Shared Memory Parallelization on the NEC SX-4

Dr. Erich Focht

NEC EHPCTC, Stuttgart

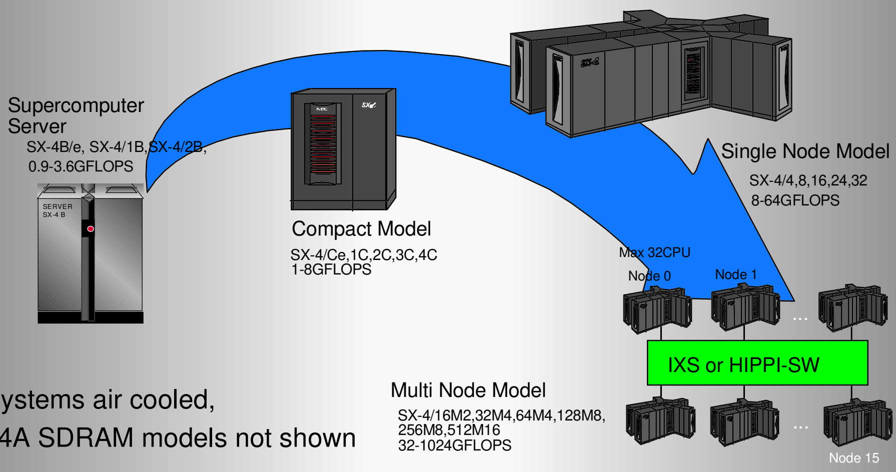
Workshop on Parallel Programming

Stuttgart, 27.1.1998

Overview

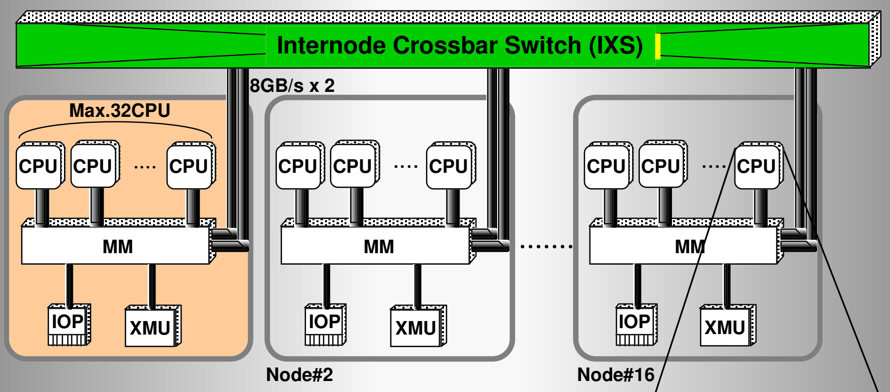
- NEC SX-4 Hardware
- Multitasking Intro
- Makrotasking
- Mikrotasking
- Autotasking
- Tools

NEC SX-4 Hardware

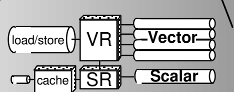


all systems air cooled,
SX-4A SDRAM models not shown

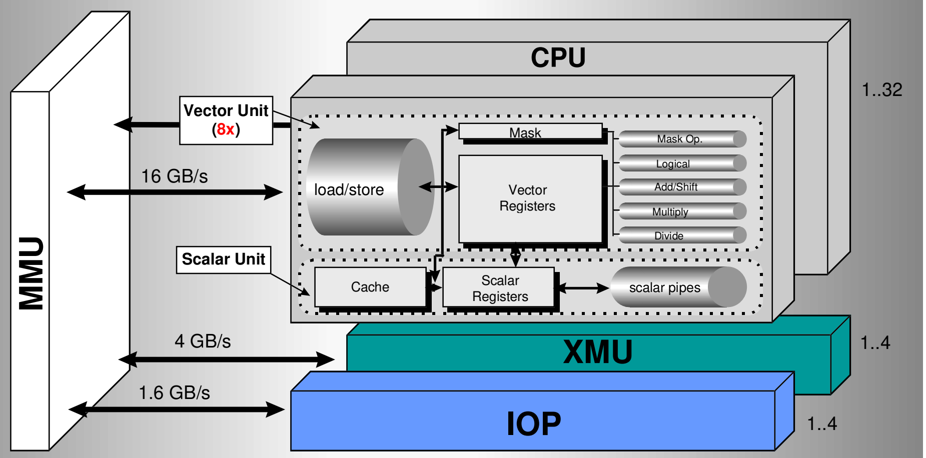
SX-4 Multi-Node System



- Max. 512 CPUs (16 nodes)
- Peak Performance : 1 Tflop/s
- Max. Memory Capacity : 256 GB (16 GB / node)



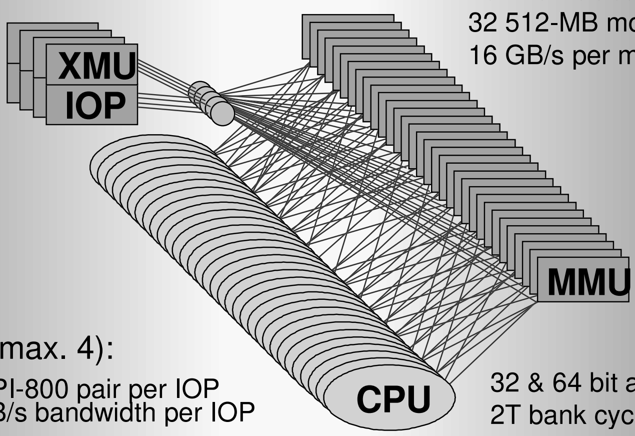
Single Node System



NEC SX-4 Crossbar

XMU: 8 GB per XMU
4 GB/s bandwidth per XMU

SSRAM technology
1024 64-bit banks
32 512-MB modules
16 GB/s per module



IOP (max. 4):
8 HIPPI-800 pair per IOP
1.6 GB/s bandwidth per IOP

32 & 64 bit access
2T bank cycle

CPU: Vector Unit

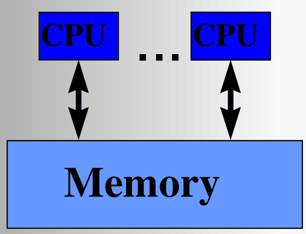
- CMOS technology, 125 MHz (8ns clock cycle)
- 4 types of vector pipelines
  - Add/Shift, Multiply
  - Divide, Logical
- 8 vector units / CPU (builtin parallelity)
  (2 Gflops peak (add/multiply))
- vector registers: (64+8) x 256 x 64 bit
- vector mask registers: 16 x 256 x 1 bit
- vector mask control register: 1 x 256 x 1 bit
- 3 floating point formats (IEEE, IBM, CRAY)

CPU: Scalar Unit

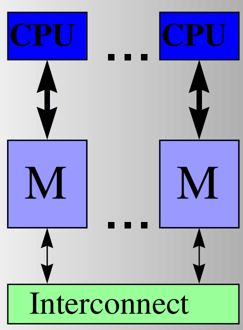
- super scalar architecture
- decodes 2 instructions / cycle
- issues up to 4 instructions / cycle
- branch prediction & out-of-order instruction issue
- 5 scalar arithmetic pipelines
- scalar registers: 128 x 64 bit
- 64 KB instruction cache
- 64 KB operand cache
- 8 KB instruction buffer

Shared Memory Architecture

•shared



•distributed



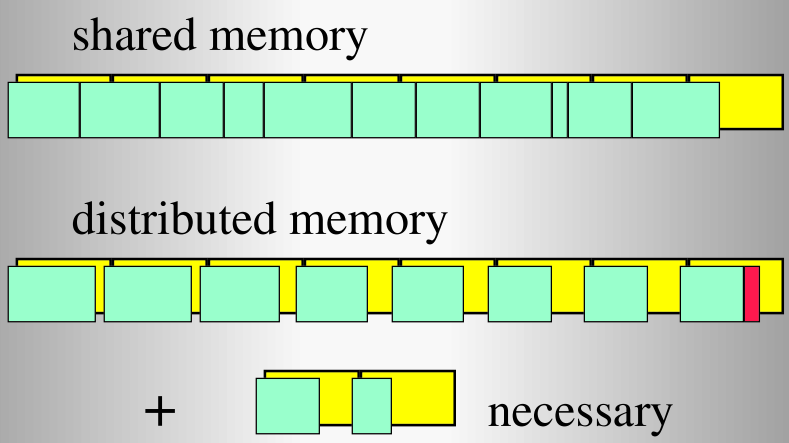
•hybrid (coupled shared memory systems)

e.g. NEC SX-4 multi-node

Advantages of Shared Memory

- advantage 1: adaptability
  - parallel work
  - parallel optimal data structuring
  - different for different portions of code or different stages in the computation
- advantage 2: only topology and geometry influence the decomposition of the physical problem
- advantage 3: easier to cope with load balancing problems
- advantage 4: better memory usage

Shared Memory in Batch Processing



Levels of Parallelism

1. data parallelism
2. parallel operations
3. parallel outer loops in loop nests
4. parallel subroutines
5. parallel processes (of different granularity)

granularity

Parallel concepts

- Program:
  - SPMD
  - Host-Node
- Implementation:
  - shared memory
  - distributed memory
  - global address space distributed memory

Parallelization on NEC SX-4

- Shared Memory
  - Macrotasking
  - Microtasking
  - Autotasking
- Distributed Memory
  - MPI
  - (PARMACS, PVM)
  - HPF
  - MPI with micro/macrotasking

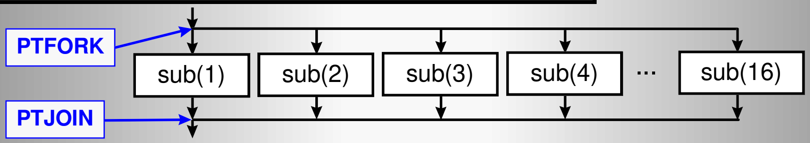
Macrotasking

- coarse grained parallelization on a top-down basis
  - subroutine
  - groups of subroutines
- task creation & synchronization performed explicitely by library calls
- FORK/JOIN, BARRIER, EVENTS, LOCKS
- compile with *-P stack* or *-P multi*
- parallelization effort: medium to large

Macrotasking: PTFORK, PTJOIN

```
program MAIN
parameter (nproc=16)
integer tid(nproc), isave(nproc)
character*32 par(nproc)
...
do I = 2, nproc
  isave(I)=I
  call PTFORK(tid(I),par(I),sub,isave(I))
end do
call sub(1)
...
do I = 2, nproc
  call PTJOIN(tid(I))
end do
...
end
```

call by reference !



Macrotask Synchronization: LOCK

- 'critical section': use lock variable as semaphore

```
subroutine SUMUP( nstart, nend, sum, a )
common / locks / ilock
real a(*)
sumlocal = 0.0
do I = nstart, nend
   sumlocal = sumlocal + a(I)
enddo
call PLLOCK( ilock )
sum = sum + sumlocal
call PLUNLOCK( ilock )
return
end
```

- additional calls outside the parallel region needed:
  - `call PLASGN(ilock)` : assign a lock to variable `ilock`
  - `call PLFREE(ilock)` : release the lock variable `ilock`
- check lock status: `IPLSTAT(ilock)`, `IPLSTATL(ilock)`

Macrotask Synchronization: BARRIER

- synchronous barrier control

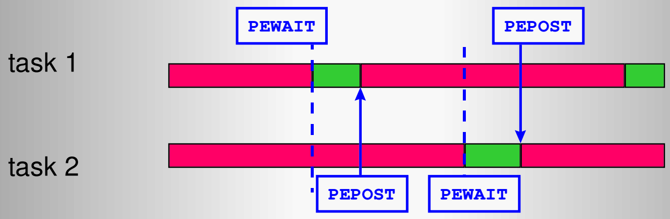
```
program MAIN
common /barr/ ibarr
...
call PBASGN(ibarr,NCPU)
do i=2,NCPU
   call PTFORK(tid(I),par(I),sub)
enddo
...
call PBSYNC(ibarr)
...
call PBFREE(ibarr)
...
```

```
subroutine SUB
common /barr/ ibarr
...
call PBSYNC(ibarr)
return
end
```

- related function call:
  - `IPBSTAT(ibarr)` : check status of barrier `ibarr`

Macrotask Signalling: EVENTS

- subroutine & function calls:
  - `call PEASGN(ievnt)` : assign an event
  - `call PEWAIT(ievnt)` : wait for an event
  - `call PEPOST(ievnt)` : post an event
  - `call PECLEAR(ievnt)` : clear an event
  - `call PEFREE(ievnt)` : release an event
  - `I=IPESTAT(ievnt)` : check the status of an event



Other Macrotasking Intrinsics

- subroutine & function calls:
  - `call PTPARAM(tpar)` : fetches task parameters
  - `I = IPTSTAT(tvar)` : checks subtask status
  - `I = IPTID()` : returns task identification number
  - `I = IPSMAX()` : returns max. number of physical tasks
  - `I = IPTNAP()` : returns no. of CPUs
  - `call PTLIST([unit])` : outputs a list showing task states
  - `call PSTUNE(key,val,...)` : change tuning parameters of task scheduler
- cf. *Fortran77/SX Multitasking User's Guide*

Macrotasking: Remarks

- increase the size of a task as much as possible
- reduce synchronization and exclusive control (locks) to improve the performance
- be careful about data scoping:
  - private / shared data
  - use locks when accessing shared data
- preferably use SPMD
  - load balancing
  - simplicity
  - replace EVENTs by BARRIERs

Multitasking Memory Management

- Task shared data:
  - COMMON or GLOBAL COMMON data
  - data specified in SAVE statement
  - actual arguments of PTFORK and microtask subroutines (microtask shared)
  - initialized data
  - data declared in LOCAL COMMON statements (microtask shared)
- Task private data
  - LOCAL COMMON data (macrotask private, cf. compiler options)
  - allocated on stack
  - stack size is estimated by the linker for each task
  - stack space is dynamically allocated at runtime for each task

Microtasking

- parallelization on a bottom-up basis
- SPMD
- parallelism of:
  - loop iterations (PARDO)
  - statement groups (PARCASE)
  - eventually subroutine calls
- task creation and synchronization is implicitly done by using compiler directives in the code
- work assignement is done automatically
- parallelization effort: medium to small

Microtasking: Directives

- FORTRAN 77

  `*PDIR ...                (1st column)`
  - compile with: **`f77 -P multi`** or **`f77sx -multi`**
- FORTRAN 90

  `!CDIR ...`
  - F77 directives also understood
  - compile with: **`f90 -P multi`**
- C, C++
  - #pragma pdir ...
- directive must be placed directly before the starting line of the code it refers to
- lower case allowed, delimiters: comma, blank

Microtasking: Task Creation

***PDIR RESERVE[=N]**

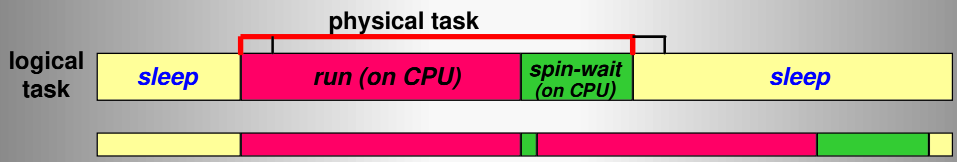
- creates N logical tasks, if N is omitted, the number of tasks created is:
  - specified by the **-reserve** compiler option, or
  - given in **$F_RSVTASK** (environment variable), or
  - MAXCPU in the RSG or no. of physical processors
- similar to PTFORK

***PDIR RELEASE**

- requests release of the logical tasks (of the microtasking group)
- similar to PTJOIN
- synchronization at `RELEASE`

Microtasking: Task Creation (2)

- `RESERVE/RELEASE` ☞ overhead
- if `RESERVE/RELEASE` are omitted, logical tasks are created and released at the beginning and the end of each parallel section
- try `RESERVE/RELEASE` only once, in the MAIN program
- only subroutines containing `PARDO` or `PARCASE` directives are executed parallely
- scheduling:



Microtasking: PARDO FOR

***PDIR PARDO FOR[=N]**

- iterations of following loop are executed in parallel, each task will be assigned 1/N of the total number of iterations

```
*pdir pardo for=4
      do i=1,16
         a(i)=a(i)+b*c(i)
      enddo
```

task	1	2	3	4
	1 2 3 4	5 6 7 8	9 10 11 12	13 14 15 16

Microtasking: PARDO BY

***PDIR PARDO BY[=N]**

- iterations of following loop are executed in parallel, each task will be assigned N iterations, until all iterations have been completed
- default: BY=1
- interaction with scheduler at start and end of each work chunk

```
*pdir reserve=4
...
*pdir pardo by=2
      do i=1,16
         a(i)=a(i)+b*c(i)
      enddo
```

task	1	2	3	4
	1 2	3 4	5 6	7 8
	9 10	11 12	13 14	15 16

Microtasking: PARDO barrier

`*PDIR PARDO FOR, NOBARR=(ENTRY,EXIT)`

- switch off barrier at start or end of the DO loop
- faster (e.g. for successive loops)
- be careful!

```
      subroutine sub(a,b,c,N)
      real a(N),b,c(N)
      ...
*pdir pardo for, nobarr=(entry,exit)
      do i=1,N
         a(i)=a(i)+b*c(i)
      enddo
      ...
      return
```

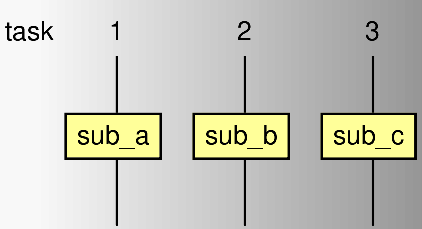
synchronization on entry to subroutine

synchronization on exit from subroutine

Microtasking: PARCASE

- blocks defined by PARCASE/CASE/ENDCASE are executed in parallel
- weighting of blocks possible (different no. of logical tasks per block) **`(*PDIR PCASE WEIGHT=N)`**
- barrier handling as for **`PARDO (NOBARR=(ENTRY,EXIT))`**

```
*pdir parcase
      call sub_a
*pdir case
      call sub_b
*pdir case
      call sub_c
*pdir endcase
```



Microtasking: Critical Section

- code block between `CRITICAL` and `END CRITICAL` is executed only by one task at a time
- example: reduction

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


Microtasking: Serial Section

- code block between `SERIAL` and `END SERIAL` is executed only by one task
- other tasks are waiting until the serial block execution is completed
- example: reduction

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

Microtasking: Other Directives

`*PDIR BARRIER` : specify a barrier

`*PDIR WAIT[=N]/POST[=N]` : synchronization

`*PDIR EXIT` : terminate parallel processing

`*PDIR SAVE [var,...]` : allocate data to the static area

`*PDIR PARTFLUSH` : partial cache flush (not .stack section)

`*PDIR PRIVATE (var,...)` : specified variables are replaced by local work variables. The values of the original variables are not taken over by the work variables. The values of the work variables are not taken over by the original variables after the PARDO or PARCASE section.

`*PDIR SUPPOSE[=(RESERVERD,NOCONCALL,NONESTCALL)]` : skip some tests when entering a microtask procedure

- further reading: *Fortran77/SX Multitasking User's Guide*

Autotasking: concept

- Pacific Sierra preprocessor 'fopp', ('copp' for C)
  - also used by other vendors!
- automatically inserts *pdir
  - for loop-nests
  - for inner loops (needs directive or switch)
  - apart from that some optimizations are performed as well
  - can be used as a learning tool for microtasking
- additional control by *odir (f77) or !cdir (f90) directives
- parallelization effort: small

Autotasking: concept (2)

- parallel regions are placed into new subroutines
- can handle sequential and critical regions automatically
- subroutines already containing *pdir directives are skiped!
- Data scoping:
  - shared variables are passed as arguments to new subroutines
  - private variables are declared locally within the new routines
- f77 -P auto [-Wo"-eci -fi -K0"]
- f77sx -multi -fopp par [e=i res=whole for=0]

Autotasking Example: Threshold Test

- original

```
DO J=1,M
   DO I=1,N
      A(I,J)=0.0
   ENDDO
ENDDO
```

- translation

```
  IF (M*N.GT.1666)THEN
     CALL T$1(M,N,A)
  ELSE
     DO J=1,M
        DO I=1,N
           A(I,J)=0.0
        ENDDO
     ENDDO
  ...
  SUBROUTINE T$1(M,N,A)
  INTEGER M,N,I,J
  REAL A(999,999)
*PDIR PARDO FOR=4
  DO J=1,M
     DO I=1,N
        A(I,J)=0.0
     ENDDO
  ENDDO
  RETURN
  END
```

Autotasking Example: Reduction

- original

```
DO J=1,M
   DO I=1,N
      S=S+A(I,J)
   ENDDO
ENDDO
```

- translation

```
  IF (M*N.GT.1666)THEN
     CALL R$1(A,N,M,S)
  ELSE
  ...
  SUBROUTINE R$1(A,N,M,S)
  INTEGER M,N,I,J
  REAL A(M,N),S,S1
  S1=0
*PDIR PARDO FOR=4
  DO J=1,M
     DO I=1,N
        S1=S1+A(I,J)
     ENDDO
  ENDDO
*PDIR CRITICAL
  S=S+S1
*PDIR END CRITICAL
  RETURN
  END
```


Autotasking: Inner Loops

- original

```
*ODIR INNER
  DO I=1,N
   A(I)=SQRT(B(I)**2+C(I)**2)
  ENDDO
```

- default: parallelize outer loop

- translation

```
  IF (N.GT.294)THEN
     CALL S$1(N,B,C,A)
  ELSE
  ...
  SUBROUTINE S$1(N,B,C,A)
  INTEGER N,I
  REAL B(N),C(N),A(N)
*VDIR NODEP
*PDIR PARDO FOR=4
  DO I=1,N
     A(I)=SQRT(B(I)**2+C(I)**2)
  ENDDO
  END
```

Autotasking: fopp Options

- passed by f77 / f90 with -Wo"..."
- very numerous!
  - man f77
  - FORTRAN 77/SX Programmer's Guide (chapter 6)
- transformation control switches: -e switch (enable), -d switch (disable)
  - -ec : parallelize loops
  - -ei : paralleize inner loops if no outer loop available
  - -e0 : generate runtime threshold test for parallel loops
- optimization control switches: -g switch (enable), -f switch (disable)
  - -fi : reserve / release only in MAIN program
- -K nnn : parallelize loops with PARDO FOR=nnn
- -L nnn : parallelize loops with PARDO BY=nnn
- -T nnn : minimum parallelization threshold
- -O {1|2|3} : optimization level

Autotasking: fopp Directives

ODIR[F,R,L] *directive

- F: directive valid for the whole file
- R: valid for the routine
- L: valid for the next loop

- *ODIR (NO)CONCUR : switch on/off parallelization
- *ODIR SKIP : fopp skips parts of the code
- *ODIR (NO)VECTOR : switch on/off vector optimizations
- *ODIR INNER : parallelize inner loop(s)
- *ODIR (NO)THRESHOLD : switch on/off threshold code generation
- *ODIR (NO)SYNC : do(n't) ignore potential overlaps of array sections
- ...

Autotasking: fopp Directives (2)

- *ODIR CNCALL : allow concurrent calls in loops

```
*odir cncall
    do i = 1, nproc
       call sub1(I,a(I))
    end do
```

- almost like macrotasking, but more efficient
- *ODIR UNSHARED (arrayname,...) : analogue to *PDIR PRIVATE
- ... vectorization directives ...

Performance Analysis Tools

- Program Execution Summary : F/C_PROGINF
- Extended UNIX Profiler : prof
- Analyzer: fanp, fan90, ca
- ... 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
  - ...

Analyzer

- detailed information up to instruction level
  - execution time
  - statement count
  - static analysis
- causes some overhead (2x - 8x)
- restart capability useful to use fanp together with make
- refer to man-pages and manuals

  Analyzer-p/SX Reference Manual

  Analyzer90/SX Reference Manual

  C-Analyzer/SX User's Guide

NEC