Vectorization on NEC supercomputers

Dr. M. Galle
NEC EHPCTC, Stuttgart

Copyright

by
Dr. Rudolf Fischer <rfischer@ess.nec.de>
NEC ESS, Duesseldorf
Dr. Erich Focht <efocht@ess.nec.de>
NEC ESS/EHPCTC, Stuttgart
Dr. Martin Galle <mgalle@ess.nec.de>
NEC ESS/EHPCTC, Stuttgart

Reproduction and usage of the contents of this presentation in own presentations without permission of the authors is prohibited.
Overview

- SX-5 Specification
- Vectorisation
  - hardware realization
  - vectorization examples: direct addressing
  - vectorization examples: indirect addressing
- Performance analysis

NEC SX-5
The Specifications

- Single Node
  - Up to 128 GFLOPS with 16 x 8 GFLOPS Processors
  - Up to 128 GBytes Shared Main Memory
- Multi Node
  - Up to 4 TFLOPS
  - Up to 32 Nodes Using SX-5 IXS
  - Up to 512 Processors
  - Up to 4 TBytes Main Memory

Vector Unit Architecture

- Multiple Vector Parallel Pipelines
- (32) 64 SX-4 Compatible Pipelines
  - Add-Shift x (8) 16
  - Multiply x (8) 16
  - Logical x (8) 16
  - Divide x (8) 16
- Each Instruction Uses (8) 16 Pipelines
  - Automatic Hardware Parallelism
- Concurrent Pipeline Set Operation
Scalar Architecture

- Superscalar
  - Decodes 4 Instructions per Clock
  - Branch Prediction
  - Out-of-Ordering
    - Scalar Instructions
    - Vector Instructions
    - Memory References
  - 64 KB Instruction + 64 KB Operand Cache
  - 8 KB Instruction Buffer

SX-5 Scalar Unit Block Diagram
IEEE Format (float0) !!!

- 4 Byte:
  - about 7.2 digits
  - $10^{-38}$ - $10^{38}$
- 8 Byte:
  - about 16 digits
  - $10^{-308}$ - $10^{308}$
- 16 Byte:
  - about 32 digits
  - $10^{-308}$ - $10^{308}$
  - not vectorizable!

Vectorization

- levels of parallelism:
  - instruction-level parallelism (e.g. superscalar processor, hardware detects instructions which can be executed in parallel on several functional units)
  - thread-level parallelism (e.g. multiprocessor machine, multiple instruction streams)
  - vector data-parallelism:
    - each instruction leads to a large number of similar operations on arrays of data
    - one decode & issue for N operations
    - very regular and well known memory access pattern, i.e. well hiding latencies
- NEC SX: all three kinds of parallelism
Segmentation, Pipelining

- Operations are decomposed into segments
- Example: floating point add
  - compare exponent
  - shift mantissa
  - add mantissa
  - select exponent and normalize

\[\begin{align*}
1.14e9 & - 2.78e8 \\
1.14e9 & - 0.278e9 \\
0.862e9 & \\
8.62e8 & 
\end{align*}\]

Segmentation, Pipelining (2)

- **no pipelining**
- **pipelining**
Segmentation, Pipelining (3)

- Superscalar pipeline (e.g. in RISC CPU)

Segmentation, Pipelining (4)

- Vector pipeline on NEC:
  - (8) 16-fold parallel on SX-5

- Better have data parallelism in mind than pipelines when thinking about vectors
Data Parallelism

- vector loop: data parallel
  each loop iteration could be executed in parallel
- scalar loop: not data parallel
  current iteration depends on results of previous one

Vectorization examples

- \( V = S + V \)
- \( V = V + V \)
- \( V = V + S \times V \)
- \( S = S + V \times V \)
- matrix multiply
ex. 1: \( v = s + v \)

- **FORTRAN:**
  
  ```fortran
  f77
  do i = 1, n
    v(i) = s + w(i)
  end do
  ```

  ```fortran
  f90
  v(:) = s + w(:)
  ```

- **what the compiler generates:**
  
  ```fortran
  do i0 = 1, n, 256
    do i = i0, min( n, i0+255 )
      v(i) = s + w(i)
    end do
  end do
  ```

ex. 1: \( v = s + v \) (cont.)

- **timing diagram 1 (false)**

- **cycles**
  
  - estimate: \( R < (8) \times 16 \times R_0 / 3 = (666) \times 1333 \text{ MFlops} \)
  
  - measured: \( R \sim (1321) \times 2654 \text{ MFlops} \)
Chaining

Memory

\begin{align*}
    y(1) \\
    y(2) \\
    y(3) \\
    y(4) \\
    y(5) \\
    y(6) \\
    y(7) \\
    y(8)
\end{align*}

vfad

\begin{align*}
    s+y(1) \\
    s+y(2) \\
    s+y(3) \\
    s+y(4) \\
    s+y(5) \\
    s+y(6) \\
    s+y(7) \\
    s+y(8)
\end{align*}

ex. 1: \( v = s + v \) (cont.)

- timing diagram 2 (false)

- cycles

- estimate: \( R < (8) 16 \times R_0 / 2 = (1000) 2000 \) MFlops
- measured: \( R \approx (1321) 2654 \) MFlops
**ex. 1: \( \nu = s + \nu \) (cont.)**

- more chimes, timing diagram:

  ![Timing Diagram]

  - R < (8) 16 * R0 * 2 / 3 = (1333) 2667 MFlops
  - measured: R ~ (1321) 2654 MFlops

**ex. 2: \( \nu = \nu + \nu \)**

- FORTRAN:
  ```fortran
  do i = 1, n
  x(i) = y(i) + z(i)
  end do
  x(:) = y(:) + z(:)
  ```

- Timing Diagram:

  ![Timing Diagram]

  cycles
ex. 2: \( v = v + v\) (cont.)

- Timing Diagram:

- \( R < (8) 16 \cdot R_0 / 2 = (1000) 2000\) Mflops
- measured: \( (991) 1981\) MFlops

ex. 3: \( v = v + s \cdot v\)

- FORTRAN:
  
  ```fortran
  do i = 1, n
    x(i) = y(i) + s * z(i)    \( x(:) = y(:) + s \cdot z(:) \)
  end do
  ```

- Timing Diagram:

- cycles
**ex. 3: \( v = v + s \times v \) (cont.)**

- Timing Diagram:
  - Timing Diagram:
  - \( \text{load} \) 
  - \( \text{store} \)
  - \( \text{load} \)
  - \( \text{vfad} \)
  - \( \text{vfmul} \)

  **cycles**

- \( R < (8) \ 16 \times R_0 \times 2 / 2 = (2000) \ 4000 \text{ MFlops} \)
- measured: \( R = (1992) \ 3986 \text{ MFlops} \)

**ex. 4: \( s = s + v \times v \)**

- **FORTRAN:**
  
  \[
  \text{do } i = 1, n \quad \text{do } i = 1, n \\
  s = s + x(i) \times y(i) \quad s = \text{dot_product}(x,y) \\
  \text{end do} \\
  \text{end do}
  \]

- Recursion? Generated Code (cum grano salis):
  
  \[
  \begin{align*}
  &\text{real stemp(256)} \\
  &\text{do } i0 = 1, n, 256 \\
  &\text{do } i = i0, \text{min}(n, i0+255) \\
  &\quad \text{stemp}(i-i0+1) = \text{stemp}(i-i0+1) + x(i) \times y(i) \\
  &\text{end do} \\
  &\text{end do} \\
  &s = \text{reduction(stemp)}
  \end{align*}
  \]
**ex. 4: s = s + v * v (cont.)**

- Timing Diagram:
  - load
  - load
  - vfad
  - vfad
  - vfmul
  - vfmul

  **cycles**

- $R < (8) 16 \times R_0 \times 2 = (4000) 8000$ Mflops
- measured: $R = (3789) 7763$ MFlops

---

**Startup and Short Vectors**

- carefully measure!
- Loop ex. 1:
  - Length 100: 543.9 MFlops (SX-4)
  - Length 256: 888.3 MFlops (SX-4)
- Loop ex. 4:
  - Length 256: 429.7 MFlops (SX-4)
- Explanation?
Measurements (SX-4)

\[ V = S + V \]
\[ S = S + V^* \cdot V \]
\[ V = V + S^* \cdot V \]

vectorization of if-blocks

- **FORTRAN:**
  ```fortran
  do i = 1, n
      if(y(i) .gt. 0.5) then
          x(i) = 1.0 + y(i)
      else
          x(i) = y(i) * y(i)
      end if
  end do
  ```

- can be vectorized by using mask-registers
vectorization of if-blocks

- FORTRAN:
  ```fortran
do i = 1, n
  if ( y(i) .ge. 0.0 ) then
      x(i) = sqrt(y(i))
  end if
  end do
```
- alternative vectorization by compress / expand
- *vdir (no)compress
vectorization of if-blocks

use compress / expand

Special SX features

- vector reduction operations
  - example for usage: SDOT (ex. 4)
  - sum up all elements of one Vector-register
- linear recurrence (etc.)! FORTRAN:

\[
\text{do } i = 2, n \\
\text{ a(i) = b(i) + a(i-1) * c(i) } \\
\text{end do}
\]

- special instructions available
- not as fast as real vector, but better than scalar
- vector data registers
**ex. s1: matrix multiply**

- FORTRAN:

  ```fortran
  do i = 1, n
    do j = 1, n
      do k = 1, n
        c(i, j) = c(i, j) + a(i, k) * b(k, j)
      end do
    end do
  end do
  ```

- which order of loops? (discuss)
- totally different on assembler level
- replaced by lib-call (compiler!)

---

**ex. s1: matrix multiply (2)**

Diagram showing matrices `a`, `b`, and `c` with indices `i`, `j`, and `k`.
ex. s1: matrix multiply (3)

- FORTRAN equivalent to library call:

```fortran
real accu(256)
do j = 1, n
  do i0 = 1, n, 256
    iend = min( n, i0+255 )
    do i = i0, iend
      accu(i) = c(i,j)
    end do
    do k = 1, n
      do i = i0, iend
        accu(i-i0+1)=accu(i-i0+1)+a(i,k)*b(k,j)
      end do
    end do
  end do
end do
end do
```

ex. s1: matrix multiply (4)

```
\begin{align*}
\text{a} & \quad \text{b} \\
\downarrow & \quad \downarrow \\
\text{k} & \quad \text{j} \\
\end{align*}
```

```
\begin{align*}
\text{c} & \quad \text{b} \\
\downarrow & \quad \downarrow \\
\text{i} & \quad \text{i} \\
\end{align*}
```
Basic Rules for Performance

- vectorize important portions
- data parallelism or reduction for innermost loop
- long innermost loop
- lots of instructions in innermost loop
- stride one or at least odd stride
- avoid indirect addressing
- keep loop structure ‘simple’

FORTRAN 90 !cdir

- (no)altcode: affects generation of alternate code
- (no)assume: assume loop length
- (no)compress: compress / expand or masked operation
- (no)divloop: affects loop division for vectorization
- loopcnt = ... : define expected loopcnt
- nodep (most important): do vectorization even if dependency might occur
- shortloop: loop length will not exceed vector register length
- (no)vector: vectorize loop if possible
- vreg
Optimization examples

- loop interchange
- loop expansion
- loop division
- call to function
- 2D recursion
- indirect addressing:
  - usage of directive
  - non-injective list vector

loop interchange

- FORTRAN:
  
  ```fortran
  do j = 1, n
    do i = 2, n
      a(i,j) = a(i-1,j) * b(i,j) + c(i,j)
    end do
  end do
  ```

- in spite of linear recurrence instruction exchange of loop will improve performance
- switch indices? depends on leading dimension
**loop expansion**

- **FORTRAN:**
  
  ```fortran
  do i = 1, n
    do j = 1, 4
      a(i, j) = a(i, j) * b(i, j) + c(i, j)
    end do
  end do
  ```

- **f90 -Wf"-pvctl expand=4":**
  
  ```fortran
  do i = 1, n
    a(i, 1) = a(i, 1) * b(i, 1) + c(i, 1)
    a(i, 2) = a(i, 2) * b(i, 2) + c(i, 2)
    a(i, 3) = a(i, 3) * b(i, 3) + c(i, 3)
    a(i, 4) = a(i, 4) * b(i, 4) + c(i, 4)
  end do
  ```

---

**loop division**

- **FORTRAN:**
  
  ```fortran
  do j = 1, n
    do i = 2, n
      b(i, j) = sqrt(x(i, j))
      a(i, j) = a(i-1, j) * b(i, j) + c(i, j)
      y(i, j) = sin( a(i, j) )
    end do
  end do
  ```

- Inner loop vectorized with help of recursion instructions
**call to function**

- **FORTRAN:**
  ```fortran
do i = 1, n
  y(i) = myfun(x(i))
end do

real function myfun(a)
myfun = sqrt(a)
return
end
```

- **solution:**
  - statement function
  - automatic inlining (Use "-pi" option!)

---

**2D recursion**

- **FORTRAN:**
  ```fortran
do j = 2, n
  do i = 2, m
    x(i, j) = rhs(i, j) - a(i, j) * x(i-1, j) - b(i, j) * x(i, j-1)
  end do
end do
```

- **solution:**
  - hyperplane-ordering:
2D recursion (2)

- FORTRAN (needs directive!):

```fortran
  do idiag=1,m+n-1
    *cdir nodep
    do j = max(1,idiag+1-m), min(n,idiag)
      i=idiag+1-j
      x(i,j)=rhs(i,j)-a(i,j)*x(i-1,j)-b(i,j)*x(i,j-1)
    end do
  end do
```

- challenge: get indices and loop parameters right!
- works for general cases, too (i.e. unstructured grids)

indirect addressing

- FORTRAN:

```fortran
  do i=1,n
    j=ija(i)
    a(j)=a(j)+b(i)
  end do
```

- simple case: **ija(i) is injective** (i→ija)
  vectorize by using directive

```fortran
  !cdir nodep
  do i=1,n
    j=ija(i)
    a(j)=a(j)+b(i)
  end do
```
**indirect addressing (2)**

- difficult case: \(ija(i)\) is not injective:

```
\[ \begin{align*}
    i & \quad j \\
    1 & \quad 3 \\
    2 & \quad 2 \\
    3 & \quad 1 \\
    4 & \quad 3 \\
    5 & \quad 4 \\
    6 & \quad 1 \\
    7 & \quad 1 \\
    8 & \quad 3 \\
    9 & \quad 2 \\
    10 & \quad 3 \\
    11 & \quad 1 \\
    12 & \quad 3
\end{align*} \]
```

```
\begin{align*}
    a(3) &= a(3) + b(1) \\
    a(2) &= a(2) + b(2) \\
    a(1) &= a(1) + b(3) \\
    a(3) &= a(3) + b(4) \\
    a(4) &= a(4) + b(5) \\
    a(1) &= a(1) + b(6) \\
    a(1) &= a(1) + b(7) \\
    a(3) &= a(3) + b(8) \\
    a(2) &= a(2) + b(9) \\
    a(3) &= a(3) + b(10) \\
    a(1) &= a(1) + b(11) \\
    a(3) &= a(3) + b(12)
\end{align*}
```

```
\begin{align*}
    \text{do } i=1,n \\
    j &= ija(i) \\
    a(j) &= a(j) + b(i) \\
    \text{end do}
\end{align*}
```

Using directive is not correct!

**indirect addressing (3)**

- difficult case: \(ija(i)\) is not injective

```
do i=1,n \\
    j = ija(i) \\
    a(j) = a(j) + b(i) \\
end do
```

- solutions without setup phase:
  - Dynamic detection of overwritings: Loop is executed as before; results are corrected afterwards
  - Vector distribution of results: Use additional injective dimension
  - Use (new) compiler option: `-Wf-pvctl listvec`

```
do i=1,n \\
    j = ija(i) \\
    a(j) = a(j) + b(i) \\
end do
```
**indirect addressing (4)**

- **Dynamic detection of overwritings**

```fortran
! do i=1,n
!   j=ija(i)
!   tmp(i)=a(j)+b(i)
!   a(j)=i
! enddo
! do i=1,n
!   j=ija(i)
!   if(a(j).ne.i)then
!     icnt=icnt+1
!     list(icnt)=i
!   endif
!   a(j)=tmp(i)
! enddo
! do k=1,icnt
!   i=list(k)
!   j=ija(i)
!   tmp(j)=a(j)+b(i)
!   a(j)=i
! enddo
```

- **calculate A+B and mark positions**

- **detect overwritings and put them into a list**

- **process list as before, repeat until list empty**

---

**indirect addressing (5)**

- **Vector distribution of results**

```fortran
! dimension a(m), av(256,m)
! do i=1,n
!   iv = mod(i-1,256)+1
!   av(iv,ija(i))=av(iv,ija(i))+b(i)
! end do
! do iv=1,256
!   do j=1,m
!     a(j)=a(j)+av(iv,j)
!   end do
! end do
```

- **Approach makes sense only if n >> m!**
indirect addressing (6)

- difficult case: \textbf{ija(i) is not injective}

- solutions with setup phase:
  - Sort array \textbf{ija}, if only few values: update for each value
  - use problem-specific structure of \textbf{ija} as \textbf{Multicoloring} or hyperplanes
  - \textbf{JAD}: reorder \textbf{ija} into injective chunks and vectorize over the chunks (reordering routine(s) available on request)

\begin{verbatim}
do i=1,n
  j=ija(i)
  a(j)=a(j)+b(i)
end do
\end{verbatim}

indirect addressing (7)

- Sort array \textbf{ija} (1/2)

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{ic0} & \textbf{j} & \textbf{ia} \\
\hline
1 & 1 & 3 \\
2 & 1 & 6 \\
3 & 1 & 7 \\
4 & 1 & 11 \\
\hline
5 & 2 & 2 \\
6 & 2 & 9 \\
\hline
7 & 3 & 1 \\
8 & 3 & 4 \\
9 & 3 & 8 \\
10 & 3 & 10 \\
11 & 3 & 12 \\
\hline
12 & 4 & 5 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|c|}
\hline
\textbf{ic}: & 1 & 2 & 3 & 4 \\
\hline
      & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
      & 12 \\
\hline
\textbf{icol} & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
\hline
\textbf{j}: & 1 & 2 & 3 & 4 \\
\hline
\textbf{ia}: & 3 & 6 & 7 & 11 & 2 & 9 & \\
\hline
\textbf{j=irow} & 1 & 4 & 8 & 10 & 12 & 5 & \\
\hline
\end{tabular}
indirect addressing (8)

- Sort array ija (2/2)
  outer loop over irow
  inner loop over icol:

```fortran
  do irow=1,nrow
    as = 0
    j = irow
    do icol=1,istop(irow)
      ic = ic0(irow)+icol-1
      as = as + b(ia(ic))
    end do
    a(j) = a(j) + as
  end do
```

- If \( b(ia(ic)) \) is constant, replace it by \( b1(ic) \) (direct addressing!)

indirect addressing (9)

- Multicoloring (1/2)

```plaintext
ic0(1)=1
  ic   ja    ia
  1    3     1
  2    2     2
  3    1     3
  4    4     5

ic0(2)=5
  ic   ja    ia
  5    3     4
  6    1     6
  7    2     9

ic0(3)=8
  ic   ja    ia
  8    1     7
  9    3     8

ic0(4)=10
  ic   ja    ia
  10   3     10
  11   1     11

ic0(5)=12
  ic   ja    ia
  12   3     12
```

```fortran
  do irow=1,nrow
    ic = ic0(irow)
    do icol=1,icol(irow)
      irow = istop(irow)
      ia(ic) = icol(irow)
    end do
  end do
```

```fortran
  icolor = ic
  ja(irow) = icolor
  ia(irow) = icolor
```

```fortran
  do irow=1,nrow
    icolor = ic0(irow)
    do icol=1,icol(irow)
      irow = istop(irow)
      ia(icolor) = icol(irow)
      ja(irow) = icolor
    end do
  end do
```
indirect addressing (10)

**Multicoloring (2/2)**

outer loop over icolor
inner loop over irow:

```fortran
do icolor=1,ncolor
  do irow=1,istop(icolor)
    ic=ic0(icolor)+irow-1
    j=ja(ic)
    a(j)=a(j)+b(ia(ic))
  end do
end do
```

- Multicoloring is frequently used if more than one dependency has to be considered: \( j1=ija(1,i) \), \( j2=ija(2,i) \)
- If \( b(ia(ic)) \) is constant, replace it by \( b1(ic) \) (direct addressing!)

indirect addressing (11)

**JAD ordering (1/2)**

<table>
<thead>
<tr>
<th>ic</th>
<th>ja</th>
<th>ia</th>
</tr>
</thead>
<tbody>
<tr>
<td>ic0(1)=1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

| ic0(2)=5 | 5   | 3   |
|          | 6   | 1   |
|          | 7   | 2   |

| ic0(3)=8 | 8   | 3   |
| ic0(4)=10| 9   | 1   |
| ic0(5)=12 | 10  | 3   |
|           | 11  | 1   |
|           | 12  | 3   |

<table>
<thead>
<tr>
<th>ic:</th>
<th>icol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ja:</th>
<th>irow</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ia:</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>
indirect addressing (12)

- JAD ordering (2/2)
  outer loop over icol
  inner loop over irow:

  ```
  do icol=1,ncol
    do irow=1,istop(icol)
      ic=ic0(icol)+irow-1
      j=ja(ic)
      a(j)=a(j)+b(ia(ic))
    end do
  end do
  ```

- Improvement of performance if \( a(j) \) is substituted by \( a1(irow) \) (direct addressing)!
- If \( b(ia(ic)) \) is constant, replace it by \( b1(ic) \) (direct addressing!)

indirect addressing example

- Laplace solver for unstructured grid:

  ```
  do i=1,n
    j1=ija(1,i)
    j2=ija(2,i)
    r=(u(j1)-u(j2))*b(i)
    a(j1)=a(j1)+r
    a(j2)=a(j2)-r
  end do
  ```
indirect addressing example (2)

- Laplace solver for unstructured grid:
  1) Multicoloring

```fortran
do icolor=1,ncolor
!cdir nodep
!cdir gthreorder
  do irow=1,istop(icolor)
    ic=ic0(icolor)+irow-1
    j1=ja(1,ic)
    j2=ja(2,ic)
    r=(u(j1)-u(j2))*b1(ic)
    a(j1)=a(j1)+r
    a(j2)=a(j2)-r
  end do
end do
```

with: $b1(ic) = b(ia(ic))$

indirect addressing example (3)

- Laplace solver for unstructured grid:
  2) JAD ordering (1/2)

```
<table>
<thead>
<tr>
<th>irow</th>
<th>ixrow</th>
<th>ipa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
  1
  12
  11
  8
  9
  5
  7
  13
  14
  15
  16
```

---

25. — Vectorization on NEC Supercomputers — 25. 25-31
indirect addressing example(4)

- Laplace solver for unstructured grid:
  2) JAD ordering (2/2)

```fortran
  do irow=1,nrow
    j=ixrow(irow)
    ul(irow)=u(j)
    a1(irow)=a(j)
  end do
  do icol=1,ncol
    do irow=1,istop(icol)
      ic=ic0(icol)+irow-1
      a1(irow)=a1(irow)+b1(ic)*
      & (ul(irow)-u[ipa(ic)])
    end do
  end do
```

with:

```
  b1(ic)=b(ia(ic))
```

indirect addressing example(5)

- Laplace solver for unstructured grid:
  Timings

  Multicoloring (no directive “gthreorder”): 112 msec
  Multicoloring (with directive “gthreorder”): 95 msec
  JAD ordering: 61 msec
vectorization example

- Vectorization of Spray module

```fortran
    do 100 idrop=1,ndrop
       do 200 while (drop_time.lt.gas_time)
          do 300 step=1,5
             compute derivatives
             update solution and drop-time
             compute error
          300     continue
          adjust drop timestep(depending on error)
          do special treatments (interactions etc.)
       200   continue
    100 continue
```

Runge-Kutta Timestep

Outermost loop running over particles: not vectorizable

vectorization example (2)

- Vectorized Implementation (1/3)

```fortran
    nndrop=ndro
    do 200 while (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
```

Reduction of drops
vectorization example (3)

- Vectorized Implementation (2/3)

```
DO 300 STEP=1,NSTEP
  DO I=1,NNDROP
    IDROP=IDROP_A(I)
    COMPUTE DERIVATIVES
  END DO
  DO I=1,NNDROP
    IDROP=IDROP_A(I)
    UPDATE SOLUTION AND DROP-TIME
  END DO
  DO I=1,NNDROP
    IDROP=IDROP_A(I)
    COMPUTE ERROR
  END DO
300 CONTINUE
```

Runge-Kutta Timestep

```
DO I=1,NNDROP
  ADJUST DROP TIMESTEP (DEPENDING ON ERROR)
END DO
```

Innermost loops running over particles: vectorizable

vectorization example (4)

- Vectorized Implementation (3/3)

```
DO I=1,NNDROP
  IDROP=IDROP_A(I)
  ADJUST DROP TIMESTEP (DEPENDING ON ERROR)
END DO
```

Innermost loops running over particles: vectorizable
**vectorization example (4)**

- Execution times (in seconds)

The vectorized implementation is 10 to 20 times faster than if original implementation

---

**Performance Analysis Tools**

- Program Execution Summary: F/C_PROGINF
- Extended UNIX Profiler: prof, loopprof
- Flow trace analysis: ftrace
- Performance tools: libperf
- ... PSUITE ...
F/C_PROGINF

- Program Execution Summary based on CPU builtin hardware counters
- performance: MOPS, MFLOPS, MIPS
- times: User, Real, System, Vector
- operations: vector, floating point...
- cache info, bank conflicts, ...
- multitasking info
- set environment variable:
  ksh: export F_PROGINF={NO|YES|DETAIL}
  csh: setenv F_PROGINF {NO|YES|DETAIL}
- in C: use C_PROGINF and compile with -hacct

prof

- Extended UNIX profiler
- link program with -p
- run program (it generates a file mon.out)
- generate profile:
  prof prog_name >profile
- info on:
  - time spent in each subroutine (including library routines)
  - for microtasked routines: time spent in each task
  - F90: number of calls of subroutine
  - ...

---

25. — Vectorization on NEC Supercomputers — 25. 25-36
**loopprof**

- Profiler on loop-level
- compile and link program with `-loopprof`
- run program (it generates a file *.pdf)
- info on:
  - time spent in each loop
  - Performance for each loop (MOPs, MFLOPs, Vector length, ...)
  - ...

**ftrace**

- simple routine based performance analysis
- compile/link with `f90 -ftrace`
- execute program
- call `ftrace`

`ftrace` output:

```
FLOW TRACE ANALYSIS LIST

Execution : Tue Mar 2 15:57:01 1999
Total CPU : 0:00'00"157

<table>
<thead>
<tr>
<th>PROG.UNIT</th>
<th>FREQUENCY</th>
<th>EXCLUSIVE</th>
<th>AVER.TIME</th>
<th>MFLOPS</th>
<th>V.OP</th>
<th>AVER. I-CACHE</th>
<th>O-CACHE</th>
<th>BANK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TIME[sec]</td>
<td>%</td>
<td>[msec]</td>
<td>RATIO</td>
<td>V.LEN</td>
<td>MISS</td>
<td>MISS</td>
<td>CONF</td>
</tr>
<tr>
<td>tst</td>
<td>0.037(61.6)</td>
<td>96.765</td>
<td>124.0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>sub</td>
<td>0.037(25.7)</td>
<td>37.258</td>
<td>350.0</td>
<td>0.00</td>
<td>99.70</td>
<td>256.0</td>
<td>0.0000</td>
<td>0.0298</td>
</tr>
<tr>
<td>subb</td>
<td>10 0.020(12.7)</td>
<td>0.0000</td>
<td>2.000</td>
<td>4520.6</td>
<td>99.57</td>
<td>256.0</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>suba</td>
<td>0.003( 1.9)</td>
<td>3.030</td>
<td>2324.3</td>
<td>0.00</td>
<td>99.39</td>
<td>256.0</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td></td>
<td>0.00000</td>
<td>0.00000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>0.157(100.0)</td>
<td>12.081</td>
<td>779.8</td>
<td>140.1</td>
<td>89.82</td>
<td>256.0</td>
<td>0.0000</td>
<td>0.0085</td>
</tr>
</tbody>
</table>
```

Simple routine based performance analysis.
**perf-tool**

- set of routines for information from hardware counters
  - usage:
    - `integer ip(34)` ! 17*8 byte wide
    - `call perf_init(ip)`
    - `call perf_start(ip)`
    - `call perf_stop(ip)`
    - `call perf_final(ip)`
- link with libperf0w.a (for IEEE)