Fredrik Svensson has been with NEC since 2001, working as an Application Analyst. Currently he is working within the Teraflop Workbench, a collaboration between NEC and HLRS.
NEC High Performance Computing Europe GmbH

NEC is dedicated to serve the European market of scientific and technical supercomputing. NEC has been active in this market segment since the early nineties.

High Performance Computing Center Stuttgart

The High Performance Computing Center in Stuttgart supports users from R&D in the use of leading edge supercomputer technology.

The mission of HLRS is to provide its users with tools and expertise to achieve to international positions in their research fields.
NEC’s SX-Series: Innovation since 1983

**SX 1/2 Series**
- The first computer in the world
- Surpassing 1gflops

**SX-3 Series**
- Shared memory
- Multi-function processor
- Unix os

**SX-4 Series**
- First full CMOS design
- Entirely air-cooling
- Introduction of clustered SMP architecture

**SX-5 Series**
- High sustained performance
- Largest ever capacity SHARED MEMORY

**SX-6 Series and SX-7**
- Single-chip vector processor
- TFlops scalability through optimized clustering

**International Recognition**
- Eckart-Mauchly Award of IEEE for CPU architecture
- IEEE and ACM Award for innovation in packaging technology

Use of latest technology to build and develop new generations of supercomputers.
Hardware Resources

(Planned) HLRS Environment:

- SX-8 72nodes
- IXS
- GbEther SW
- GbEther SW
- FC Switch
  - 114 ports
- Nocona ~200 Nds.
- GFS Server TX-7
- Infiniband 216 ports
- FC Disk
  - #1
  - #2
  - #17
  - #18
  - #56
  - #71
  - #72

© NEC HPCE, 2003, finishdesign
SX-8 Vector Architecture

Multiple Vector Parallel Pipelines
4 Pipelines per Operation = Functional Unit
Each Instruction Uses 4 Pipelines
Automatic Hardware Parallelism
Concurrent Pipeline Set Operation
Equally Fast Division Unit
New Square Root Unit
More Efficient Strided Memory Access
No Degradation for Stride Two
Hardware – SQRT

\[ y(i) = \sqrt{x(i)} \]
Hardware – even vs. odd stride

\[ y(i) = x1(i) \times x2(i) \quad i=1,2n,2 \]

\[ y(i) = x1(i) \times x2(i) \quad i=1,3n,3 \]

**Graphs:**

- **Graph 1:**
  - Equation: \[ y(i) = x1(i) \times x2(i) \quad i=1,2n,2 \]
  - Data points for SX-8 and SX-6+
  - Axes: Looplength vs. MFLOPS
  - Range: Loopenlength 1-100000
  - MFLOPS: 0-3000

- **Graph 2:**
  - Equation: \[ y(i) = x1(i) \times x2(i) \quad i=1,3n,3 \]
  - Data points for SX-8 and SX-6+
  - Axes: Looplength vs. MFLOPS
  - Range: Loopenlength 1-100000
  - MFLOPS: 0-3000
Compiler – SX-8

11 Apr 2005 – rev 313

1. `-sx8` was added to support SX-8 systems
2. `-pvctl (no)vsqrt` specifies that hardware instruction is used for vector sqrt
3. `-verbose` option passes option list to standard error for f90
4. `!CDIR NOIEXPAND` Procedures are not expanded
5. Message improvements
6. Automatic parallelization improvement
7. Inline expansion feature is improved
8. Improved unrolling if there is a outer loop around OpenMP do directive
9. Bug fixes
Compiler – SX-8

15 Aug 2005 – rev 315

1. –pi modout Used to keep information for –pi expin=
2. –P i / –P ni Result of intrinsic function SELECTED_INT_KIND(N)
3. Filesuffix decides –pi expin= when –f0/-f3/-f4 is not specified
4. -ftrace specifies that –D_FTRACE is passed to the C preprocessor in F90
5. Optimization for dummy arguments improved
6. Improved error detection in conjunction with OpenMP directive
7. Improved error messages
8. Bug fixes
Compiler Directive

(no)altcode: affects generation of alternative code

(no)assume: assume loop length

(no)compress: compress / expand or masked operation

(no)divloop: affects loop division for vectorisation

loopcnt = … : define expected loopcnt

nodep (important): do vectorisation even if dependency might occur

shortloop: loop length will not exceed vector register length

(no)vector: do (not) vectorise loop if possible

(no)overlap: for usage with pointers
Libraries

**MathKeisan**
- BLAS - basic vector and matrix operations
- PARBLAS – SMP version of BLAS
- CBLAS – C BLAS interface
- Sparse BLAS – Targets sparse vector op.
- LAPACK – solving systems of linear equ.
- ScaLAPACK – MPI LAPACK
- BLACS – MPI library for Linear algebra
- FFT – Fast Fourier Transforms
- ParFFT – OpenMP parallel FFT
- METIS – part / order matrices/graphs
- ParMETIS – MPI version of METIS
- SOLVER – sparse symmetric linear systems
- PARDISO – SMP sparse sym/unsym systems
- ARPACK – Large-scale eigenvalue problems

**ASL – Advanced Scientific Library**
- ASL incorporates the latest advances in numerical analysis and employs algorithms having superior speed and precision
- Optimized from vector processing and SX hardware.
- Matrix algebra, Least Square, Eigenvalues, Eigenvectors, Simultaneous linear equations, FFT, Numerical differentials and integration, Sorting, Root of Equations
- SMP Parallel versions of some of these routines

*Can only be run on a machine with valid license*
Vectorization example

Vectorization of Spray module

\[
\begin{align*}
&\text{do 100 } \text{idrop=1,ndrop} \\
&\quad \text{do 200 while (drop_time(idrop).lt.gas_time)} \\
&\quad \quad \text{do 300 step=1,5} \\
&\quad \quad \quad \text{compute derivatives} \\
&\quad \quad \quad \text{update solution and drop-time} \\
&\quad \quad \quad \text{compute error} \\
&\quad \quad 300 \quad \text{continue} \\
&\quad \quad \quad \text{adjust drop timestep(depending on error)} \\
&\quad \quad \quad \text{do special treatments (interactions etc.)} \\
&\quad \quad 200 \quad \text{continue} \\
&\quad \quad 100 \quad \text{continue}
\end{align*}
\]

Outermost loop running over particles: not vectorizable

Runge-Kutta Timestep
Vectorization example

Vectorized Implementation (1/3)

```plaintext
nndrop = ndrop
DO 200 nndrop.gt.0
  icount = 0
  DO idrop = 1, nndrop
    IF (drop_time(idrop).lt.gas_time) THEN
      icount = icount + 1
      idrop_a(icount) = idrop
    END IF
  END DO
  nndrop = icount
END DO
```

Reduction of drops
Vectorization example (3)

Vectorized Implementation(2/3)

```fortran
  do 300 step=1,5
    V-- do i=1,nndrop
       idrop=idrop_a(i)
       compute derivatives
     V-- end do
  V-- do i=1,nndrop
     idrop=idrop_a(i)
     update solution and drop-time
   V-- end do
  V-- do i=1,nndrop
     idrop=idrop_a(i)
     compute error
  V-- end do
300 continue
```

Runge-Kutta Timestep
Vectorization example (4)

Vectorized Implementation (3/3)

Innermost loops running over particles: vectorizable

```plaintext
V— do i=1,nndrop
   idrop=idrop_a(i)
   adjust drop timestep (depending on error)
V— end do
V— do i=1,nndrop
   idrop=idrop_a(i)
   do special treatments (interactions)
V— end do
200 continue
```
Vectorization – Vector registers
Vectorization – Short loop

In a JAD structure it is possible to reduce the number of memory access
Using vector registers.

```fortran
do 202 i=1,npoin
    res(i)=0.0
202  continue

do 210 k=1,nband_sys
    do 215 j=band_adr(k),band_adr(k+1)-1
        res(j-band_adr(k)+1)=res(j-band_adr(k)+1)
        * jad(j)*vek(komp_e(j))
    215  continue

210  continue
```
Vectorization – Short loop

Stripmine the loop and use vector registers and short loops to save memory accesses

```fortran
integer strip
parameter (strip=256)
!cdir vreg(res1)
real res1
dimension res1(strip)

!cdir shortloop
  do i0=1,npoin,strip
    !cdir shortloop
    do i=1,strip
      res1(i)=0.0
    enddo
  enddo
```

© NEC HPCE, 2003, finishdesign
Vectorization – Short loop

```fortran
!cdir nounroll
  do k=1,nband_sys
    !cdir shortloop
      do i=i0,min(i0+strip-1,band_adr(k+1)-band_adr(k))
        j=i+band_adr(k)-1
        res1(i-i0+1)=res1(i-i0+1)+jad(j)*vek(komp_e(j))
      enddo
    enddo
  enddo
!cdir shortloop
  do i=i0,min(i0+strip-1,npoin)
    res(i)=res1(i-i0+1)
  enddo
enddo
```
Vectorization – Vector registers
Parallelization - Theory

Parallel section
Parallel loops – a special case
where the Index is treated as a
global variable

Critical region
Parallelization - Theory

Serial section

Barrier
Threads low level

The multithreading environment needs to provide a few primitives:
- Thread creation and thread destruction
- Thread synchronization through monitors
- Barriers
- Work-Queues

This can be created with low level primitives in hardware. Atomic operations that can deal with:
- Test-and-set
- Memory fences
- Fetch-and-add
Parallelization – Thread model

**Shared memory parallelization**
Threads share the program and the data section, but not the stack and register values.

**Microtasking**
The most primitive parallelization structures on SX

**Autotasking**
SX automatic parallelization

**OpenMP**
A standard that is also available on SX machines
SPMD and Fork-Join

SPMD:

Fork-Join:
Comparison

Microtasking: SPMD
Autotasking: Fork-Join
OpenMP:
  ‘outside’ parallel region: Fork-Join
  inside parallel region: SPMD
SX Threads

Microtasking
These are the most primitive ones, synchronized with thread library, and system calls
- Better performance
- Better understanding what is happening
- Learn how to avoid errors in OpenMP

Autotasking
Autotasking uses Microtasking directive to construct automatic parallel regions.
- The compiler finds out which can be parallelized
- The compiler tries to find out what variables are local and global
- The compiler generates a subroutine for parallel constructs
Using report flags under FORTRAN shows parallelization.
Barriers – Good and bad

This code will produce wrong result. It has a race between the threads.

```
subroutine parsum( n,a,sum )
  real a(n)
  sum      = 0.0
  sumlocal = 0.0
  !cdir pardo for, nobarr=(entry,exit)
  do i = 1, n
    sumlocal = sumlocal + a(i)
  end do
  !cdir critical
  sum = sum + sumlocal
  !cdir end critical
  return
end
```

Summation reproducibility problems can also occur in this example.
So where are we waiting

Explicit barrier
Implicit barrier – loop beginning
Implicit barrier – loop ending
Serial section
Critical section
End of microtasked subroutine

!cdir barrier | ex_lpmbarr
!cdir pardo nobarr=(exit) | ex_lpminit
!cdir pardo nobarr=(entry) | ex_lpmterm
!cdir serial/endserial | ex_lpmbgs2
!cdir critical/endcritical | ex_lpmbgcr
return | ex_lpmret

dt * n * (n-1)/2 = time spent in CS with n threads with the worktime dt.
OpenMP

OpenMP is an API for writing multithreaded applications in a shared memory environment

• It consists of a set of compiler directives and library routines
• Relatively easy to create multi-threaded applications in Fortran, C and C++
• Standardizes the last 15 or so years of SMP development and practice
• Is portable between platforms
• OpenMP does not parallelize automatically
• If the step is taken to OpenMP one should consider to use some way to keep the microtaskning that is more efficient on the SX
Parallelization – MPI

MPI is a standard for message passing. It is used to communicate between processes with their own memory address space.

MPI/SX is based on a shared memory MPI implementation

Three different protocols depending on size
- Short message: $0 \leq n \leq 1024$ bytes
- Medium message: $1024 < n \leq 204792$ bytes
- Large messages: $n > 204792$ bytes
Domain decomposition
Domain decomposition
Domain decomposition
Original Communication Pattern

Original version followed a pattern of:

1. Reorganization of the data per CPU
2. Sending the data with MPI_ALLTOALL
3. Reorganization of the data for FFT
   - Doing the FFT
   - Reorganization of the data per CPU
4. Sending the data with MPI_ALLTOALL
5. Reorganization of the data in the original format

Multithreaded part
Serial MPI Call
Domain decomposition

FFT

Flow

Strip-mine
New Communication Pattern

Strip-mining the communication does not bring really a change pre se, but enables the next step with overlapping communication with communication.
New Communication Pattern

Then new communication limits the number of barrier thanks to the division of the work between independent arrays.
New Communication Pattern
Comparison

1
2
3
4
5

1
2
3
4
5
New Communication Pattern

1 2 3 4 5

Time

1 2 3 4 5

Time
MPI – Global memory

Since MPI/SX is a shared memory implementation it is using special memory copies to move data between the threads.

There exists a special memory region called Global Memory.
MPI – Global memory

By using a special array that already is located in the Global memory it is possible to avoid one of the memory copies. This array would be the normal send array that one prepares anyway for exchanging data.
Using !cdir GM_ARRAY(arraynamn)

```
real(dim),dimension(:,:),pointer,save::buffer
!cdir GM_ARRAY(buffer)
...
allocate(buffer(BUFFERSIZE*NCPU,4))
...
for i=1,4
  call MPI_ALLTOALL(buffer(1,i),BUFFERSIZE,MPI_REAL
                inbuffer(1,i),BUFFERSIZE,MPI_REAL,
                MPI_COMM_WORLD,ierr)
enddo
```
MPI – Global Memory – method 2

Using MPI standard constructs

```fortran
integer(KIND=MPI_ADDRESS_KIND) iptr
pointer(tmpptr,tmparray(BUFFERSIZE))
real(dim) tmparray
real(dim),dimension(:,,:),pointer,save::buffer
...
call MPI_ALLOC_MEM(BUFFERSIZE*4*NCPU,MPI_INFO_NULL,
                   iptr,ierr)
tmppptr=iptr
call SETIT(temparray(1),buffer,BUFFER_SIZE,4)
...
for i=1,4
call MPI_ALLTOALL(buffer(1,i),BUFFERSIZE,MPI_REAL
                  inbuffer(1,i),BUFFERSIZE,MPI_REAL,
                  MPI_COMM_WORLD,ierr)
enddo
```
MPI – Global Memory – method 2

At the end, within the routine where SETIT is used (here mpicall)

contains

    subroutine setit(a,b,id0,id1)
      integer(dim) :: id0,id1
      real(dim), target :: a(id0,id1)
      real(dim),dimension(:,:),pointer :: b
      
      b => a
      
      end subroutine setit
    end subroutine mpicall
MPI – Global Memory – method 3

Upcoming Fortran 2003 standard, not yet in use

```fortran
integer :: shape(1)
real(dim), dimension(:), pointer :: buffer

type(c_ptr) p

... call MPI_ALLOC_MEM(BUFFERSIZE*NCPU, MPI_INFO_NULL, p, ierr)
shape(1) = n

call C_F_POINTER(p, buffer, shape)

... call MPI_ALLTOALL(buffer(1), BUFFERSIZE, MPI_REAL
inbuffer(1), BUFFERSIZE, MPI_REAL, MPI_COMM_WORLD, ierr)
```
MPI – Global Memory

**Advantage**
- Faster MPI communication since copying is reduced
- Possibility to use one sided MPI communication

**Disadvantage**
- SX-8 is not truly a shared memory machine, IXS have some support for GM
- Avoid allocating and deallocating the Global Memory many times
- Not swapable
MPI – One sided communication

One sided communication requires global memory

```fortran
call MPI_ALLTOALL(buffer(1),BUFFERSIZE,MPI_REAL
                inbuffer(1),BUFFERSIZE,MPI_REAL,
                MPI_COMM_WORLD,ierr)
```

Setup environment for one sided communication

```fortran
call MPI_WIN_CREATE(inbuffer,BUFFERSIZE*MPI_REAL,
                    MPI_REAL,MPI_INFO_NULL,MPI_COMM_WORLD
                    win,ierr)
```

```fortran
call MPI_COMM_GROUP(MPI_COMM_WORLD,group,ierr)
```
MPI – One sided communication

Doing the one sided communication

```
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
call MPI_WIN_POST(group,MPI_MODE_NOCHECK,win,ierr)
call MPI_WIN_START(group,MPI_MODE_NOCHECK,win,ierr)

ip = me + 1
if ( ip .eq. nprocs ) ip = 0
ito = me*BUFFERSIZE
do while (ip .ne. me)
    iso=ip*BUFFERSIZE
    call MPI_PUT(buffer(1+iso),BUFFERSIZE,MPI_REAL,
                ip,ito,BUFFERSIZE,win,ierr)
    ip = ip + 1
    if ( ip .eq. nprocs ) ip = 0
endo
```
MPI – One sided communication

Finishing the one sided communication

iso = ito
inbuffer(1+ito:1+ito+BUFFERSIZE) = buffer(1+ito:1+ito+BUFFERSIZE)

call MPI_WIN_COMPLETE(group, MPI_MODE_NOCHECK, win, ierr)
call MPI_WIN_WAIT(group, MPI_MODE_NOCHECK, win, ierr)
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
MPI – One sided communication

The more complex one sided communication reduces the synchronization needed in the exchange of the data in the MPI call.
I/O

- Reduce the numbers of data transfers between the I/O buffers and the user data area
- Improving the efficiency of using the I/O buffer
- Reducing the number of I/O operations
- Removing conversion
- Using asynchronous I/O
Summary

Vectorize

Parallelize

- Autotasking
- Microtasking
- OpenMP if portability is needed (Speed / Portability)