Efficient Address Translation

Matthias M. Müller

Institute for Program Structures and Data Organization,
Universitt Karlsruhe, Germany.
muellerm@ira.uka.de
Technical Report No. 2000-12

Abstract. The address calculation for distributed data access plays a major role for the performance of fine-grained data-parallel applications. This paper reports about the hardware *centrifuge* of the Cray T3E which enables the shift of the address calculation from software into hardware. This shift minimizes address calculation overhead reducing communication cost of dynamic communication patterns. The centrifuge is compared with complex integer division and modulo and with integer mask and shift operations. The measurements show for a one-dimensional dynamic communication pattern for several distributions a runtime advantage of T3E's hardware centrifuge of at least a factor 1.9 over integer division arithmetic. But, the centrifuge is barely faster compared with integer mask and shift operations.

1 Introduction

The address calculation for distributed data accesses plays a major role in fine-grained data-parallel applications. Many data-distributions has been proposed for different purposes and all of them come with a more or less complex calculation scheme. First of all, the usage of a special data-distribution depends on the supposed work-distribution among the processors. But if the locality of a data element cannot be determined efficiently, all the intended benefits of a data-distribution are meritless. Thus, the calculation of data-distribution information has to be fast to be effective. This paper studies block-cyclic distributions that can be computed with bitwise mask and shift operations. As these distributions can be processed by Cray T3E's hardware *centrifuge*, the main focus of this paper is the question about the benefits from using this hardware translation mechanism.
Scott says in [6] about the *centrifuge*

> The T3E supports the data distribution features of many implicit programming languages [he cites HPF, CRAFT, Fortran D and Vienna Fortran] via an integrated hardware *centrifuge*.

To my knowledge, nobody mentioned its benefits nor referred to its usage at all. I compare the address translation mechanism of the hardware centrifuge with complex division and modulo arithmetics and with bitwise mask and shift

operations. The first comparison shows the impact of time consuming arithmetics on the computation time while the second one explains the advantage of the hardware over fast integer manipulation.
This paper is part of the work done in the context of the HPF KarHPFn compiler [2, 3, 1] and latency hiding techniques [7, 4].
The organization of this paper is as follows. The next section explains the address translation for special block-cyclic distributions using mask and shift operations. Thereafter, basic E-register addressing and programming of the hardware centrifuge is explained. Afterwards, the basic communication technique used throughout the measurements is explained. The result section shows the performance of the different address translation mechanisms for BLOCK, CYCLIC and block-cyclic distributions.

2 Data Distributions

This section explains the structure of block-cyclic distributions with block size k. The section focuses, as the whole paper does, on distributions where the number of processors P, the block size k, and the local problem size[1] V are powers of two. Only then, it is possible to calculate the processor number and the local address with bitwise mask and shift operations.

2.1 Structure of an Address for Block-Cyclic Distributions

A global address for a block-cyclic distributed array with block size k consists of three fields, see Figure 1.

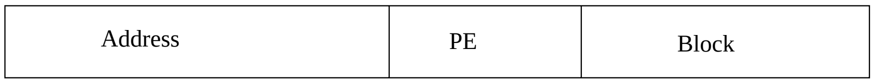

Fig. 1. Composition of a global address for a block-cyclic distribution

The rightmost field indicates the offset within a block of size k. The next field on the left hand side describes the processor number. The leftmost field contains the remaining bits to form the local address. The sizes of the fields are calculated using the logarithm with basis two.
For example, the following global address of a block-cyclic distribution with block size $k = 8$ and $P = 16$ processors

$$\boxed{001010}\boxed{0011}\boxed{010}$$

references the local address $\boxed{001010010} = 82$ on processor three.
BLOCK and CYCLIC distributions are special block-cyclic distributions. A BLOCK distribution has $k = V$ while a CYCLIC distribution sets $k = 1$.

[1] The local problem size is called *virtualization*, too.

2.2 Calculation with mask and shift operations

The address translation needs a mask to select those bits which form the processor number. In our example, the bit field

$$M = \boxed{000000}\boxed{1111}\boxed{000}$$

would be such a mask. The calculation of the processor number is done in two steps. First, the bits from the global address are selected using the mask. The second step shifts the mask $log_2(k)$ bits to the right. Hence, the processor number PE is calculated from a global address G with the following Fortran 90 commands

$$PE = ISHFT(AND(G, M), -log_2(k))$$

The local address A is formed from the two remaining fields. The first field consists of the offset within a block and the second one of the address. The offset is obtained with one mask operation while the remaining address needs two shift operations:

$$A = OR(ISHFT(ISHFT(G, -log_2(P) - log_2(k)), log_2(k)), AND(G, k-1))$$

The binary operations consume only a few processor cycles, and they are expected to improve communication time compared with integer DIV and MOD operations which are done in the floating point unit of the T3E's Alpha processors.

3 Hardware Centrifuge

This section presents T3E's hardware *centrifuge*. E-registers build the center for remote data-accesses within the T3E. Thus, discussion of the centrifuge starts with an overview of their functionality. Afterwards, programming of the E-registers is shown explaining address translation within the centrifuge. The remaining paragraph explain its initialization.

3.1 E-registers

The network interface consists of 512 user and 128 system E-registers, memory mapped into the address space of each processor. E-registers provide the only means to transfer data between processors. Reads and writes between E-registers and global memory are called *gets* and *puts*. To load a global memory content into the processor, a *get* and a subsequent read of the E-register has to be executed. The latter operation stalls the processor until the value arrives. This is achieved in hardware using the readiness state of the E-register. On a *put*, the memory of a remote node is modified and the cache is updated [5]. Hence,

the T3E implements a *global address space* with locally consistent memory. E-registers address this global address space which includes memory local to the issuing processor.
Eight E-registers can be combined to a vector. Distance between successive vector elements have to be equidistant to ensure correct address translation.

3.2 Programming the E-registers

E-registers are memory-mapped into the I/O-space of the processor. Therefore, every E-register command is a store into I/O-space:

```
E-register-command ( E-register-number ) = Index.
```

The left hand side of the assignment accounts for the operation and the selected E-register. Every pair of operation and register number points to a separate memory location. The right hand side provides the source of the operation which can be an arbitrary address of the global address space. The hardware *centrifuge* performs calculation of the node number and the local node address in two steps. For that purpose, it needs an additional block of four E-registers. A pointer in the upper half of the `Index` addresses this block, see Figure 2.

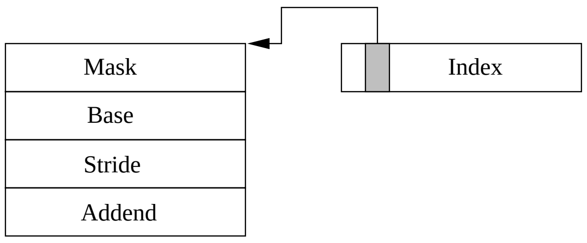

Fig. 2. E-registers for non-local data access

Every address translation needs the *Mask* and the *Base*. The *Stride* is used to calculate consecutive addresses in vector commands. Atomic operations like *fetch-and-add* use the *Addend* as additional parameter. The pointer is scratched out from *Index* after referencing the additional E-register block. The first step of the address calculation uses the *Mask* to select the virtual node number from the *Index*, see Figure 3.
The result of this step is the *Offset* with the selected bits scratched out and the virtual processor number *PE*. The second step adds *Offset* and *Base* forming the local virtual node address. Further transformations to physical addresses do not matter and are left out for brevity.
To move address calculation from soft- to hardware *Base* points to the local start of a distributed array. Consequently, the *Index* provides only the global

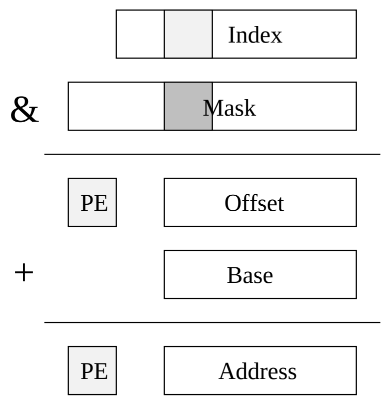

Fig. 3. Address calculation within the Hardware Centrifuge

index of the array needed for the calculation of the processor number *PE* and the local *Offset*. The *Mask* is initialized according to Section 3.3 to point to the bits significant for the processor number. Now, a data-parallel program that wants to perform address calculation in hardware sets up a separate block of four E-registers for each distributed array or it provides separate *Mask* and *Base* if the number of E-registers does not suffice.

3.3 Initializing the Hardware Centrifuge

The initialization is similar to Section 2.2. The *Mask* selects those bits from *Index* forming the virtual node number. For this purpose the *Mask* is divided into four segments, see figure 2. Their meaning is described supposing an arbitrary block-cyclic distribution with block size k. V denotes the local problem size in data elements and P accounts for the number of processors.

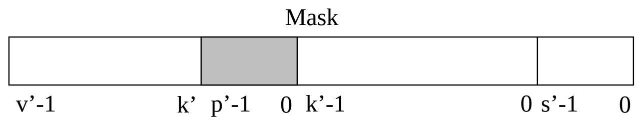

Fig. 4. Calculation of Mask from figure 2

The leftmost segment contains those bits responsible for a correct alignment of the appropriate data type. Its size in bits is $s' = log_2(sizeof(datatype))$. The following $k' = log_2(k)$ bits describe block size. These first $k' + s'$ bits are set to zero. The next $p' = log_2(P)$ bits are set because they select the bits forming the virtual node number. The remaining $log_2(V) - k'$ bits are reset again.

The results section show how far the hardware centrifuge improves runtime compared with software mask and shift operations.

4 Used Communication Technique: VSCAP

The measurements are done in the context of overlapping communication. The technique used is called VSCAP (software controlled access pipelining with vector commands) an extension to SCAP which was developed by Warschko [7]. The major aim of VSCAP is network latency hiding through overlapping of computation and communication by splitting non-local memory access into low overhead prefetch and access. The duration for issuing the prefetch instructions dominates communication time. Therefore, fast prefetch instructions caused by fast address calculation result in fast communication leading to a lower execution time of a data-parallel application. Hence, the goal of this section is to give a short overview about VSCAP to understand the communication technique used for the measurements.

4.1 Basic Idea of SCAP and its extension to VSCAP

The aim of SCAP is a runtime improvement achieved by overlapping several communication requests leading to a communication pipeline in fine-grained data-parallel applications. For a better understanding of the basic principle of SCAP, we first explain how communication is usually done.

The processor issues a request to the network (downwards arrow in Figure 5) and waits until the network replies (upwards arrow). Only then, the processor continues its execution and issues a new request. This is done as long as the processor requires remote data elements to perform its local part of computation. As the processor blocks after each data request, we call this kind of communication the *blocking mode*, see the upper half of Figure 5.

Now, let us assume the processor could issue all its communication requests and the network would be able to process them in an overlapped fashion. This would lead to a shorter waiting period for the processor accessing the first and all other successive remote data elements. Finally, communication could be performed faster compared with the above mentioned blocking execution. We call this kind of communication *overlapping communication*, see the lower half of Figure 5. To enable overlapping communication, the network interface has to provide a *prefetch buffer* that decouples the processor from the network execution. The second task of the prefetch buffer is to synchronize the processor with the network execution. The synchronization becomes necessary if the processor wants to access a data element which has not been delivered by the network yet. In this case, the processor is stalled until the value arrives.

VSCAP extends SCAP by the means of vector commands for prefetch and access. Instead of issuing a communication requests for each non-local data element, the processor prefetches and accesses $L > 1$ data elements at once. L is the vector length of the vector commands. VSCAP's vector commands reduce prefetch and access overhead of SCAP and improve communication time further.

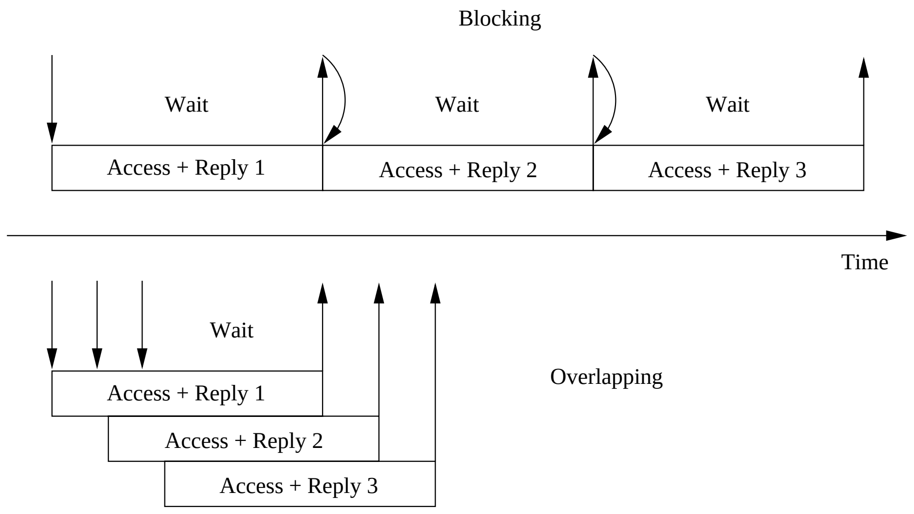

Fig. 5. Basic idea of SCAP

4.2 Transformation rules

This paragraph describes the techniques used in the transformation from a data-parallel forall-statement to VSCAP. The communication loop used in this section is used for the measurements.

The transformations are illustrated using the following simple forall-statement:

```
FORALL i = 0 TO N-1
  A(i) := B(q(i));
END
```

The program fragment updates array A in parallel, indexing array B with array q. A parallelizing compiler maps the problem size N onto P real processors ($N > P$). This technique is called *virtualization*. Assuming that P divides N each processor emulates $V = \frac{N}{P}$ virtual processors within a virtualization loop. Both A and B are distributed over the P processors using the the *owner-computes* rule. Since the value of $q(i)$ can not be determined at compile time, the compiler has to insert remote memory accesses. Each remote memory access causes an address translation from global to local addresses.

The following transformation of the loop shows the virtualization and how communication and computation can be overlapped:

```
FORALL j = 0 TO P-1
  FOR k=j*V TO (j+1)*V-1                // Prefetch loop
    adr := calculate_address(B(q(k)));  // Calculate remote address
    prefetch(adr);                      // Start read request
  END
  FOR k=j*V TO (j+1)*V-1 STEP L         // Vector access loop
    adr := calculate_address(B(q(k)));
    vector_access(adr,A(k));            // Access L data elements
  END
END
```

In this transformation, the main loop is split into two instances: a prefetch and an access (or calculation) loop. Instead of stalling on a remote memory access as in blocking mode, the processor issues remote memory prefetch requests. After the prefetch loop is executed the calculation loop accesses non-local memory without waiting time (if the data is already present) in the prefetch buffer. Due to the dynamic communication pattern, vector commands can only be used for access. Thus, the second loop is blocked with block size L. For simplicity, we assume that L divides V. Otherwise, additional element wise access operations has to be used to get the remaining $V\ mod\ L$ data elements which do not fill a vector of length L. Within the loop, the vector access `vector_access(adr,A(k))` copies L entries from the prefetch buffer starting at address `adr` to successive memory locations beginning with `A(k)`. If we assume that a vector access lasts as long as an element wise access operation, the duration of the vector access loop is decreased about the factor L of the vector length.
If the number of non-local memory accesses is too large to fit into the prefetch buffer, VSCAP's transformation rule uses a three loop execution pattern where the middle loop alternates between access and prefetch instructions. This transformation is not shown for the sake of brevity.
The measurements replace the call to `calculate_address` in the prefetch loop with integer division, integer mask and shift, or it is completely omitted in the case of the hardware centrifuge. On the Cray T3E, the call to the function `calculate_address` is not needed in the second loop. Therefore, manipulations take place only in the prefetch loop.

5 Results

The runtimes of *Indirect*, the example code shown in 4.2, for BLOCK, CYCLIC and block-cyclic distributions are given. The discussion of each distribution includes two plots. The first one shows the runtimes, and the second one presents the relative performance compared to an execution with integer division and modulo commands (DIVMOD). Each plot contains three different versions of *Indirect*: DIVMOD shows a VSCAP execution with integer division and modulo commands for address translation. MASKSHIFT uses integer mask and shift operations while HWC indicates an execution with the hardware centrifuge. DIVMOD and HWC are compiled by the KarHPFn compiler. MASKSHIFT was hand coded with the DIVMOD version as starting point. Tests were measured on 32 processors varying the local problem size V from 1 to 32768 vector elements. The Figure 6 shows the results for a BLOCK distribution.

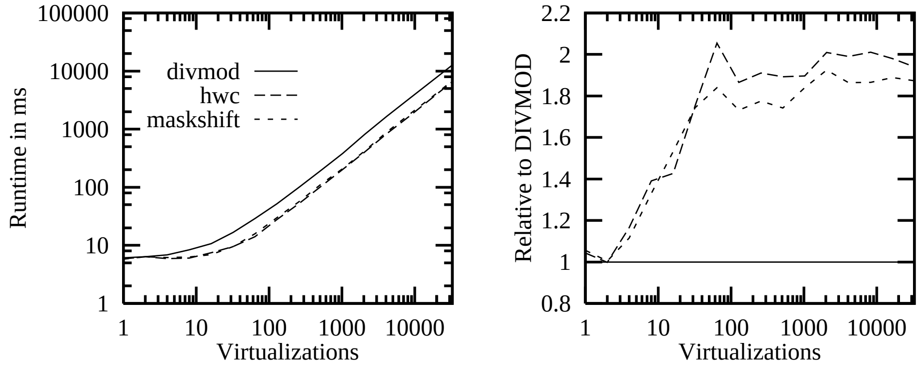

Fig. 6. Runtimes and speed for BLOCK distribution

The runtime plot on the left hand side shows two lines. The upper line belongs to DIVMOD. The lower line contains the runtimes for HWC and MASKSHIFT. The plot indicates two facts: First, integer division and modulo arithmetic is very slow compared with mask and shifts, and second, the hardware centrifuge HWC has only a minor advantage over MASKSHIFT. The plot on the left hand side of Figure 6 quantifies the advantage of HWC and MASKSHIFT over DIVMOD. HWC and DIVMOD are about 1.9 times and 1.8 times faster, respectively. The small difference between HWC and MASKSHIFT is astonishing. MASKSHIFT is at most 7% slower, although, compared with HWC it executes 3 additional integer operations for each non-local memory access. This behavior is due to the multiple integer units of the Alpha processors which overlap several operations. Another reason for this behavior could be a limited issue bandwidth for the E-register commands which is reached by the MASKSHIFT version. Then, faster prefetch operations, as the ones issued by the HWC version, would have no effect. But this is speculation beyond my scope.
A similar result shows Figure 7 for a CYCLIC distribution.
The plot on the left hand side shows the runtimes of the three different versions. And again, there are only two lines. The upper one explains the runtime of DIVMOD while the lower one denotes the runtime of MASKSHIFT and HWC. The CYCLIC distribution shows the same runtime behavior as the BLOCK distribution: MASKSHIFT and HWC are faster than DIVMOD and the former two versions are equally fast. The plot on the right hand side emphasizes these observations. MASKSHIFT and HWC are more than 2.1 times faster than DIVMOD and MASKSHIFT is as fast as HWC ($\pm 2\%$).
The results obtained so far are confirmed by the block-cyclic distribution, see Figure 8.

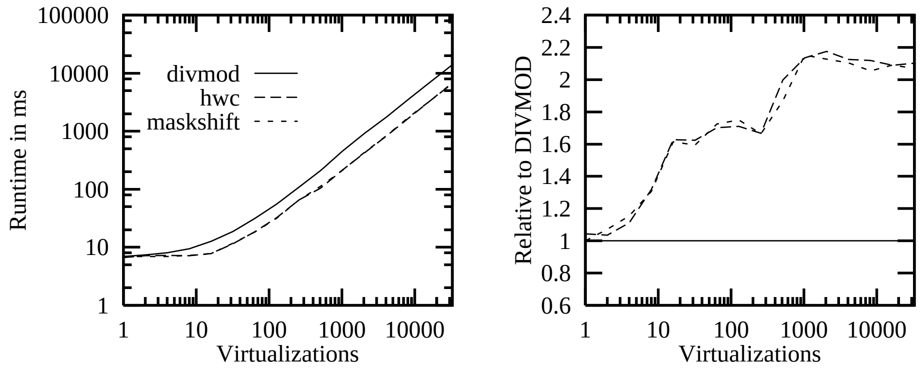

Fig. 7. Runtimes and speed up of *INDIRECT* for CYCLIC distribution

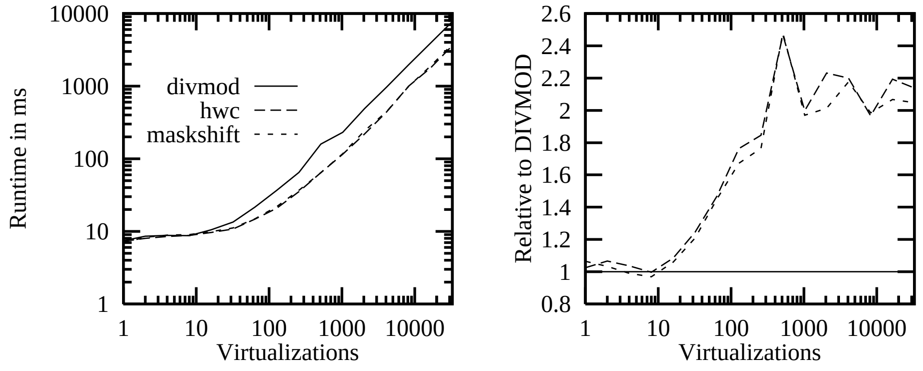

Fig. 8. Runtimes and speed up of *INDIRECT* for the block-cyclic distribution with block size $k = 8$

The runtimes of HWC and MASKSHIFT are equally fast and they show a substantial advantage over DIVMOD. The former two versions are at least 2 times faster than DIVMOD.
The measurements show two results. The first one confirms the expectation, that integer mask and shift operations for address calculations are faster than ordinary integer and modulo arithmetic. The second more astonishing result is the behavior of the fast integer operations compared with the hardware centrifuge.

6 Conclusions

This paper investigated the benefits from using Cray T3E's hardware *centrifuge*. The address translation mechanism was compared with complex integer division and modulo arithmetic and with integer mask and shift operations.
The hardware centrifuge is in a dynamic communication kernel about 1.9 times faster than integer division arithmetic. But, and this result is quite surprising,

it is only a few per cent (<7%) faster than integer mask and shift operations. This is caused by the multiple integer units provided by the Alpha processors which can overlap several integer operations.
The results show performance for one-dimensional arrays. The advantage of the hardware centrifuge would be a little larger if the measurements had focused on multidimensional arrays. Then, the software address calculation overhead would be larger leading to a more significant advantage of the centrifuge.
As this work emphasizes support for fast address calculation, it also shows the weakness of T3E's hardware centrifuge in doing this job for one-dimensional arrays compared with software mask and shift operations.

References

1. Matthias M. Müller. Compiling Applications with the KarHPFn Compiler. Technical Report 2000-11, School of Computer Science, Universität Karlsruhe, April 2000.
2. Matthias M. Müller. KaHPF: Compiler generated Data Prefetching for HPF. In *High Performance Computing in Science and Engineering 1999*, pages 474–482. Springer, 2000.
3. Matthias M. Müller. *Latenzzeitverbergung in datenparallelen Sprachen.* PhD thesis, School of Computer Science, Universität Karlsruhe, February 2000.
4. Matthias M. Müller, Thomas M. Warschko, and Walter F. Tichy. Prefetching on the Cray-T3E. In *12th International Conference on Supercomputing*, pages 368–375, Melbourne, July 13–17, 1998.
5. Wilfried Oed. Massiv-paralleles Prozessorsystem CRAY T3E. Technical report, Cray Research GmbH, München, November 1996.
6. Steven L. Scott. Synchronization and communication in the T3E multiprocessor. *ACM SIGPLAN Notices*, 31(9):26–36, September 1996.
7. Thomas M. Warschko. *Effiziente Kommunikation in Parallelrechnerarchitekturen.* PhD thesis, School of Computer Science, Universität Karlsruhe, 1997.