using this mechanism, computation is allowed to proceed as soon as data has arrived over the network and the transfers of subsequent parts of the message are transparently overlapped with the computation in the `for` loop.

It is obvious from this description that, for automatic strip mining, the granularity of interaction between CPU and NIC is constrained to the page level and for smaller transfers, HUNT should revert to the default modus operandi. In Section 6.1 we show that adding non-blocking behavior to small transfers is able to improve performance in the case where independent overlap is present in the program.

For each data transfer HUNT opportunistically adds overhead to the program execution in the hope that the extra work it performs will help reduce the communication latency and the overall execution time. It is therefore important to make sure that we keep the extra overhead to a minimum.

Before we present the techniques we use to minimize overhead, it is important to understand analytically what are the determining factors.

5.1 Performance Aspects of Automatic Strip Mining

Analyzing the behavior of automatic strip mining needs to take into account network performance, the performance of `mprotect`, the cost of servicing a signal, and the cost of manipulating the run-time meta-data associated with a transfer decomposition.

The LogGP [7, 12] network performance model approximates the cost of a data transfer as the sum of the costs incurred by the transfer in all system components. This is illustrated in Figure 3. Parameters of special relevance to us are o_s and o_r, the message send and receive overhead of a message; G, the inverse network bandwidth; and g, the minimal gap required between the transmission of two consecutive messages. According to the model, the cost of a single message transfer can be divided into two components, the software overhead on both the send and receive side ($o = o_s + o_r$ in the following), as well as the time the message actually spends in the network.

In addition, let $M(N)$ denote the cost of calling `mprotect` on a sequence of N elements and I denote the cost of servicing a signal. We assume that I is spent evenly between entering and exiting the signal handler. The cost of manipulating the message decomposition meta-data can be ignored for simplicity. It is pure software overhead for either recording a very small amount of data or looking it up. At any entry into HUNT, it is proportional with the number of outstanding network transfers. This choice is also validated by the experimental data.

Consider the code in Figure 2, and assume that no independent overlap exists and we need to exploit only the fine-grained overlap. The total execution time of the original code is $o + RTT + G * size + C(size)$, where $C(size)$ is the compute time on $size$ bytes of data.. Assume now that we apply automatic strip mining and we decompose the transfer into 2 smaller transfers, first one with size N_1, second one with size $size - N_1$.

Our first observation is that there is no computation available to be overlapped with the transfer of the first strip. This means that we can use this time for the message de-

[1] Or `SIGBUS` depending on the processor architecture and the operating system.

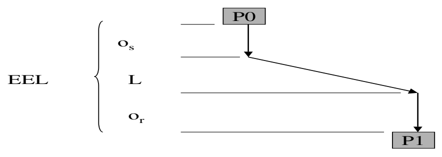

Figure 3. Traditional LogP model for sending a message from processor P0 to processor P1.

composition, changing the access rights, servicing half of the signal, and reverting access rights. This translates into $o+M(size)+\frac{I}{2}+M(N_1) < RTT+G*N_1$ and is independent of the computation to follow. This constraint directly determines the value of N_1 and implicitly determines the lower bound of the transfer size for which benefits can be expected.

In order to obtain any benefit from the optimization, the second requirement is that the combined latency of communication and computation on N_1 completely covers the latency of the second transfer and the additional overhead of the signal handler execution. This translates into $RTT+G*N_1+\frac{I}{2}+C(N_1) > RTT+G*(size-N_1)+\frac{I}{2}+\frac{I}{2}+M(size-N_1)$. While the other parameters are fixed or under our control (N_1), satisfying this constraint is determined mostly by the structure of the computation $C(N_1)$ which is determined by the application. However, network transfers place data only into the main memory and each computation following a transfer has a residual latency determined by the cost of moving the data from main memory to caches. This gives us a lower bound on the computation latency. We use this lower bound for the worst case scenario decomposition.

The values of the parameters considered in this analysis are system characteristics determined off-line and used to compute a lower bound on the transfer size. Transfers smaller than this value are not decomposed in HUNT.

5.2 Minimizing Overheads

In order to obtain good performance from automatic strip mining, the overhead of changing memory access rights and executing signal handlers needs to be minimized. Since these are system characteristics and we cannot improve their execution time, we try instead to minimize the number of interrupts and the size of the protected memory area.

In order to do this, we try to exploit the program structure and optimize for the case where transferred data is accessed using a monotonically increasing or decreasing sequence of addresses. This is indeed a very common case in scientific applications. Affine array index expressions trivially satisfy this property. In addition, Paek et al [23] examine a large class of scientific kernels and applications and present evidence that, except for ”subscripted-subscripts”, all non-affine subscript expressions present were compile-time provably monotonic.

For the sequential scan case, we do not need to protect the whole memory region involved in the transfer. We use two observations: 1) program execution starts processing a strip with its first (or last) page and 2) if indeed overlap is present in the application, at the time of performing the demand driven synchronization for a strip, some transfers for subsequent strips have already finished.

Based on these observations, we need to protect only a boundary page of any strip and not all the strip boundary pages need to be protected at once. We use a lazy protection technique that distributes the work evenly between message initiation and synchronization events and also decreases the number of signals that need to be serviced. At any time, we protect only the first page of a strip that program execution will access next. When the time to finish the transfer for that strip has arrived, the signal handler checks whether transfers for subsequent strips have finished. Any transfer that has already finished does not need any pages protected. The signal handler protects only the first page of the first transfer still outstanding that it is found.

For programs that access transferred data in a truly random manner, the implementation reverts to protecting everything. This changes the lower bound on the transfer size we consider for decomposition.

Most of the mechanisms described here are widely supported by modern operating systems. However, we have purposefully ignored so far the implied requirement that network interface cards are able to transfer data into protected memory regions. From the RMA networks we considered, Myrinet and Infiniband use physical addresses for the transfer targets, satisfy the requirement and all the mechanisms described can be implemented at user level. Quadrics uses virtual memory addresses and provides an on-board TLB that mirrors the processor TLB. A user level only implementation on Quadrics hardware aborts the program when transfers into protected areas are attempted. We can implement correctly our technique for sequential scan scenarios or scenarios where the access pattern of the program is communicated to HUNT by using a minor modification. Instead of protecting the first to be accessed page of an outstanding transfer, we protect the last accessed page of the previous transfer. This ensures that we can correctly determine a point where the program will soon start accessing data from a strip.

5.3 Message Scheduling

The message scheduling component of HUNT tries to exploit overlap whenever possible while preserving the performance of already optimized applications. It uses a set of simple heuristics to decide when to perform non-blocking communication operations and how to apply the automatic strip mining. The following are principles that we have found to work well in practice:

- For any blocking transfer, automatic strip mining should be attempted subject to the minimum size constraints presented in Section 6.1. HUNT supports two automatic modes for message decomposition: equal sized strips or variable sized strips. The principles and the trade-offs of these two strategies are described in [17].

- If there are any other outstanding transfers, use this as an indication that independent communication-communication overlap is available. Non-blocking communication is performed but without message decomposition (mprotect only). This case occurs in practice in program regions where one-to-many (scatter/gather) or all-to-all communication patterns occur. For a large number of outstanding transfers it is very likely that the latency of the latter transfers is completely hidden by the combined latency and computation of the previous transfers. Thus, for a "train" of outstanding messages performing the `mprotect` and servicing the signal handlers for the messages at end of the train is pure overhead. In this case, the implementation can choose to perform non-blocking communication without the mprotect and piggy-back the synchronization operations on the last expected demand-driven synchronization operation.

- For programs using non-blocking transfers (Berkeley UPC extensions), use this as an indication that independent overlap is already exploited in the program. In this case, we either do not attempt any decomposition or attempt a more aggressive (very large strips) decomposition. The size of the first strip is chosen to represent a significant percentage of the transfer and the handle returned by the operation is the one associated with the first strip. Synchronization for subsequent strips is demand driven.

5.4 Preserving Program Correctness

In HUNT, we change the semantics of communication operations and this might silently affect program correctness. Since we synchronize a transfer whenever the processor attempts to access data involved in it, correctness is maintained on each processor. Thus, we need only to preserve data consistency between processors.

In the general case, parallel programs usually use either barriers or point-to-point synchronization operations to enforce inter-processor data consistency. Thus, at any explicit synchronization operation HUNT needs to complete all the transfers whose behavior it had modified. Besides explicit synchronization operations, programs sometimes use point-to-point messages to implement various forms of synchronization. These operations typically follow a set pattern and HUNT can be trained to detect those and retire all outstanding transfers.

In the UPC case, the language has a memory consistency model that correct programs need to observe. The language provides for explicit synchronization through the use of barriers or locks. At any of these operations, HUNT retires all outstanding transfers. In addition, the language provides a form of consistency enforcing memory accesses (strict). In the UPC case, at any strict memory operations all transfers are retired. Due to the presence of the memory model, the runtime does not need to detect point-to-point synchronization patterns.

5.5 Implementation

HUNT is implemented as a change to the default Berkeley UPC runtime library. It inherits the bootstrapping functionality from the current Berkeley UPC runtime and it modifies all the communication functionality exposed to applications (e.g. `upc_memget`, `upc_memput`, `upc_barrier` ...) to implement the concepts presented throughout Section 5. All the described functionality is dynamic: the runtime maintains statistics about the communication behavior of the application and it uses these statistics to guide the decomposition and scheduling choices. Figures 4 and 5 show details about the implementation.

The implementation provides default values for all the parameters. For example, to guide message decomposition we implement the techniques described in [17]. Finer control over the functionality is exposed at the application level through a a simple interface. Application programmers can provide hints to guide the message decomposition and scheduling. For message decomposition, we provide interfaces to instruct the runtime to use a more aggressive decomposition (`set_decomp(factor)`) or to change the order of the strips transferred (`set_order(UP|DOWN)`). To control message scheduling we provide interfaces to instruct the runtime to start piggybacking after a certain number of transfers (`start_piggyback_thresh(num)`).

6 Evaluation

We have experimented with HUNT on a variety of production large scale parallel systems as well as on smaller experimental clusters. These systems are described in Table 1 and each system's characteristics influences the performance of our implementation in different ways. On the network side, the Quadrics network has a much shorter round-trip time than Infiniband or Myrinet, thus the overhead of message decomposition has a greater impact on overall performance. On the processor side, the PPC 970 processor has a signal service cost that significantly exceeds

```
upc_memget(.., size) {

     if (outstanding_transfers > TO) {
         - initiate non blocking communication (size)
         if ( need_mprotect())
             mprotect(...);
          else
             - mark for piggyback
     } else {
         - determine decomposition parameters
         - initiate all non  blocking communication
         - mprotect(...)
     }

     - try retire transfers that have already completed
     - update stats
}
```

Figure 4. upc_memget implementation

```
sigsegv_handle() {
     - identify transfer/strip for addr
     - retire strip and unprotect
      while  (more_strips()) {
          - check next strip completion
          if( !done) {
               mprotect(...)
               break;
          }
     }

     if(have_piggyback()) {
        - retire all completed transfers
     }

     - update stats
}
```

Figure 5. Signal handler implementation.

the network round-trip time. Thus, the message decomposition requires higher granularity and it is very important to minimize the number of signals serviced at run-time.

We use micro-benchmarks to illustrate the performance aspects of our implementation and validate these results with a series of application benchmarks: NAS [2, 15] (class B) FT, IS, MG. While the results presented are only for remote memory read operations, similar trends are observable for remote write operations. For write operations, the initiator typically blocks until it is safe to modify the source buffer. Protecting the memory of the source buffer with HUNT is used in this case and writes trigger segmentation faults.

The results presented for the AlphaServer and PowerPC systems use complete and correct user level implementations of HUNT. As explained in Section 8, we consider some of the results on Linux based systems as simulation.

6.1 Performance Aspects

Performance Parameters: As indicated in Section 5.1, the behavior of our implementation is influenced by the combination of network and processor performance parameters. Figures 6 and 7 present these parameters for the AlphaServer and the PPC/Infiniband systems. The processor performance parameters for the x86/Itanium processors all show similar trends to those of the Alpha processor, while the performance parameters of Myrinet are similar to those of the Infiniband network.

On most systems, the signal service time (I) is smaller than the network RTT. This indicates that even for very small transfers demand-driven synchronization has the potential to exploit for overlap a fraction of the RTT. On all systems, the transfer time increases with data size much faster than any other component and accordingly the message decomposition benefits will increase with the transfer size. This is illustrated in Figure 6.

Figure 7 shows the values of the performance parameters for small transfers on the PPC/Infiniband system. On this system, finer grained overlap is harder to achieve due to the much higher overhead of the demand-driven synchronization ($I >> RTT$).

Demand-driven Synchronization (DDS): The performance of demand-driven synchronization is determined by the relationship between the network latency (RTT) and the CPU signal overhead (I). We use two worst case scenario micro-benchmarks to illustrate this interaction and compare the performance of DDS with the performance of source level blocking and non-blocking communication. Table 2 shows our results.

The first benchmark applies DDS to an 8 byte data transfer and captures the overhead our method. On the Opteron/Itanium/PPC systems, this overhead is proportional with the speed of the processor signal handling. On the Opteron/Infiniband systems the overhead is smaller than the network round-trip time, while on the PPC system it is twice as high. On the AlphaServer system, the overhead of our method is much higher than the signal time. The Quadrics network contains an on-board TLB that is maintained coherent with the CPU TLB. We attribute the observed difference to the overhead of maintaining TLB coherency.

The second micro-benchmark performs two consecutive 8 byte transfers. Data involved in the transfer is located within the same page. On the Opteron and Itanium based systems, DDS is able to exploit some of the overlap available. The non-blocking implementation is still faster. This result indicates that even for the smallest transfers, when independent overlap is present in the application, our approach is able to find and exploit it. On the AlphaServer and PPC/Infiniband, the granularity of the overlap available is still smaller than the overhead of the implementation.

For very fine grained messages, due to TLB synchronizations, the overhead of our approach increases proportionally with the number of messages on Quadrics based systems. Note also that for a small number of transfers on the Quadrics networks it is not necessarily true that non-blocking communication improves performance in the fine

CPU/OS	Network	o (μs)	G ($\mu s/KB$)	RTT (μs)	I (μs)	`mprotect` (μs)
AMD Opteron 2.2GHz/Linux	4xInfiniband	1.8	1.23	11.6	2	1
Itanium2 1GHz/Linux [3]	Myrinet	3	4.7	25.5	3	1
PowerPC 970FX 2.3GHz/MacOSX [6]	4xInfiniband	5.2	1.26	20.2	37	1
AlphaServer ES45 1GHz/Tru64 [20]	Quadrics	2	3.96	8	5	1

Table 1. Systems Used for Benchmarks

	1 transfer (μs)			2 transfers (μs)		
	n-block	DDS	block	n-block	DDS	block
Opteron/Infiniband	11.67	15.37	11.57	20.68	22.36	23.42
Itanium/Myrinet	26.14	29.48	25.61	45.72	48.93	51.22
PPC/Infiniband	18.62	61.03	18.25	29.12	67.65	36.19
AlphaServer	7.78	43.48	7.55	15.83	52.65	14.93

Table 2. Performance of demand-driven synchronization.

grained case. For the pinning based networks, for processors with high interrupt time, our approach starts showing performance benefits for a train of small messages of roughly length $\frac{I}{RTT} + 1$. For the PPC/Infiniband system this threshold is around 5-6 transfers. In general, increasing the number of transfers provides better amortization of the DDS overhead and the relative performance benefits of our approach improve when compared with the non-blocking implementation.

Message Decomposition: The performance benefits of automatic strip mining are illustrated using a micro-benchmark that times the duration of a
memcpy() of transfered data. In a sense, this is a hard case to optimize for, since it contains only a small amount of computation. Figures 8 and 9 present our results. The line labeled *self-induced overlap* corresponds to a scenario where computation immediately follows the network transfer. The line labeled *independent overlap* corresponds to the case where an independent computation whose latency is large enough to hide the transfer latency has been inserted between the transfer and the memcpy operation.

On the systems with fast signal handling we observe a lower bound of the transfer sizes that benefit from decomposition in the range of $30KB - 64KB$. On systems with slow signal handling (PPC), this threshold has a value of $\approx 200KB$. The benefits of message decomposition increase with the transfer size. On all systems, for transfers in the MB range we observe speed-up in the $25\% - 35\%$ range.

On all systems, our technique is successful at finding and exploiting the independent overlap. In this case, the program has to service only one signal and the overhead of our technique is much smaller than transfer overheads.

We have also timed the scenario where transfers are perfectly overlapped using non-blocking communication primitives. For lack of space we do not show these results. On the systems with fast signal handling, the overhead of our technique is minimal and ranges from 1% of the total running time for the smaller transfers to changes in the third decimal digit for larger sizes.

Figure 9 shows the performance behavior on the AlphaServer system. On this system, besides the overhead introduced by maintaining the NIC TLB coherence which increases the decomposition size threshold, we observe a large variation in performance for the smaller transfer sizes that our performance models and previously presented micro-benchmarks do not account for. We think this behavior is caused by the system scheduler which might lower the priority of processes that service many signals in a short period of time.

6.2 Application Benchmarks

NAS FT and IS: These two benchmarks illustrate the case where both independent and self-induced overlap is present in the application. The main data redistribution step in both benchmarks consists of an one-to-many communication step, followed by either a Fourier transform (FT) or a histogram computation (IS). The released version of the benchmarks has communication implemented with blocking primitives (stock UPC).

Figure 10 shows performance results for the FT benchmark. We use as a base case the performance of the original implementation (FT) and normalize the performance of the other implementations. The series labeled FT-nb presents the performance of the benchmark where the blocking communication is replaced with non-blocking communication. For this benchmark source level strip mining is cumbersome and lowers significantly the performance of the serial code. Thus, this implementation represents the extent to which one can realistically expect a programmer to manually optimize the implementation without a complete rewriting and exploits only communication-communication overlap. The series labeled FT-md presents the performance with only message-decomposition enabled, while the series FT-H presents the performance of message decomposition combined with message scheduling.

Figure 11 shows the performance results for the IS benchmark. The series labeled IS-nb corresponds to an implementation where communication is non-blocking and the communication for the data received from one processor is

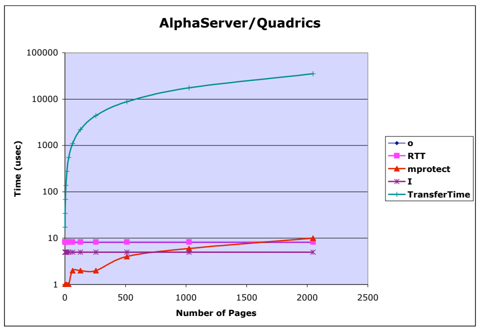

Figure 6. Asymptotic behavior of performance parameters

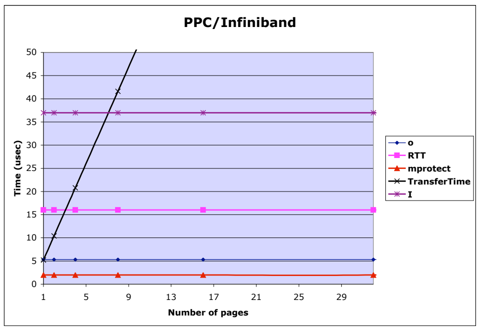

Figure 7. Performance parameters detail

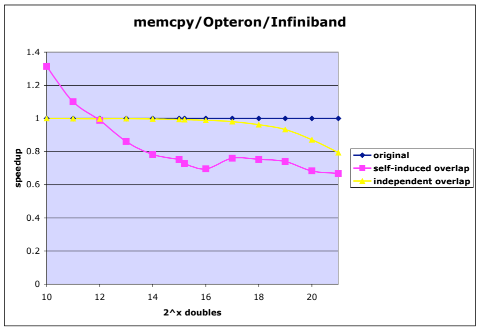

Figure 8. Automatic strip mining

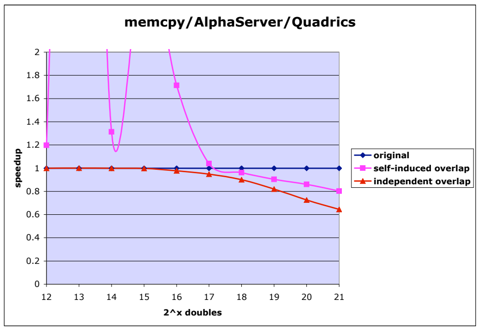

Figure 9. Automatic strip mining

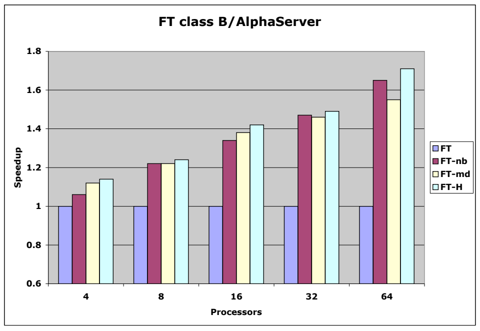

Figure 10. FT performance on AlphaServer/Quadrics.

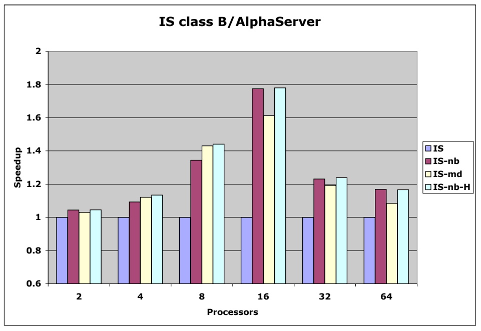

Figure 11. IS performance on AlphaServer/Quadrics.

overlapped with both communication and computation for the data from distinct processors. Again, this represents a realistic expectation of what a programmer might attempt. The series labeled IS-md shows the performance benefits when enabling only message decomposition. The series labeled IS-H shows the performance of hunting the non-blocking implementation of the benchmark.

HUNT is able to find and exploit the overlap present in the implementation of both benchmarks. Adding non-blocking behavior to the application improves performance by up to 77%. For the FT benchmark, HUNT outperforms the non-blocking implementation for up to 32 processors by up to 6%. Increasing the number of processors decreases the message sizes and the benefits of automatic strip mining while it increases the number of signals that need to be serviced in order to achieve correct synchronization. Enabling message scheduling lowers this overhead and our implementation outperforms the source-level non-blocking implementation even for higher processor counts. For the IS benchmark, IS-nb and IS-md show similar trends. Also, IS-H consistently outperforms the non-blocking version by a small amount.

Comparing the relative behavior for the two benchmarks, we note that our approach produces better results for the FT benchmark. This is explained by the differences in implementation. The implementation of IS exploits both communication-communication and communication-computation overlap. The FT implementation exploits only communication-communication overlap and thus exhibits a larger percentage of idle times.

User level hints were not used for any of the implementations presented here. Since both benchmarks perform the equivalent of a `gather` communication call, we do believe that further tuning of the application performance is possible using hints.

NAS MG: This benchmark uses a point-to-point communication pattern. The volume of transfered data varies with the program execution and the only applicable optimization is message strip-mining. Manual application of this technique is cumbersome due to the variation in message sizes and the structure of the computation that follows transfers. There are six distinct computation blocks immediately following a transfer operation, program control being able to choose any of them depending on the execution stage.

On the Opteron/Itanium systems the execution with HUNT enabled consistently outperforms the execution of the original program for the configurations measured (up to 64 processors). The performance benefits of HUNT in terms of end-to-end execution time are not as large for this benchmark when comparing with the benefits seen for FT and IS. MG spends only a small fraction of the total running time in communication operations, it transfers a much smaller volume of data (largest message size is $\approx 250KB$ versus $MByte$ messages in FT/IS) and performs a large number of transfers whose size is under the implementation threshold for automatic strip mining. The performance improvements for individual communication stages vary from 5% to 25% depending on the message size while the overall program performance improves by at most 1%. On the AlphaServer and PPC/Infiniband systems, according to our implementation heuristics, HUNT disables automatic strip mining for MG class B.

7 Programmability

HUNT exports all the interfaces required by the original UPC language specification and the non-blocking Berkeley UPC library extensions. In addition it exports a very simple set of single-argument functions to guide the heuristics involved, if so desired by the programmer. It has the ability to run unmodified UPC programs and transparently add non-blocking behavior to the original program thus implicitly finding the overlap wasted by the presence of blocking communication operations.

Explicit handle management for non-blocking communication can prove to be cumbersome and we believe that HUNT can increase programmer productivity by eliminating explicit communication management. In addition, it performs automatic strip mining, eliminating the need to perform it manually if it is not already performed by compilers (not supported or fails due to complicated code or third party libraries).

Since in our system no code reordering to increase the overlap potential is possible at run-time, manual code restructuring might be desired where compilers fail to perform it. In these cases, our run-time allows programmers to incrementally try different restructuring alternatives without the need to explicitly manage non-blocking communication. After determining the best implementation alternative, UPC programmers can change their code to fully use non-blocking communication if so desired.

To obtain best performance, HUNT might require programmer hints about the program access pattern on transferred data. When using third party libraries or complicated codes, this pattern might be hard to determine. We provide a diagnostic and feature extraction functionality to determine the page trace of data accesses after a transfer. We plan to extend this with a simple profiling infrastructure to assist programmers to decide if they need to switch to a fully non-blocking implementation.

8 Future Work

The user level implementation of demand-driven synchronization introduces unnecessary overhead due to the lack of proper OS level interface support and the lack of proper CPU/NIC integration. Kernel and driver modifications have the potential to eliminate parts of this overhead.

On TLB coherent networks, such as Quadrics, manipulating page access rights (`mprotect`) forces TLB update operations that increase the overhead of DDS. Unless the virtual-to-physical memory mapping changes, these updates are not necessary and network driver modifications might alleviate this problem.

The conclusions in this paper were drawn mostly from non-Linux systems because on all Linux based systems we investigated, user level DDS affects program correctness due to poor OS/NIC integration. Under Linux, calls to `mprotect` at any program point cause intermittent incorrect program behavior, even after the restoration of `READ/WRITE` access and even for blocking communication operations (no DDS). While our micro-benchmarks execute correctly under Linux, the NAS benchmarks suffer from verification failures for some processor configurations. This is a generic problem for all the Linux user-level network drivers we have examined: Infiniband, Quadrics patch-free and Myrinet. Quadrics networks function correctly with kernel patched drivers at the expense of extra overhead due to maintaining TLB coherence. We are currently investigating `mprotect` and driver implementations for pinning-based networks and believe we can achieve correct functionality with no extra overhead. Thus, we believe that our performance results are achievable under a correct Linux implementation.

HUNT is constrained to use standard operating system level interfaces and thus performs an unnecessary number of system calls. For example, the most common operation inside the signal handler is restoring access rights to one page while restricting the access to another. This currently requires two system calls that can be executed together with simple interface modifications.

The message scheduling heuristics in HUNT are designed to improve scalability and performance in the most general case and do require programmer hints for optimal performance. One-to-many and all-to-all library communication primitives already contain extra semantic information that can be used for message scheduling and we plan to investigate the implementation of such primitives. For example, the UPC collective operations, besides directly capturing the program communication pattern, contain as arguments programmer hints for the data synchronization requirements of the operation. Efficiently using these hints requires both run-time and compiler support. We believe that a HUNT implementation can make use of programmer hints and alleviate the need for compiler support.

9 Related Work

Wakatani and Wolfe [27, 28] introduce message strip-mining and analyze its impact for array redistribution in HPF and a code that implements a simple inspector-executor. Iancu et al. [17] examine message-strip mining for a variety of contemporary networks from the combined viewpoint of performance and portability. The idea of decomposing message traffic also appears in [24, 25] where it is transparently implemented inside the network layer in order to alleviate switch contention.

Using virtual memory support in parallel runtimes is an approach often used in practice. Dubnicki et al. [14] present the principles of virtual-memory-mapped-communication (VMMC). Their work explores software support for remote-direct-memory-access (RDMA) transfers and has since been supplanted by hardware RDMA support. Virtual memory services are widely used in the implementation of page based distributed-shared-memory (DSM) systems [21, 26] to enforce data coherence. More recent approaches [18] address the granularity problem of page based DSMs by using virtual memory support for multiple virtual mappings of the same physical page. Similar techniques might be applicable in HUNT based approaches to decrease the granularity of data interaction for demand-driven-synchronization.

Communication optimizations for parallel programs have been extensively studied [10, 11, 16, 19] both from the user level and compiler directed optimization point of view. All these studies indicate that non-blocking communication is able to increase program performance, especially under a one-sided communication model. For compiler optimizations, explicitly managing synchronization operations not only unnecessarily complicates the analysis but it also restricts the range of communication movement. A HUNT-like approach would eliminate most of these constraints and greatly simplify the analysis.

Intelligent run-time systems have received renewed interest in the last years and show very promising potential in terms of performance, scalability and programmer productivity. The approach most closely related to ours is present in the CHARM [1] runtime. Charm provides for latency hiding through an abstract execution model based on processor virtualization and message-driven execution. Charm decomposes the computation and achieves overlap by rescheduling threads that block for communication. In a sense, Charm schedules computations and our approach is orthogonal since it decomposes and schedules communication.

10 Conclusion

In this paper we explore the design and performance aspects of run-time techniques able to find and exploit the idle times caused by the presence of communication operations in parallel programs. Our runtime is implemented in user space and exploits overlap by using a combination of demand-driven synchronization, automatic message strip mining and message scheduling. Such an approach eliminates the need for explicit non-blocking communication and has the potential to increase programmer productivity, simplify the analysis required, and improve the quality of optimizations in compilers for parallel languages.

Performance is directly influenced by a combination of network and CPU performance characteristics and on all systems explored, the run-time is able to find and exploit "relatively large" granularity overlap. Exploiting very fine grained overlap is dependent on processor and network characteristics. On processors with a large signal service time ($I >> RTT$), our approach is not always beneficial for fine grained messages. Networks maintaining TLB coherency (Quadrics) also exhibit a large overhead for demand-driven synchronization, but driver level modifications might alleviate this problem. Very fine grained overlap can be efficiently exploited for a combination of pining-based networks (Infiniband, Myrinet) and processors with fast interrupts (Itanium, Opteron, x86). In order to further increase the performance of our approach, a tighter integration of CPU and NIC through kernel and driver level modifications is required.

We believe that intelligent run-time systems able to manage non-blocking communication on behalf of the application programmer or compilers are a necessity on future very-large scale parallel machines. Our work constitutes an initial evaluation of the design and performance aspects of such a runtime from a software only perspective. However, some of the principles presented capture the general nature of the problem and even hardware based implementations will have to implement them in some form. Thus, a software based approach like ours can be easily used to explore the design trade-offs of hardware implementations.

References

[1] CHARM++ project web page. Available at http://charm.cs.uiuc.edu.

[2] The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB.

[3] The Rice Terascale Cluster. http://support.rtc.rice.edu/.

[4] UPC Collective Operation Specification, v 1.0 . Available at http:/upc.lbl.gov/docs/system.

[5] UPC Language Specification, Version 1.0. Available at http://upc.gwu.edu.

[6] Virginia Tech Terrascale Computing Facility : SystemX. http://www.tcf.vt.edu/systemX.html.

[7] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[8] D. Bonachea. GASNet Specification, v1.1. Technical Report CSD-02-1207, University of California at Berkeley, October 2002.

[9] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National Laboratory, 2004.

[10] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global Communication Analysis and Optimization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, 1996.

[11] S. E. Choi and L. Snyder. Quantifying the Effects of Communication Optimizations. Technical Report TR-97-04-05, 1997.

[12] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[13] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A Multiplatform Co-array Fortran Compiler. In *Parallel Architecture and Compilation Techniques (PACT04)*, Antibes Juan-les-Pins, France, 2004.

[14] C. Dubnicki, L. Iftode, E. Felten, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of 10th International Parallel Processing Symposium*, April 1996.

[15] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In Proceedings of Supercomputing2002.

[16] M. Gupta, E. Schonberg, and H. Srinivasan. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, July 1996.

[17] C. Iancu, P. Husbands, and W. Chen. Message Strip Mining Heuristics for High Speed Networks. In the Proceedings of the Sixth International Meeting for High Performance Computing for Computational Science (VECPAR'04).

[18] A. Itzkovitz and A. Schuster. MultiView and Millipage - Fine-Grain Sharing in Page-Based DSMs. In *Operating Systems Design and Implementation*, pages 215–228, 1999.

[19] M. T. Kandemir, A. N. Choudhary, P. Banerjee, J. Ramanujam, and N. Shenoy. Minimizing Data and Synchronization Costs in One-Way Communication. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1232–1251, 2000.

[20] Lemieux. http://www.psc.edu/machines/tcs/lemieux.html.

[21] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–239, New York, NY, 1986. ACM Press.

[22] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science*, 1586:533–??, 1999.

[23] Y. Paek, J. Hoeflinger, and D. Padua. Simplification of Array Access Patterns for Compiler Optimizations. In *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*.

[24] L. Prylli, B. Tourancheau, and R. Westrelin. The Design for a High-Performance MPI Implementation on the Myrinet Network. In *PVM/MPI*, pages 223–230, 1999.

[25] L. Prylli, R. Westrelin, and B. Tourancheau. Modelling of a High-Speed Network to Maximize Throughput Performance: The Experience of BIP over Myrinet. In H. Arabnia, editor, *Parallel and Distributed Processing Techniques and Applications - PDPTA*, volume II, pages 341–349, 1998.

[26] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 325–337, 1994.

[27] A. Wakatani and M. Wolfe. A New Approach to Array Redistribution: Strip Mining Redistribution. In *Proceedings of PARLE'94 (Athen, Greece)*, Jul 1994.

[28] A. Wakatani and M. Wolfe. Effectiveness of Message Strip-Mining for Regular and Irregular Communication. In *PDCS (Las Vegas)*, Oct 1994.

[29] Y. Zhu and L. J. Hendren. Communication Optimizations for Parallel C Programs. *Journal of Parallel and Distributed Computing*, 58(2):301–332, 1999.