Performance Optimization

John Urbanic
urbanic@psc.edu

April 11-12, 2002

PITTSBURGH
SUPERCOMPUTING
CENTER

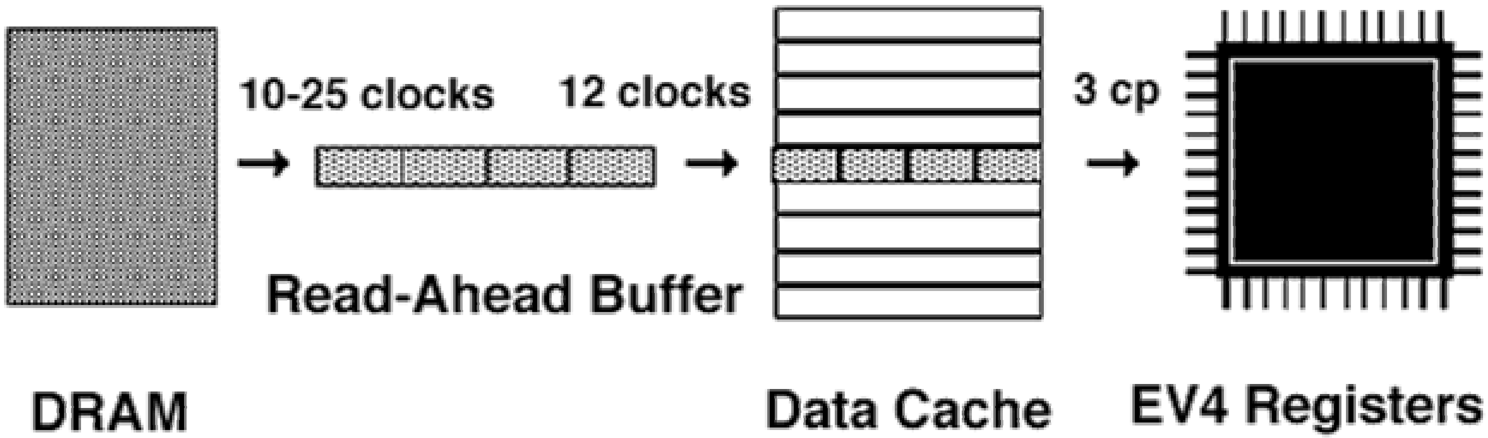
T3D Architecture

Peak Performance Data

- 150 MHz Alpha EV4 (21064)
- 150 MFLOP/s
- 1.2 Gbyte/s BW from DCACHE
- 320 Mbyte/s from DRAM

T3D Architecture

Data Stream



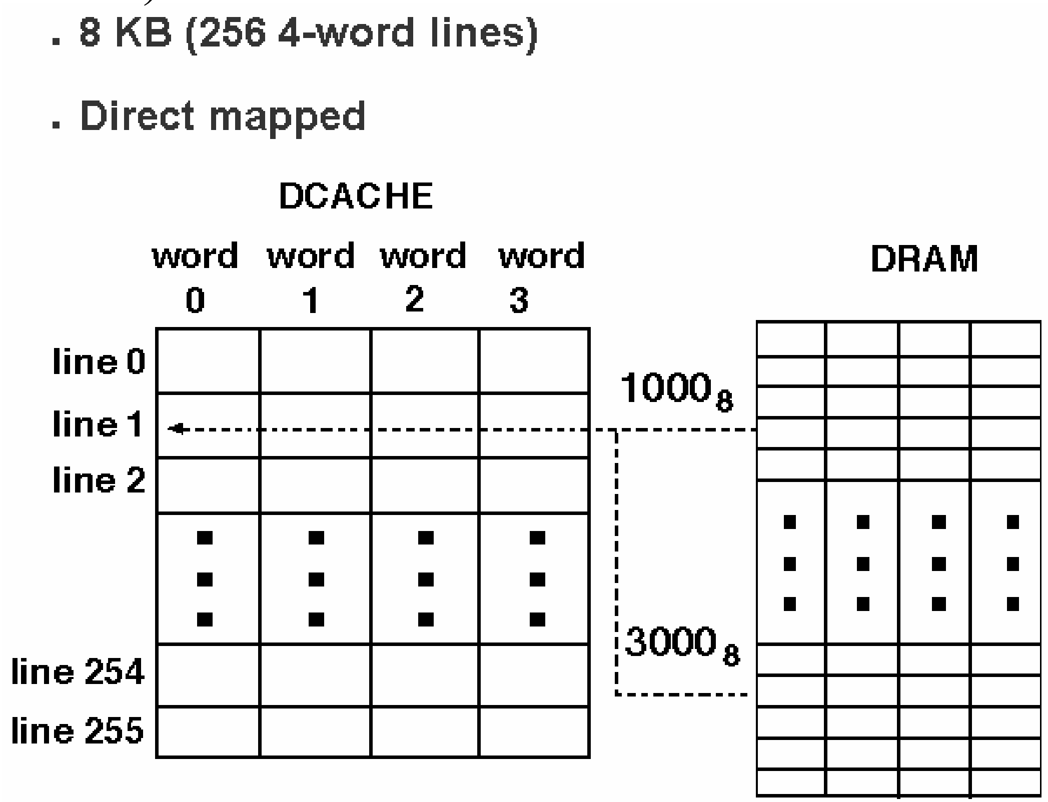
T3D Architecture

Real Performance Data

	Clocks	MBytes/s
Page Hits	27	177
Page Miss	42	114
Read Ahead	15	320

T3D Architecture

Data Cache
•8 KB (256 4-word lines)
•Direct mapped



Memory Writes

```
A (8192, 5)

DO I = 1, 8192

        A(I, 1) = 0.0
        A(I, 2) = 0.0
        A(I, 3) = 0.0
        A(I, 4) = 0.0
        A(I, 5) = 0.0
   ENDDO
```

The loop was coded for 1 to 5 output streams with the following results:

Memory Writes

Number of Streams	Clocks per Word	MBytes/sec
1	2.6	462
2	7.2	167
3	7.9	152
4	29.5	41
5	28.7	42

Note that theoretical peak for the write operation is one cache line per 9 clock periods. This equates to 533 Mbytes/sec. Jim Schwarzmeier has measured over 500 Mbytes/sec with a better scheduled loop.

Example: QCD

Generally, QCD codes spend the majority of their time in 3x3 matrix multiplications. On parallel vector processors (PVP), this is usually vectorized across multiple matrices with excellent resulting performance. On the T3D, results are less than optimal at about 3.5 Mflops. In the following code fragments, 2 Mflop figures are given. The first is without read ahead mode enable and the second is with read ahead.

PVP code – 3.5 Mflops/3.5 Mflops

```
COMMON/XXX/ A(1024,3,3), B(1024,3,3), C(1024,3,3)
CALL MM3V0(A,B,C,1024)
...
SUBROUTINE MM3V0(A,B,C,N)
REAL A(N,3,3),B(N,3,3),C(N,3,3)
 DO I=1,3
     DO K=1,3
          DO L=1,N
          C(L,I,K)=A(L,I,1)*B(L,1,K)
&              +A(L,I,2)*B(L,2,K)
&              +A(L,I,3)*B(L,3,K)
          ENDDO
     ENDDO
 ENDDO
 RETURN
 END
```

As written for the PVP systems, this code fragment has 6 distinct input streams for reading the matrices. This is because the first dimension of A and B is the matrix number. There is only one output stream so writes should be pretty well optimized.

As a first step, it makes sense to reverse the sense of the array and loops. Instead of working on vectors of 3x3 matrix multiplies, we work on one 3x3 matrix multiply at a time. Also reversing the array indices as x(3,3,n) will mean that the 9 elements of each matrix will be contiguous in memory, allowing for the opportunity to reduce the number of input streams to 2 (one each for the a and b matrices).

PVP code – 3.5 Mflops/3.5 Mflops

Because there is space allocated for 1024 3x3 matrices for each of a, b, and c, we will have direct map cache conflicts between any particular 3x3 matrix multiply problem. To alleviate this, we place pad arrays between the two arrays that are read. We choose size 512 for the pad array since this is 1/2 the size of the data cache and we have 2 arrays. Note that EV4 does not put values of c in the data cache since c is write only. If c had appeared on both the left and right hand sides of the equal sign, we would have had to worry about the data cache for c as well and chosen a different padding strategy.

Stream Reduction 15.7 Mflops/16.9 Mflops

```
 COMMON/XXX/ A(3,3,1024), PAD1(512),
&         B(3,3,1024), C(3,3,1024)
 CALL MM3V1 (A,B,C,1024)
 ...
 SUBROUTINE MM3V1 (A,B,C,N)
 REAL A(3,3,N),B(3,3,N),C(3,3,N)

 DO L=1,N
     DO I=1,3
          DO K=1,3
          C(I,K,L)=A(I,1,L)*B(1,K,L)
&                   +A(I,2,L)*B(2,K,L)
&                   +A(I,3,L)*B(3,K,L)
          ENDDO
      ENDDO
 ENDDO
 RETURN
 END
```

In this construct, however, matrix c is no longer a stride-1 write. This will cause problems with the write buffers. To alleviate this, we can unroll the i loop which is the first dimension of c. In addition, unrolling will expose more re-use to the compiler and three elements of b can be held in registers.

Unroll I 23.8 Mflops/27.3 Mflops

```
 COMMON/XXX/ A(3,3,1024), PAD1(512),
&            B(3,3,1024), C(3,3,1024)

 CALL MM3V2(A,B,C,1024)
 ...
 SUBROUTINE MM3V2(A,B,C,N)
 REAL A(3,3,N),B(3,3,N),C(3,3,N)

 DO L=1,N
     DO K=1,3
          C(1,K,L)=A(1,1,L)*B(1,K,L)
&                 +A(1,2,L)*B(2,K,L)
&                 +A(1,3,L)*B(3,K,L)
          C(2,K,L)=A(2,1,L)*B(1,K,L)
&                 +A(2,2,L)*B(2,K,L)
&                 +A(2,3,L)*B(3,K,L)
          C(3,K,L)=A(3,1,L)*B(1,K,L)
&                 +A(3,2,L)*B(2,K,L)
&                 +A(3,3,L)*B(3,K,L)
     ENDDO
 ENDDO

 RETURN
 END
```

Now we can unroll the k loop as well. The 9 elements of a and the 9 elements of b are fully exposed to the compiler and can be held in registers for the calculations in the loop.

Unroll K and I 26.9 Mflops/27.5 Mflops

```
 COMMON/XXX/ A(3,3,1024), PAD1 (512),
&           B(3,3,1024), C(3,3,1024)
 CALL MM3V3(A,B,C,1024)
 ...
 SUBROUTINE MM3V3(A,B,C,N)
 REAL A(3,3,N),B(3,3,N)C(3,3,N)
 DO L=1,N
     C(1,1,L)=A(1,1,L)*B(1,1,L)
&           +A(1,2,L)*B(2,1,L)
&           +A(1,3,L)*B(3,1,L)
     C(2,1,L)=A(2,1,L)*B(1,1,L)
&           +A(2,2,L)*B(2,1,L)
&           +A(2,3,L)*B(3,1,L)
     C(3,1,L)=A(3,1,L)*B(1,1,L)
&           +A(3,2,L)*B(2,1,L)
&           +A(3,3,L)*B(3,1,L)
     C(1,2,L)=A(1,1,L)*B(1,2,L)
&           +A(1,2,L)*B(2,2,L)
&           +A(2,3,L)*B(3,2,L)
     C(2,2,L)=A(2,1,L)*B(1,2,L)
&           +A(2,2,L)*B(2,2,L)
&           +A(2,3,L)*B(3,2,L)
     C(3,2,L)=A(3,1,L)*B(1,2,L)
&           +A(3,2,L)*B(2,2,L)
&           +A(3,3,L)*B(3,2,L)
     C(1,3,L)=A(1,1,L)*B(1,3,L)
&           +A(1,2,L)*B(2,3,L)
&           +A(1,3,L)*B(3,3,L)
     C(2,3,L)=A(2,1,L)*B(1,3,L)
&           +A(2,2,L)*B(2,3,L)
&           +A(2,3,L)*B(3,3,L)
     C(3,3,L)=A(3,1,L)*B(1,3,L)
&           +A(3,2,L)*B(1,3,L)
            +A(3,3,L)*B(3,3,L)
 ENDDO
 RETURN
 END
```

Arithmetic Pipelines

The DEC EV4 processor has segmented functional units for floating point multiply and addition. Although a multiply or addition can be issued every clock period, the result is not ready for 6 clock periods. Thus, in order to get top performance from FORTRAN, the code must expose functional unit parallelism to the compiler.

Functional Time Units

Operation	Clocks	Pipeline?
FP Multiply	6 cp	Yes
FP Add	6 cp	Yes
FP Divide	61 cp	No

To see the effect of functional unit transit time, we test some simple loops on the CRAY T3D.

Arithmetic Pipelines

For example, the following loop does a single floating-point multiply on a scalar variable. The data for this loop can be completely held in registers by the compiler:

```
do i = 1, 1024
       t = t * t
enddo
```

We would expect a floating-point multiply result approximately every 6 clock periods. No functional unit pipelining is possible here, because the result is used in the next pass of the loop. One result per 6 clock periods equates to 25 Mflops on the CRAY T3D system. The measured result for this loop is 24.5 Mflops.

We would expect the following loop to do much better:

```
do i = 1, 1024
       t1 = t1 * t1
       t2 = t2 * t2
       t3 = t3 * t3
       t4 = t4 * t4
       t5 = t5 * t5
       t6 = t6 * t6
enddo
```

Arithmetic Pipelines

Indeed, measured performance for this loop is 110 Mflops. In this case, all 6 multiplies can fire one after the other and we can achieve near-peak performance.

Unrolling inner loops often exposes more functional unit parallelism to the compiler and can dramatically improve the results of loops. In the following example, arrays A, B, C, and D are all size 256 and so the operands can reside in cache. The loop is repeated many times to get a cache-resident performance figure:

```
DO I = 1, 256
     A(I) = B(I) + 2.0 * C(I) + D(I)
ENDDO
```

Arithmetic Pipelines

As written here, with no unrolling, we see about 18 Mflops. In this case, we need the result of the multiplication for a subsequent addition. Unrolling exposes much more functional unit parallelism to the compiler. Unrolling by 8 gives us 75 Mflops for the same loop:

```
DO I = 1, 256, 8
   A(I)    = B(I) +    2.0*C(I)    +D(I)


     A(I+1) = B(I+1) + 2.0*C(I+1) + D(I+1)
     A(I+2) = B(I+2) + 2.0*C(I+2) + D(I+2)
     A(I+3) = B(I+3) + 2.0*C(I+3) + D(I+3)
     A(I+4) = B(I+4) + 2.0*C(I+4) + D(I+4)
     A(I+5) = B(I+5) + 2.0*C(I+5) + D(I+5)
     A(I+6) = B(I+6) + 2.0*C(I+6) + D(I+6)
     A(I+7) = B(I+7) + 2.0*C(I+7) + D(I+7)
END DO
```

The CFT77 compiler will unroll simple inner do-loops by using the -vU option. An alternative is to use the fpp pre-processor to unroll loops with the unroll directive. The divide operation is not pipelined and so presents a special set of challenges. It is covered in detail in the next section.

Divide Operation

The divide operation is expensive at 61 clock periods. The divide unit is not pipelined, so it is not possible to issue a second divide while a first divide is in progress.

Generally, the best advice with divides is to try to avoid them whenever possible. The CFT77 compiler currently follows the IEEE rules, which state that a divide cannot be replaced with multiplication by a reciprocal (this may change in the future with a flag in CFT77 to ignore the IEEE rules).

In the following example, A, B, C, and D are all cache resident and we achieve about 9 Mflops.

```
cfpp$ unroll (8)
     DO I = 1, 256
        A(I) = (B(I) + 2.0 * C(I) + D(I)) / x
     ENDDO
```

Since this divide is loop invariant, we can simply multiply the reciprocal:

```
     xinv = 1.0 / x
cfpp$ unroll(8)
     DO I = 1, 256
        A(I) = (B(I) + 2.0 * C(I) + D(I)) * xinv
     ENDDO
```

Resulting code performance here is a little better at 93 Mflops!

Divide Operation

Other operations can proceed when a divide operation is in progress. If it's not possible to move a divide outside of a loop, it's sometimes possible to pre-schedule it from FORTRAN.

The following code segment is from Amber. This inner loop has a divide, and the result is immediately used. Initial performance of this loop is 20 Mflops.

```
      DO 1300 JN = 1, LPR
      J = IAR2(JN+LPAIR)
      IC = ICO(IACI+IAC(J))
      XW1    = tmp1-X(1,J)
      XW2    = tmp2-X(2,J)
      RWTMP  = tmp3-X(3,J)
      R2INV  = 1.0E0/(XW1**2+XW2**2+RWTMP**2)

c  problem is here.  Result of divide is used in next
c  calculation.  We wait about 60 clock periods.
      DF2 = CGI*CG(J)*R2INV
      EELT = EELT+DF2
      R6 = R2INV**3
      F1 = CN12(1,IC)*(R6*R6)
      F2 = CN12(2,IC)*R6
      ENBT = ENBT + (F2-F1)
      DF = (DF2+6.0E0*((F2-F1)-F1)*R2INV
      FW1 = XW1*DF
      FW2 = XW2*DF
      FW3 = RWTMP*DF
      F(1,J) = F(1,J)   +FW1
      F(2,J) = F(2,J)   +FW2
      F(3,J) = F(3,J)   +FW3

      tmp4 = tmp4 -FW1
      tmp5 = tmp5 -FW2
      tmp6 = tmp6 -FW3
1300    CONTINUE
```

Divide Operation

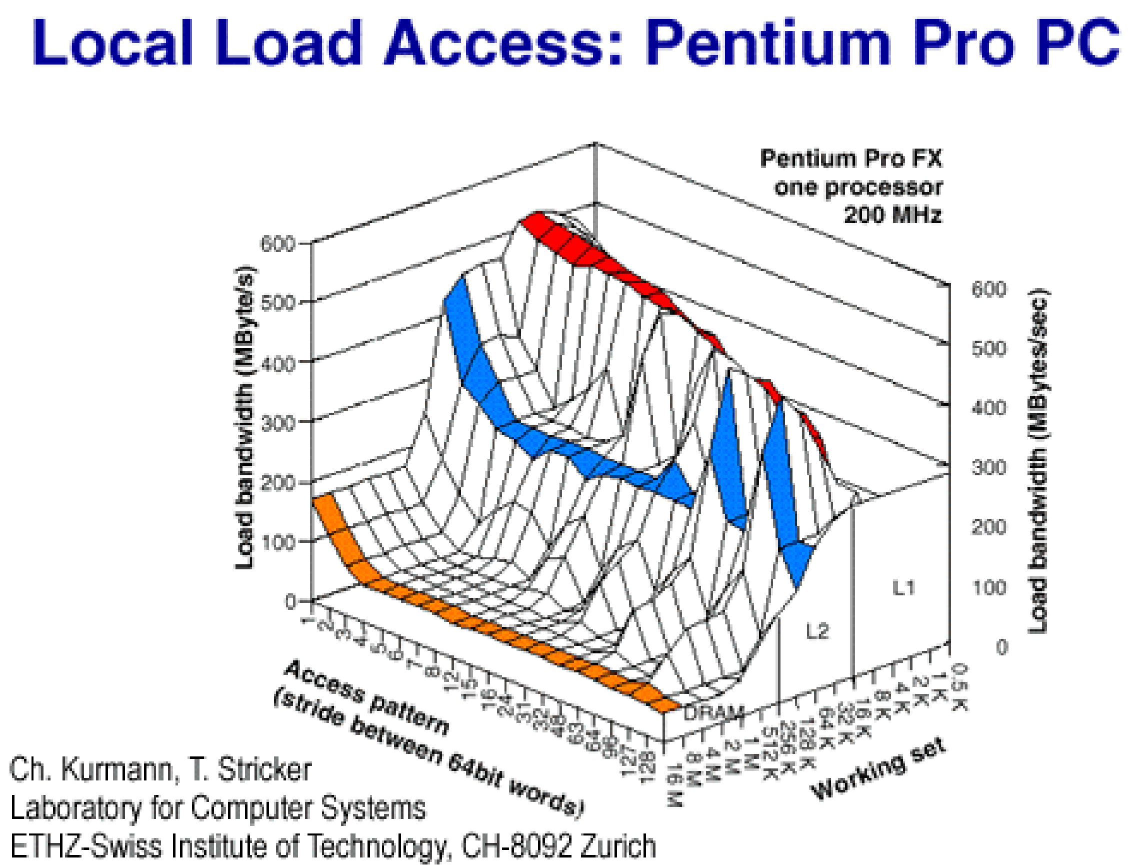
We can use a technique similar to bottom-loading where we compute the divide that is required for the next iteration of the loop in advance. The result of the divide is not needed until the next pass of the loop and hence the floating-point operations following the divide can overlap with the 61 clocks. This increases performance to 25 Mflops at the expense of nice-looking code:

Divide Operation

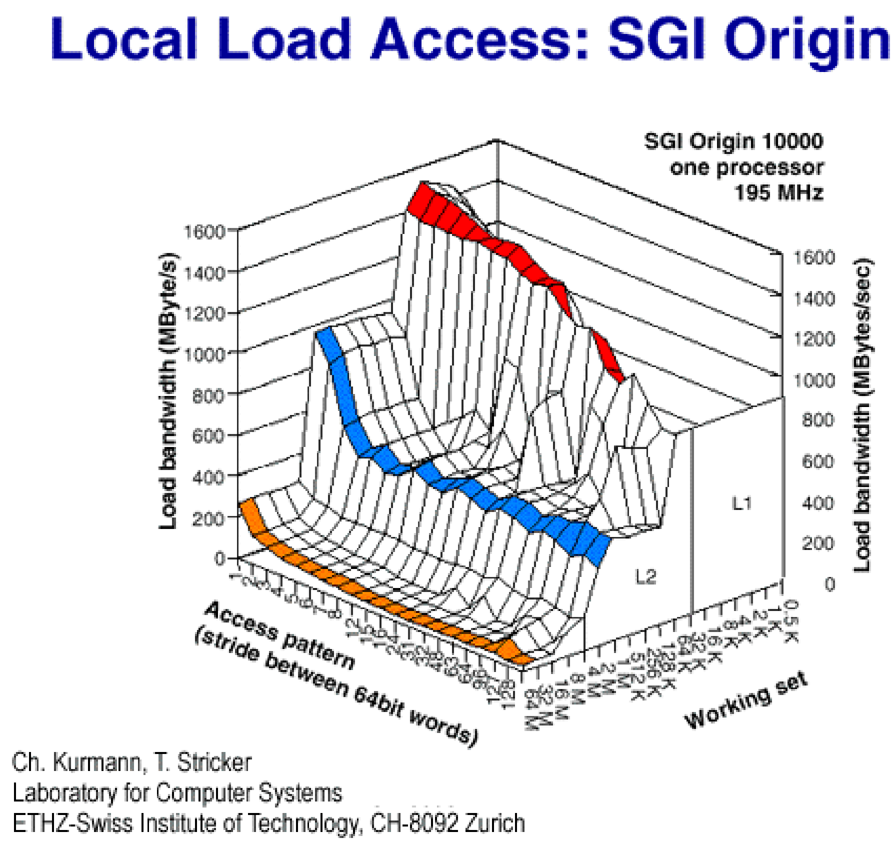
```
c    first divide computed
        J = IAR2(1+LPAIR)
        XW1     = tmp1-X(I,J)
        XW2     = tmp2-X(2,J)
        RWTMP   = tmp3-X(3,J)
        R2INV   = 1.0E0/(XW1**2+XW2**2+RWTMP**2)

        DO 1300 JN = 1,LPR
c    compute divide needed for next pass
        J_next = IAR2((JN+1)+LPAIR)
        XW1_next     = tmp1-X(1,J_next)
        XW2_next     = tmp2-X(2,J_next)
        RWTMP_next   = tmp3-X(3,J_next)
        R2INV_next   = 1.0E0/(XW1_next**2+XW2_next**2+RWTMP_next**2)
        IC = ICO(IACI+IAC(J))
        DF2 = CGI*CG(J)*R2INV
        EELT = EELT+DF2
        R6 = R2INV**3
        F1 = CN12(1,IC)*(R6*R6)
        F2 = CN12(2,IC)*R6
        ENBT = ENBT + (F2-F1)
        DF = (DF2+6.0E0*((F2-F1))*R2INV
        FW1 = XW1*DF
        FW2 = XW2*DF
        FW3 = RWTMP*DF
        F(1,J) = F(1,J)    +FW1
        F(2,J) = F(2,J)          +FW2
        F(3,J) = F(3,J)    +FW3

        tmp4 = tmp4 -FW1
        tmp5 = tmp5 -FW2
        tmp6 = tmp6 -FW3
c    juggle the values for the next pass.
        J      = J_next
        XW1    = XW1_next
        XW2    = XW2_next
        RWTMP = RWTMP_next
c    result of divide not needed until here.  All the work above this
c    can proceed concurrently with the divide.
        R2INV = R2INV_next
   1300 CONTINUE
```

Note that the last iteration is potentially unsafe since we may go out of bounds. The last iteration may need to be special-cased.

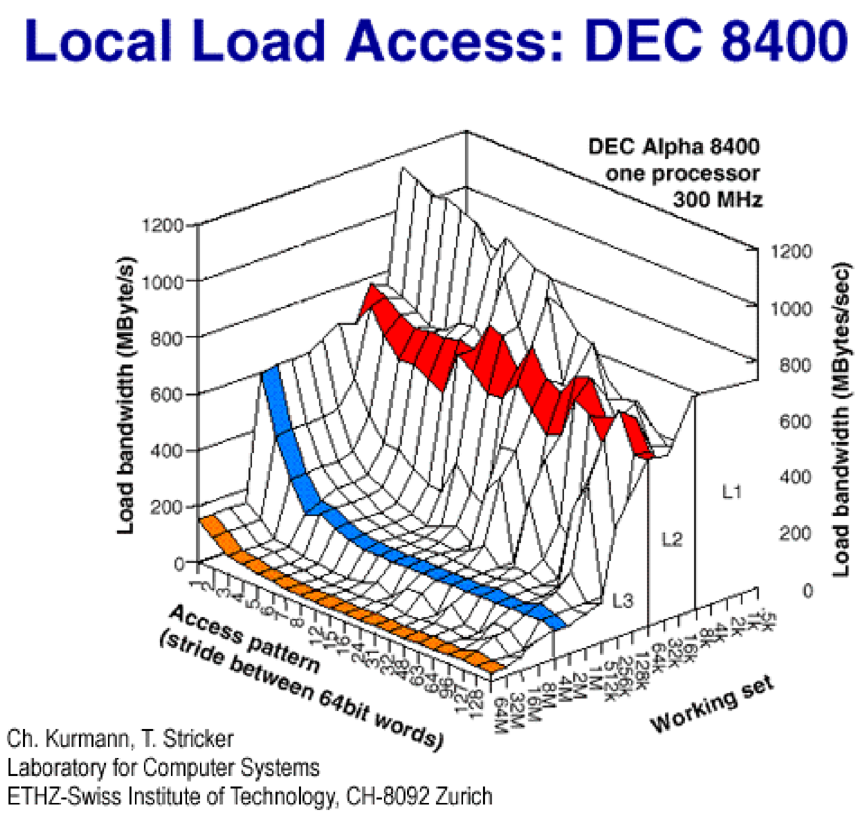
Local Load Access: Pentium Pro PC



Local Load Access: SGI Origin



Local Load Access: DEC 8400

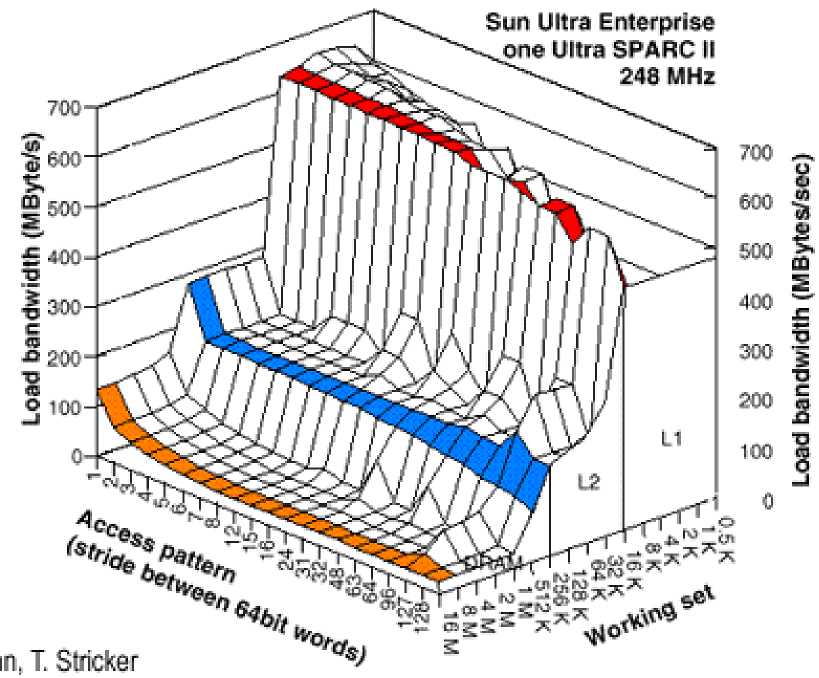


April 11-12, 2002

PITTSBURGH SUPERCOMPUTING CENTER

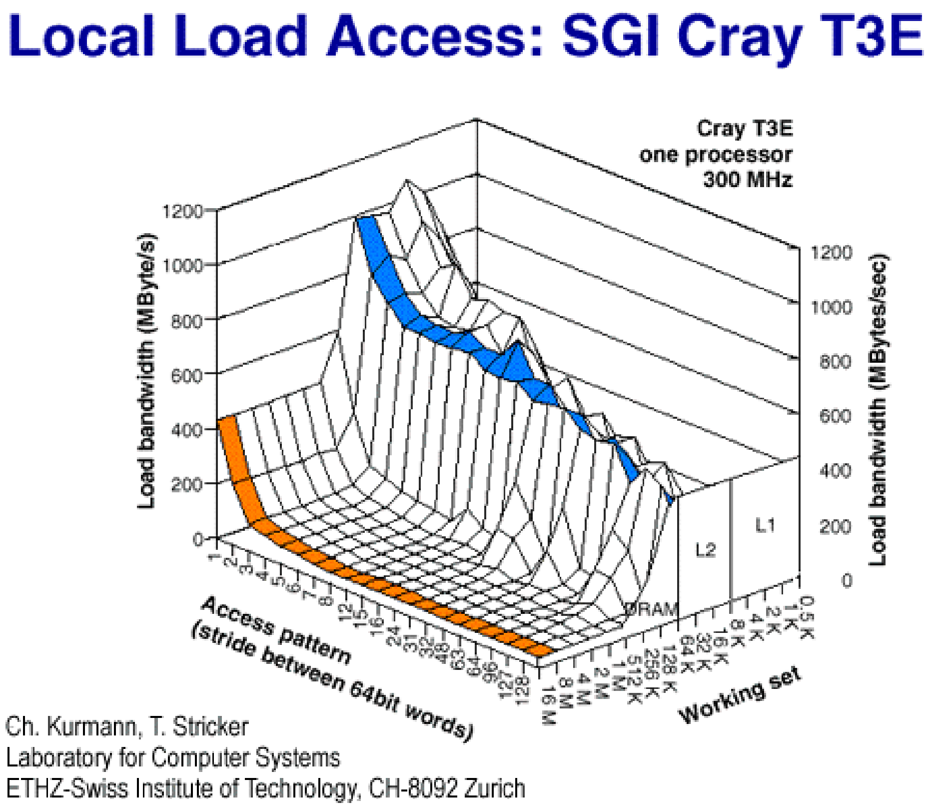
Local Load Access: Sun Enterprise

Local Load Access: Sun Enterprise

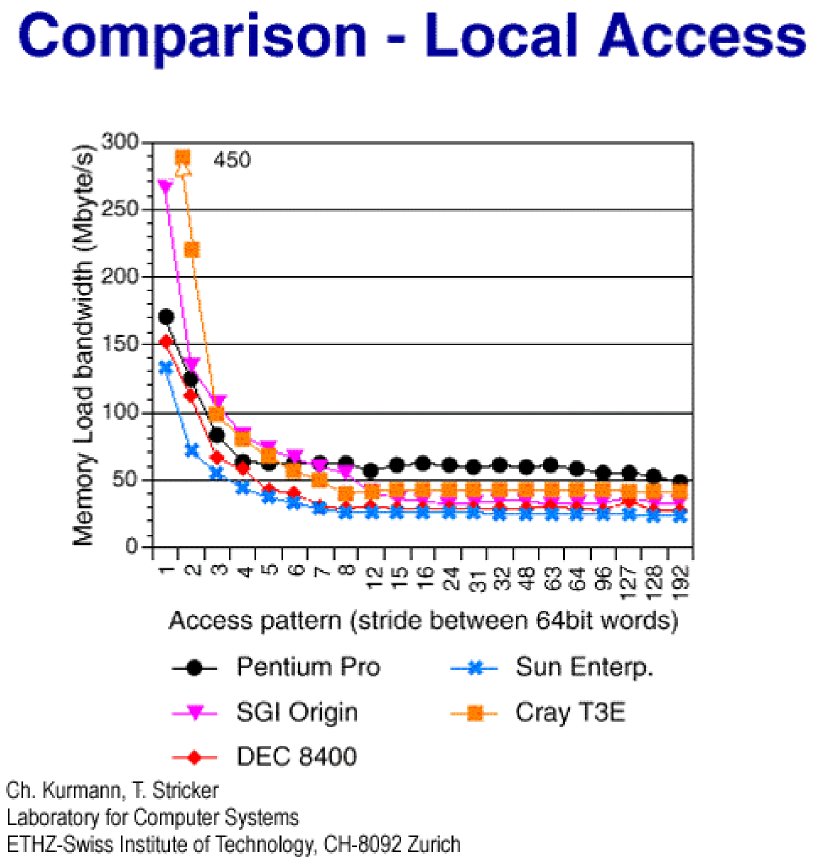


Ch. Kurmann, T. Stricker
Laboratory for Computer Systems
ETHZ-Swiss Institute of Technology, CH-8092 Zurich

Local Load Access: SGI Cray T3E



Comparison – Local Access



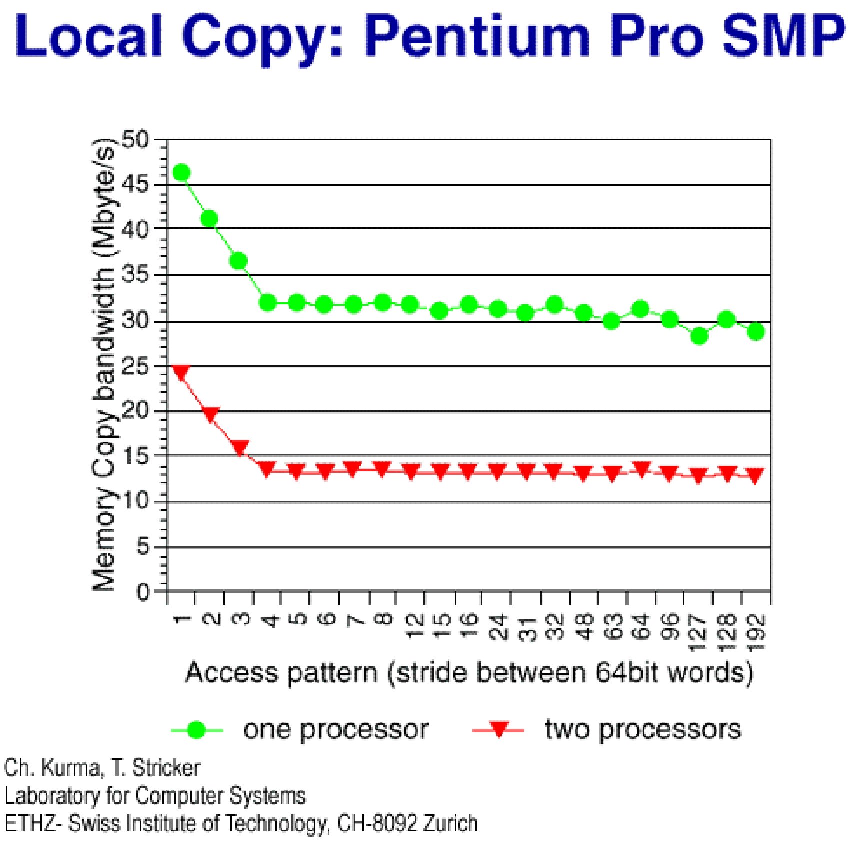
Performance in an SMP Setting

•Copy bandwidth decreases for simultaneous access with 1, 2, 4 and 8 processors

•Topics of interest:
- small working sets in caches: performance remains same
- large working sets in memory: interesting differences
- behavior for even/uneven strides

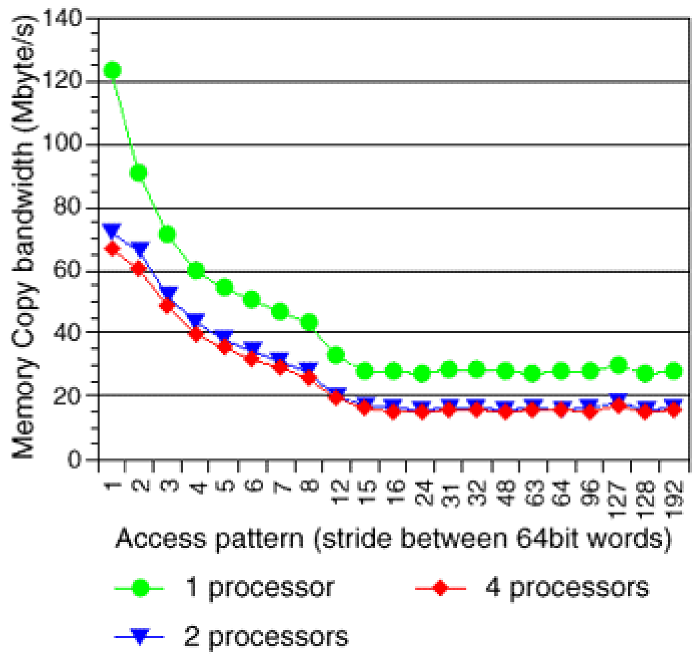
•"Gather copy stream"
(strided load/contiguous store)

Local Copy: Pentium Pro SMP



Local Copy:
SGI Origin CC-NUMA

Local Copy: SGI Origin CC-NUMA

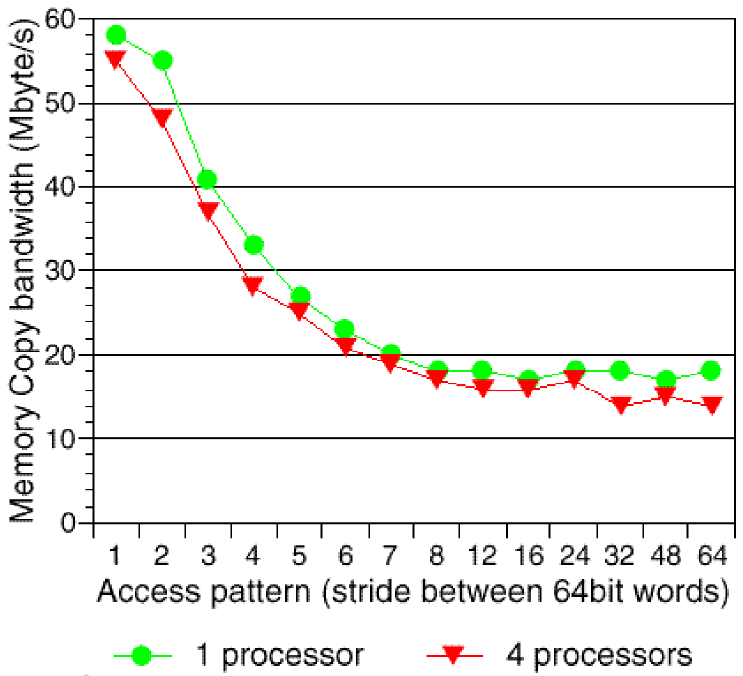


Ch. Kurmann, T. Stricker
Laboratory for Computer Systems
ETHZ-Swiss Institute of Technology, CH-8092 Zurich

PITTSBURGH SUPERCOMPUTING CENTER

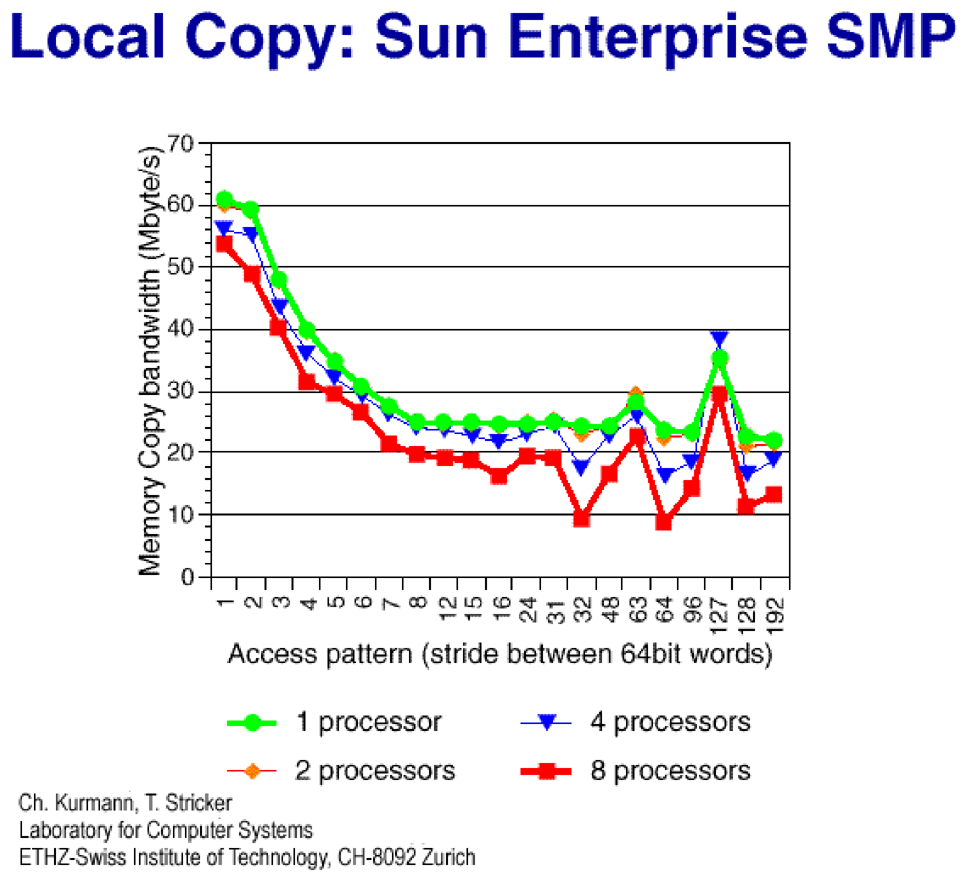
Local Copy: DEC 8400 SMP

Local Copy: DEC 8400 SMP

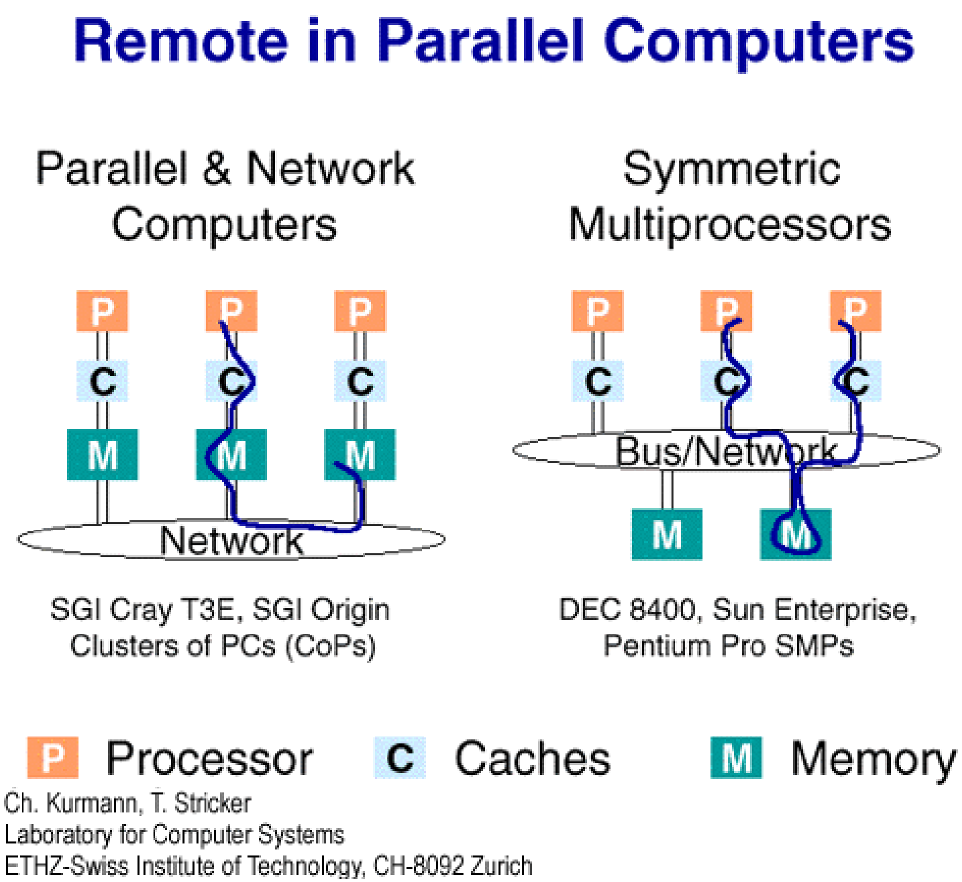


Ch. Kurmann, T. Stricker
Laboratory for Computer Systems
ETHZ-Swiss Institute of Technology, CH-8092 Zurich

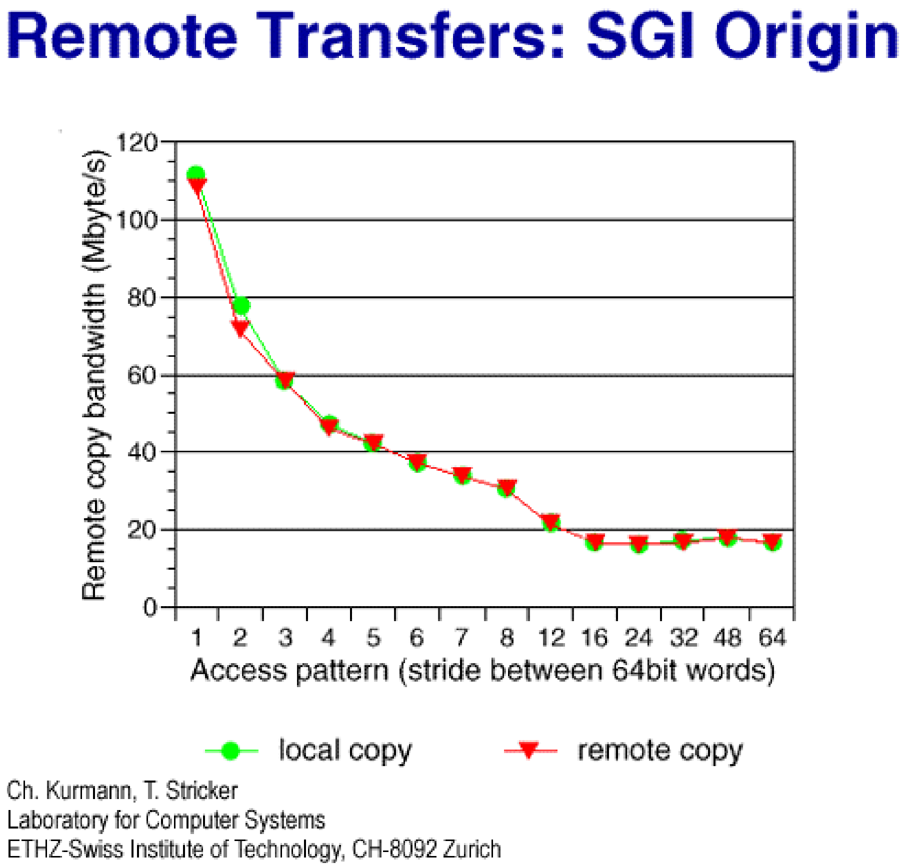
Local Copy: Sun Enterprises SMP



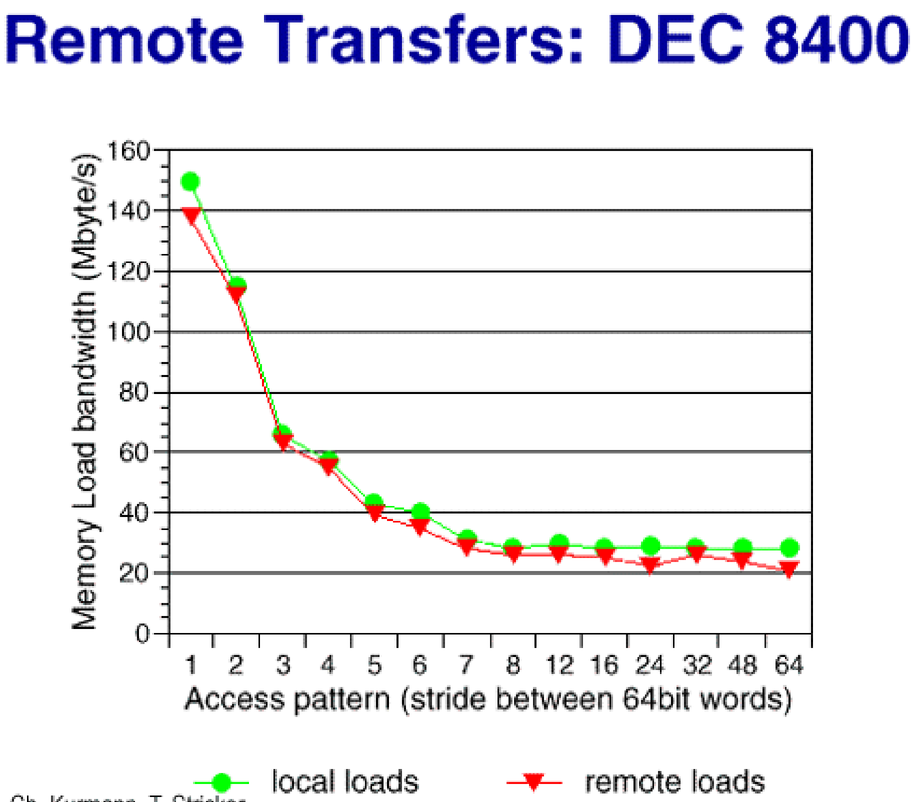
Remote in Parallel Computers



Remote Transfers: SGI Origin

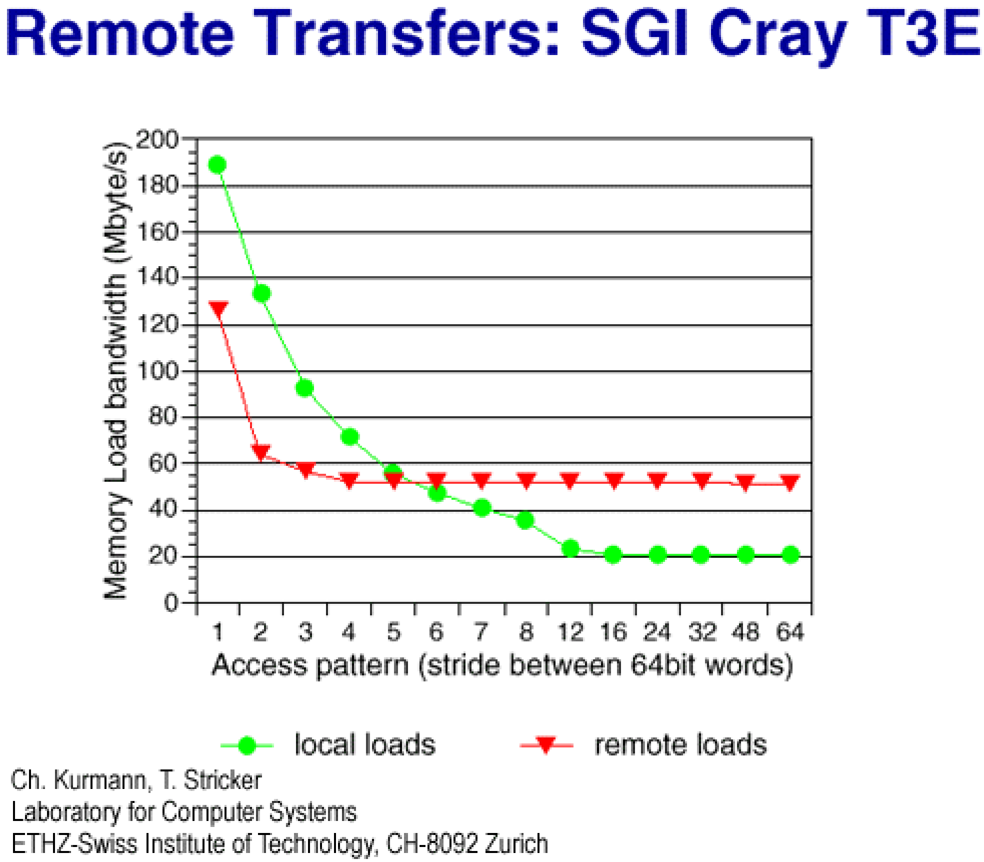


Remote Transfers: DEC 8400



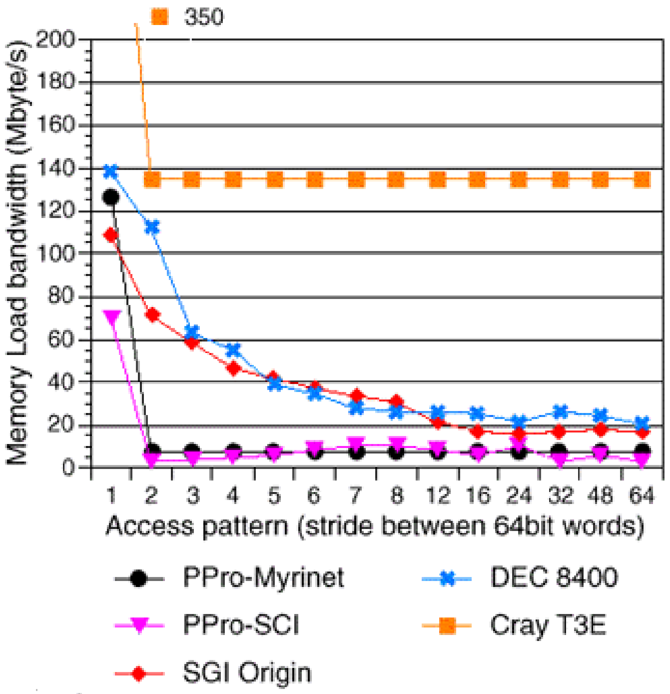
Ch. Kurmann, T. Stricker
Laboratory for Computer Systems
ETHZ-Swiss Institute of Technology, CH-8092 Zurich

Remote Transfers: SGI Cray T3E



Comparison – Remote Transfers

Comparison - Remote Transfers



Ch. Kurmann, T. Stricker
Laboratory for Computer Systems
ETHZ-Swiss Institute of Technology, CH-8092 Zurich