



# NTNU

Innovation and Creativity

## Njord Advanced Usage

IBM AIX Power5

Jørn Amundsen, Vegard Eide

NTNU IT

2010-09-01

2

## Contents

- 1 Introduction to Power hardware
- 2 Compiling programs
- 3 Libraries
- 4 Runtime considerations
- 5 Debugging
- 6 Profiling
- 7 Advanced I/O
- 8 Efficient use of the queueing system
- 9 Documentation

3

## Introducton to Power hardware

Cache and memory

Memory hierarchy

SMT

▶ TOC

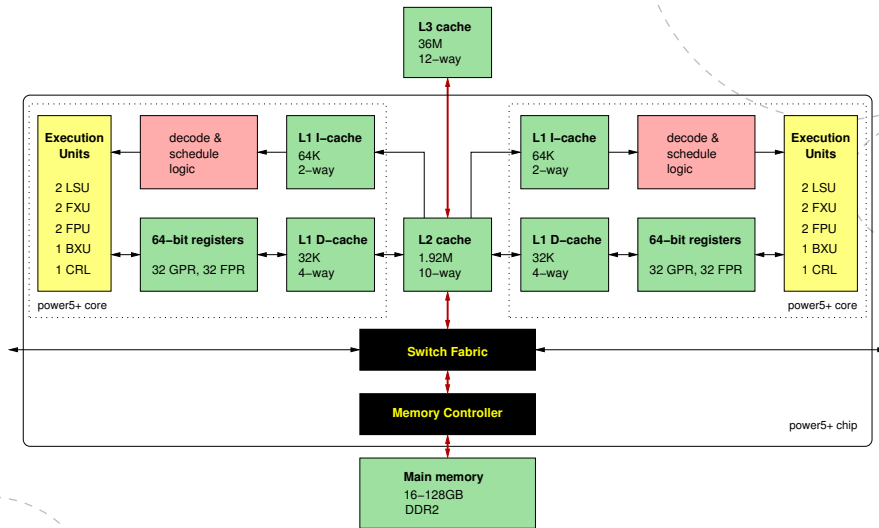
4

## Cache and memory

- 16 x 64-bit word cache lines (32 in L3)
- Hardware cache line prefetch on loads
- Reads from memory are written into L2
- External L3, acts as a victim cache for L2
- L2 and L3 are shared between cores
- L1 is write-through
- Cache coherence is maintained system-wide at L2 level

5

## Cache and memory (2)

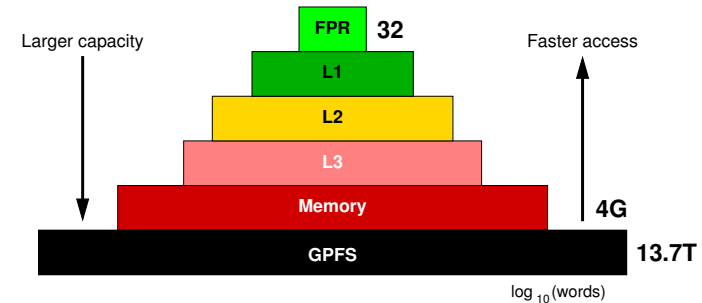


www.ntnu.no

Jørn Amundsen, Vegard Eide, NTNU IT

6

## Memory hierarchy



www.ntnu.no

Jørn Amundsen, Vegard Eide, NTNU IT

7

## SMT

- In a concrete application, the processor core might be idle 50-80% of the time, waiting for memory
- An obvious solution would be to let another thread execute while our thread is waiting for memory
- This is known as *hyper-threading* in the Intel/AMD world, and **S**imultaneous **M**ultithreading (SMT) with IBM
- SMT is supported in hardware throughout the processor core
- SMT is more efficient than hyper-threading with less context switch overhead
- Power5 and 6 supports 1 thread/core or SMT with 2 threads/core, while the latest Power7 supports 4 threads/core
- SMT is enabled on a node with the (privileged) command `smtctl`

www.ntnu.no

Jørn Amundsen, Vegard Eide, NTNU IT

8

## SMT (2)

*When should I use SMT and how do I enable it?*

- SMT is beneficial if you are doing a lot of memory references, and your application performance is memory bound
- Enabling SMT doubles the number of MPI tasks per node, from 16 to 32. Requires your application to be sufficiently scalable.
- It is not useful if your application has a small memory footprint and keeps the functional units (integer or floating-point) busy all the time
- SMT is only available in batch, with the SMT feature:

```
#@ requirements = ( Feature == "SMT" )
```

www.ntnu.no

Jørn Amundsen, Vegard Eide, NTNU IT

## Compiling programs

### XL compilers

#### Compiling programs

#### Floating-point options

#### Performance options

#### Optimization macros

#### Using `-qstrict`

#### 32- or 64-bit compiles

► TOC

## XL compilers

It is the same compiler (Fortran or C), only default settings for options changed (e.g. source code format, file suffix)

Language	Serial	MPI threadsafe	OpenMP
Fortran 77	xlf	mpxlf_r	xlf_r
Fortran 90	xlf90	mpxlf90_r	xlf90_r
Fortran 95	xlf95	mpxlf95_r	xlf95_r
C	xlc	mpcc_r	xlc_r
C++	xlc	mpCC_r	xlc_r

Check the configuration file `/etc/xlf.cfg.53` to see default options

## Compiling programs

### OpenMP programs

specify `-qsmp=omp` with a thread-safe compiler invocation, e.g.

```
$ xlf90_r -qsmp=omp:noauto omp.f90
$ xlf_r -qsmp=omp:noauto -qnosave omp.f
```

### MPI programs

the "mp..." commands are wrapper scripts that invoke the appropriate xlf (or xlc) compiler. MPI library are automatically linked in, i.e. `-lmpi` is not needed

```
$ mpxlf90 mpi.f90
```

### Mixing OpenMP and MPI

use a thread-safe "mp..." compiler invocation with `-qsmp=omp`, e.g.,

```
$ mpxlf90_r -qsmp=omp mpi_omp.f90
```

## Floating-point options

Bytes of storage allocated for scalar variables of a given type with specified compiler options

	default	-qrealsize=8	-qautodbl=dbl4	-qautodbl=dbl8	-qautodbl=dbl
REAL	4	8	8	4	8
REAL(4)	4	4	8	4	8
REAL(8)	8	8	8	16	16
DOUBLE PRECISION	8	16	8	16	16
COMPLEX	8	16	16	8	16
COMPLEX(4)	8	8	16	8	16
COMPLEX(8)	16	16	16	32	32
DOUBLE COMPLEX	16	32	16	32	32

## Floating-point options (2)

```
real(8) :: var1, var2, var3
var1 = dmax1(var2, var3)
end
```

```
$ xlf90 test.f90
```

...will work, but

```
$ xlf90 -qrealsize=8 test.f90
```

```
"test.f90", line 2.14: 1513-041 (S) Arguments of the wrong
type were specified for the INTRINSIC procedure "dmax1"
```

...fails because double precision function `dmax1` is promoted to 16 bytes while `var2` and `var3` remains 8 bytes. Use generic names for intrinsics!

## Performance options

`-qarch=auto`

automatically detects the processor architecture of the compiling machine for which the code (instructions) should be generated

`-qtune=auto`

generates object code optimized for the hardware platform on which the program is compiled

`-qhot`

high order transformations (blocking and loop transformations)

`-qipa`

interprocedural analysis (IPA)

`-qinline`

subroutine in-lining

`-qsmp`

enables automatic (shared memory) parallelization and optimization of program code

## Optimization switches

option `-O3`:

- `-O2`
- memory and compile-time intensive optimizations

option `-O4`:

- `-O3`
- `-qhot`
- `-qipa=level=1`

option `-O5`:

- `-O4`
- `-qipa=level=2`

## Using `-qstrict`

- The `-qstrict` option ensures that any optimization done will not alter the semantics of a program, and that the numeric results of a program will be identical with those produced by an unoptimized program.
- `-qstrict` suppresses many optimizations, such as
  - Common expressions
  - Loop exchange
  - Inner loop unrolling

```
do i = 1, n
  do j = 1, n
    x(j)*a(j,i) ...
  end do
end do
```

→

```
do i = 1, n
  do j = 1, n, 2
    x(j)*a(j,i) ...
    x(j+1)*a(j+1,i) ...
  end do
end do
```

## 32- or 64-bit compiles

- On Njord, the IBM and GNU compilers are configured to make 64-bit executables by default
  - it is the recommended setup for HPC users
  - it is not the default with AIX
- This is achieved by presetting the `OBJECT_MODE` environment variable
- The toolchain components (`xlc/xlf`, `gcc`, `ar`, `ld`, `nm`, etc.) reads this variable to determine if 32- or 64-bit format should be applied
- **`OBJECT_MODE` should always be 64 (or 32) !**
- 32 or 64 bit usually does not matter, except for in a few but important cases
- There are more caveats for C than Fortran users

## 32- or 64-bit compiles (2)

- **32-bit executables**
  - addresses (and by default integers) are 32 bits
  - have limited memory addressing capability (4 GiB)
  - stack size is limited to 768 MiB
  - have limited shared memory segment addressability (not enough memory now)
  - (integer) registers are 32 bits wide
  - AIX utilities are 32-bit by default
- **64-bit executables**
  - addresses (and by default integers) are 64 bits
  - have unlimited (e.g. as defined in `/etc/security/limits`) addressing, stack and shared library addressing capabilities
  - (integer) registers are 64 bits wide
  - larger stack frames and slightly larger executables
  - use slightly more memory BW

## 32- or 64-bit compiles (3)

- Libraries can contain a mix of 32- and 64-bit objects
  - try: `ar -t -Xnn /usr/lib/libpapi.a, nn=32,64.`
- Might experience linking problems if a library lacks 64-bit support
- Might experience 32/64-bit linking problems if inadvertently linking executables of wrong format:
  - `XCOFFnn` object files are not allowed in `mm`-bit mode.
- Might compile for a specific format with `xlc/xlf -qnn` or `gcc -maixnn`
- CPP automatically defines the symbol `__64BIT__` if compiling 64-bit (the default)

## 32- or 64-bit compiles (4)

What is wrong with this code ?

```
njord$ cat allocx.c
void allocx(int n, double **x)
{
    *x = (double *)malloc(n*sizeof(double));
}
njord$
```

# Libraries

ESSL

PESSL

MASS

▶ TOC

# ESSL

- ESSL (**E**ngineering and **S**cientific **S**ubroutine **L**ibrary) is a collection of mathematical subroutines tuned for performance
  - Linear Algebra Subprograms
  - Matrix Operations
  - ...and more
- specify `-lessl` at the linking stage
 

```
$ xlf90 prog.f90 -lessl
```
- ESSL contains a subset of routines that correspond to the standard set of LAPACK. The rest of the LAPACK routines can be linked from the `lapack` library
 

```
$ module load lapack
$ xlf90 prog.f90 -llapack -lessl
```
- adding `-qessl -lessl` allows the compiler to substitute ESSL routines in place of Fortran 90 intrinsics

# PESSL

- Parallel ESSL (PESSL) is a scalable mathematical subroutine library that supports parallel processing applications using SPMD programming model, i.e. tasks doing the same set of instructions on different sets of data
- for computations, PESSL uses the ESSL subroutines
- for communication, PESSL includes the Basic Linear Algebra Communications Subprograms (BLACS), which use MPI
- compile/link
 

```
$ mpxlf_r prog.f -lesslsmpl -lpesslsmpl -lblacssmpl
```
- at runtime specify the number of threads, for pure MPI programs set `OMP_NUM_THREADS=1`

# MASS

- MASS (**M**athematical **A**cceleration **S**ubsystem) consists of mathematical functions tuned specifically for optimum performance on POWER architectures
- include both scalar and vector functions
- intended for use in applications where slight differences in accuracy can be tolerated
- vector version need code change, e.g.
 

```
y = exp(x) --> call vexp(y, x, n)
```
- ...but higher order optimizations (`-O3 -qhot` and above) does this automatically
- compile/link
 

```
$ xlf90 prog.f90 -lmass -lmassvp5
```

## Runtime considerations

Running programs

XLFRTEOPTS

POE environment variables

Multiple page size

▶ TOC

## Running programs

Running serial and SMP programs

- type the name of the executable:

```
$ ./a.out
```

- for SMP programs specify the number of threads with the `OMP_NUM_THREADS` variable (by default this is set to the number of cpus per node), e.g.

```
$ env OMP_NUM_THREADS=4 ./a.out
```

- Fortran runtime can be controlled by XLFRTEOPTS settings

## XLFRTEOPTS

- options for I/O, EOF error-handling, the specification of random-number generators, e.g.

- `namelist={new | old}`
- `ufmt_littleendian={units_list}`, e.g.

```
$ export XLFRTEOPTS=ufmt_littleendian=2
```

- settings can be specified in a program using the `setrteopts` subroutine. This will override the environment variable

```
call setrteopts("namelist=new:ufmt_littleendian=2")
```

## Running programs (2)

Running MPI programs (interactively)

- start the executable with the `poe` command. Specify the POE runtime environment using `poe` flag options or environment variables. E.g.

```
$ poe ./a.out -hostfile hfile -procs 4
```

or equivalent

```
$ env MP_PROCS=4 MP_HOSTFILE=hfile poe ./a.out
```

- running interactively without specifying the hostfile gives

```
ERROR 0031-808 Host file or pool
number must be used to request nodes
```

## POE environment variables

### MP\_LABELIO

output from the parallel tasks are labeled by task id

```
$ poe ./a.out -procs 3 -labelio yes
1: Hello, I am 1 of 3
0: Hello, I am 0 of 3
2: Hello, I am 2 of 3
```

### MP\_STDINMODE, MP\_STDOUTMODE

determines how STDIN and STDOUT is managed for the tasks, e.g.

```
$ poe ./a.out -procs 3 -stdoutmode 2
Hello, I am 2 of 3
```

## MPI point-to-point communication

### Eager Protocol

In the Eager protocol, the sender process, eagerly sends the entire message to the receiver via a protocol managed buffer (early arrival buffer) and the MPI send is marked as complete. In order to achieve this, the receiver needs to provide sufficient buffers to handle incoming messages. This protocol has minimal startup overheads and is used to implement low latency message passing for smaller messages

### Rendezvous Protocol

The Rendezvous Protocol negotiates the buffer availability at the receiver side before the message is actually transferred, i.e. the MPI send will block until a matching receive is posted. This protocol is used for transferring large messages when the sender is not sure whether the receiver actually has the buffer space to hold the entire message

## POE environment variables (2)

### MP\_BUFFER\_MEM

- default value for `MP_BUFFER_MEM` is 64 MB. This can be increased to 256 MB
- if the `MP_BUFFER_MEM` is set to high and the available memory is not enough (e.g. `ConsumableMemory` is set too small) this can result in error

```
"ERROR: 0031-309 Connect failed during message
passing initialization, task NN, reason:
There is not enough memory available now.
ERROR: 0031-007 Error initializing communication
subsystem: return code -1"
```

## POE environment variables (3)

### MP\_EAGER\_LIMIT

- The following will work or deadlock depending on the size of the message and the `MP_EAGER_LIMIT` value:

```
if (rank == 0) then
  call mpi_send(...message to rank 1...)
  call mpi_recv(...message from rank 1...)
elseif (rank == 1) then
  call mpi_send(...message to rank 0...)
  call mpi_recv(...message from rank 0...)
endif
```

- The default for value `MP_EAGER_LIMIT` depends on the number of mpi tasks. The maximum value is 256 KB. A value of zero bytes is valid, and indicates that eager send mode is not to be used for the job. This can be used to verify that the application is MPI-safe

## Multiple page size

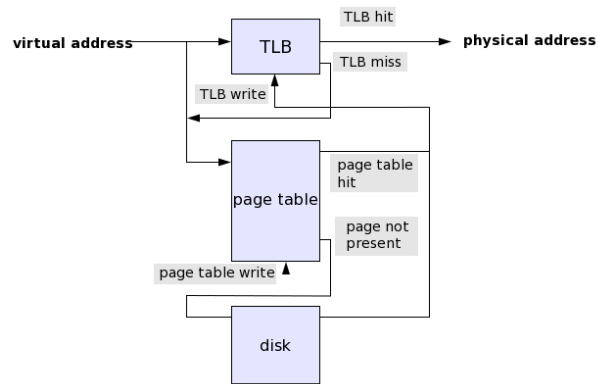


Figure: Virtual to physical address translation process

## Multiple page size (2)

Default page size is 4KB. Reasons for increasing page size

- keep prefetch going
- avoid TLB misses

Page sizes can be specified for a process's address space using the `LDR_CNTRL` environment variable, or the `ldedit` or `ld` commands.

E.g. specifying 64KB pages when building a model, in makefile:

```
LDFLAGS= -btextsize:64K -bdatapsize:64K -bstacksize:64K
```

or in an existing executable (`prog`):

```
ldedit -btextsize=64K -bdatapsize=64K -bstacksize=64K prog
```

or in a job script:

```
export LDR_CNTRL=DATAPSIZE=64K@TEXTPSIZE=64K@STACKPSIZE=64K
```

## Debugging

**Compiler switches**

**Trapping FPE's**

**IBM dbx/pdbx**

**Totalview**

**Memory debugging**

► TOC

## Compiler switches

`-g`

generates debug information for debugging tools

`-qfullpath`

records the full or absolute path names of source and include files in object files compiled with debugging information

`-C`

run-time checking of array bounds

`-qfltrap[=<suboptions_list>]`

detect and trap runtime floating-point exceptions

`-qsigtrap`

produces a traceback and stops the program. Include the `-g` option to get line numbers in the traceback listings.

`-qinitauto`

initializes uninitialized automatic variables to a specific value

## Trapping FPE's, example 1

```

real :: a, b
a = 1.0; b = 0.0
write(*,*) a/b
end

$ xlf90 prog.f90; ./a.out
INF

$ xlf90 -qflttap=enable:zerodivide prog.f90; ./a.out
Trace/BPT trap (core dumped)

$ xlf90 -qflttap=enable:zerodivide -qsigtrap -g prog.f90
$ ./a.out
Signal received: SIGTRAP - Trace trap
FP division by zero
Traceback:
Offset...in procedure _main, near line 3 in file prog.f90

```

## Trapping FPE's, example 2

Debugging programs with uninitialized variables. Specify NAN (hexadecimal number FF) for the initial value and compile with trapping

```

real :: a, b, c, d
a = 1.0
b = 2.0
d = c + a*b
end

$ xlf90 -g -qinitauto=FF -qflttap=nanq:enable -qsigtrap prog.f90
$ ./a.out
Signal received: SIGTRAP - Trace trap
NaN detected
Traceback:
Offset...in procedure _main, near line 4 in file prog.f90

```

## IBM dbx/pdbx

- Occasionally, programs fail
- You might want to know why
- A traditional approach is to insert printouts to track progress
  - I am here
  - Now I am here ...
- This is time consuming and might also change program behavior
- There are several tools to investigate program errors and/or performance bottlenecks
- The **dbx** and **pdbx** programs are simple command-line tools which might be a first approach before diving into Totalview ®
- They are similar to **gdb** on Linux, although **dbx** is the traditional Unix debugger

## Introducing dbx

- Compile with `-g`, or preferably not higher than `-g -O2`, add FPE trapping switches if applicable
- **Caveat**, `-O0` runs slowly, you might want to only compile parts of the application w/o optimization
- Post mortem investigation is done by compiling with `-g`, running your application in **batch** and looking into the `core` file with
 

```
$ dbx myprog core
```
- **Core files are limited to 1 GiB interactive and 120 GiB in batch**
- Among the most important you can do is to print call stacks, step through the program and investigate memory (variables)
- The most useful commands are shown by typing `alias` after startup, then `help usage` or `help` on any command

## Sample dbx session

```
f05n071:src$ dbx a.out core
...
Trace/BPT trap in main at line 19
   19          x = sqrt(0.5 - x);
(dbx) alias
p      print
l      list
...
(dbx) help stop
stop if <condition>
stop at <line-number>          [if <condition>]
stop in <procedure>           [if <condition>]
stop <variable>               [if <condition>]
stop <variable> at <line-number> [if <condition>]
stop <variable> in <procedure>  [if <condition>]
stop on load ["<module>"]      [if <condition>]
stop on load ["<module>(<member>)"] [if <condition>]
Stop execution when the given line is reached,
procedure or function entered, variable changed,
module loaded or unloaded, or condition true.
```

## Sample dbx session (2)

- A sample (non-core) debug session starts by invoking dbx  
\$ dbx a.out
- Then set some breakpoints with  
(dbx) stop at *a\_line\_number*  
(dbx) stop in *my\_function*
- Next run to the first “interesting” point with  
(dbx) run [*my\_args*]
- Continue with *c*, *n* or *s* (continue, step over or step into)

## Examining data with dbx

- The most important feature with **dbx** is the ability to examine data with arbitrary interpretation
- This is documented online with the (undocumented)  
(dbx) help display
- The syntax is  
*<address>*, *<address> / [<mode>] [> <filename>]*  
*<address> / [<count>] [<mode>] [> <filename>]*
- Similar to the **gdb** *x* command (examine), but different syntax
- Consider two C variables, `static int one = 1` and  
`double x[N] = 0, 1, 22, 32, ..., (N - 1)2.`

## Examining data with dbx (2)

```
(dbx) &one/D
0x0000000110000700:  1
(dbx) &one/4h
0x0000000110000700:  00 00 00 01

(dbx) &x[8]/8g
0xffffffffffff460:  64 81
0xffffffffffff470:  100 121
0xffffffffffff480:  144 169
0xffffffffffff490:  196 225

(dbx) &x[8]/8llx
0xffffffffffff460:  4050000000000000 4054400000000000
0xffffffffffff470:  4059000000000000 405e400000000000
0xffffffffffff480:  4062000000000000 4065200000000000
0xffffffffffff490:  4068800000000000 406c200000000000
```

D: integer, h: byte, g: double, llx: long long hex

## Introducing pdbx

- This debugger is a simple POE command line extension on top of `dbx`, to allow for debugging of parallel processes
- Useful as a tool to investigate live MPI hangs, interactive or batch
- Also useful as a lightweight alternative to Totalview
- Requires a POE hostfile or pool, e.g. environment variable `MP_HOSTFILE` and `MP_HOSTFILE` should be set
- Application compile requirements as for `dbx`
- Most important extensions `attach [all]` and `detach`
- Use `detach` after `attach`, dont use `q` to quit!
- Use `dbx` or `Totalview` on POE core dumps, directly on `coredir.<taskid>/core`

## Attaching to a job with pdbx

- In order to attach to a running job, batch or interactively, we need to identify the node running the `poe` process, and if necessary log into that node with `rsh`
- Interactively we must be on the right login node, in batch it usually is the first node in the node LoadLeveler node list (e.g. from `llq`)
- After attaching we can use the usual `dbx` commands, now collectively on every attached task

```
f05n071:src$ ps -fujjoern|egrep 'PID|poe'
      UID      PID      PPID      C      STIME      TTY      TIME  CMD
joern 205302 422598      0 09:46:53 pts/11    0:00 /bin/sh
/usr/linux/bin/egrep PID|poe
joern 295826 520776      1 09:46:52 pts/22    0:00 poe
f05n071:src$ pdbx -a 295826
...
pdbx(none) attach all
...
pdbx(attached) detach
f05n071:src$
```

## Totalview

- compile with debug options
 

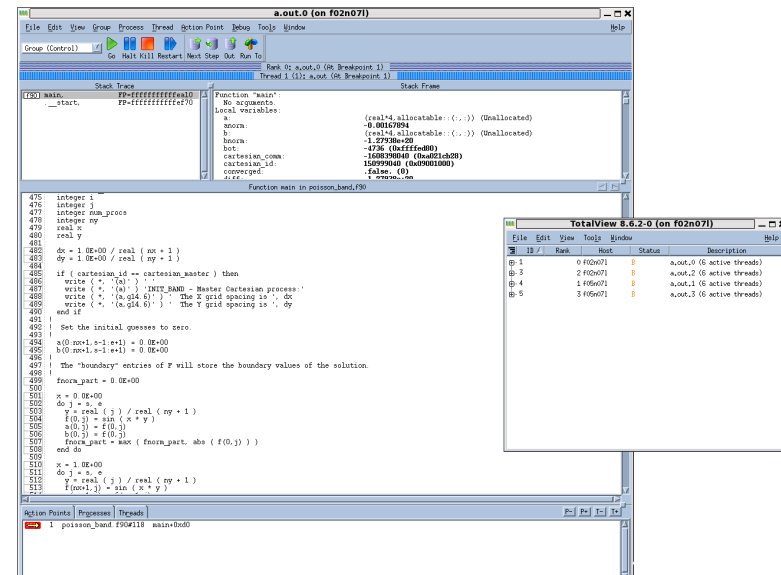
```
$ xlf90 -g -qfullpath -o prog
```
- load the totalview module (environment)
 

```
$ module load totalview
```
- start Totalview
 

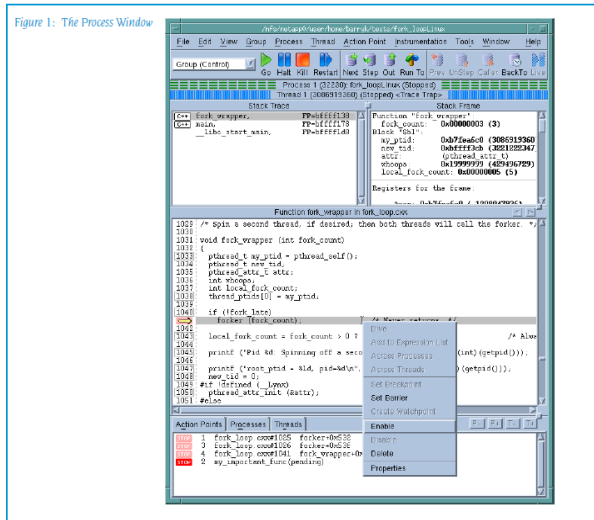
```
$ totalview prog
```
- debugging a core file
 

```
$ totalview prog core
```

## Totalview (2)



## Totalview (3)



## Memory debugging

- Dynamic memory errors are among the hardest ones to find on Unix systems
- The most frequent errors are due to overwrites of allocated memory or by accessing freed memory.
- Memory errors tend to corrupt the malloc subsystem
- The malloc subsystem is used with C `malloc()` and Fortran `allocate`, in addition to most library functions like C `printf()`, Fortran formatted read/write etc.
- Often, the error hit you at a point totally unrelated to the point it was made
- This can make it extremely difficult to track down the root cause of a memory error

## Memory debugging (2)

- A number of tools have been developed over the years, with SunOs Purify, later Rational Purify <sup>®</sup>, being among the most known
- Rational Purify is available on njord with `module load purify`
- Not very useful with AIX, due to problems with debugging parallel applications
- Linux systems provide several checkers, like the Open Source *Electric Fence* (efence)
- **Electric Fence is not useful with AIX**
- AIX malloc provides an internal malloc debugging tool

## AIX debug malloc tool

- AIX memory debugging is applied through two environment variables, `MALLOCTYPE` and `MALLOCDEBUG`
- 32-bit applications should link with `-bmaxdata:0x80000000`
- It is recommended to run with
 

```
ulimit -d unlimited
ulimit -s unlimited
```
- We will look at a sample problem, starting with
 

```
export MALLOCTYPE=debug
export MALLOCDEBUG=align:4,catch_overflow,debug_range:8:0,\
postfree_checking
```
- The `catch_overflow` option identifies memory overwrites, overreads, duplicate frees and writes to freed memory, cases usually exiting with SIGSEGV
- Specifying `align:4` with `catch_overflow` allows overruns down to 4 bytes to be detected

## AIX debug malloc tool (2)

- The option `debug_range:8:0` specifies that only allocations of size  $[8, \infty)$  should be checked
- Finally, the `postfree_checking` is causing a segfault on any access (read) to freed memory
- **Postfree checking consumes large amounts of extra memory**
- Enclosed is a sample with both overwrite errors and postfree errors
- **You will need to comment out the overwrite to check the postfree access**

## AIX debug malloc tool (2)

```
$ mpxlrf -g -O2 -qtune=auto -qarch=auto -o memdebug memdebug.f
$ MP_HOSTFILE=$PWD/hostfile MP_PROCS=5 MALLOCCTYPE=debug \
MALLOCDEBUG=align:4,catch_overflow,debug_range:8:0,postfree_checking \
./memdebug
Debug malloc detected...
/etc/ksh_env[8]: 303956 Memory fault(coredump)
/etc/ksh_env[8]: 374178 Memory fault(coredump)
ERROR: 0031-250 task 2: Segmentation fault
ERROR: 0031-250 task 0: Terminated
ERROR: 0031-250 task 1: Terminated
ERROR: 0031-250 task 3: Terminated
ERROR: 0031-250 task 4: Terminated

$ memdebug$ dbx memdebug coredir.2/core
...

Segmentation fault in main at line 18 in file "memdebug.f"
    18          a(i) = sqrt(real(i,8))
      (dbx)
```

### memdebug.f, line 1-20

```
program main
use MPI
integer n, myrank, npes, ipid, getpid, ierr
real(kind=8) :: x
real(kind=8), allocatable :: a(:)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,npes,ierr)
ipid = getpid()
allocate(a(npes+myrank))
call MPI_BARRIER(MPI_COMM_WORLD, ierr)

! fill arbitrary data
n = npes + myrank
if (myrank == npes/2) n = n + 1 ! 8-byte overwrite
do i = 1, n
    a(i) = sqrt(real(i,8))
end do
x = a(1) + a(n-1)
```

### memdebug.f, line 21-32

```
call sleep_(mod(myrank+2,npes))
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
deallocate(a)

! postfree read
!if (myrank == 1) x = x + a(2)
call MPI_BARRIER(MPI_COMM_WORLD, ierr)

write(*,*) 'mem-debug: x = ', x
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
call MPI_FINALIZE(ierr)
end
```

# Profiling

Timing your code

Simple profiling

Xprofiler (NYI)

IPM (NYI)

► TOC

# Timing

- Often it is of interest to time one or a few sections of code
- The compilers provides low overhead wall clock built-ins for this purpose
- The Fortran `REAL*8` function `rtc()` returns the value of the processor clock in seconds
- In C, a utility function `read_wall_time(2)` is provided to read the time base register, and `time_base_to_time(2)` to convert into seconds and nanoseconds.
- The C compiler builtin `unsigned long __mftb(void);` might be used to read the ticks only (64-bit mode)

The functions above must be called at the beginning and the end – the wall time is the difference

# Simple profiling

- When optimizing your code, it is best to compile CPU-intensive routines only with aggressive optimization
- Aggressive optimization produces more code and increase the risk of introducing compiler bugs breaking the code or producing wrong results
- The Unix `gprof` utility is a simple application to find the routines using the most time by instrumenting a typical execution
- The procedure is:
  - 1 compile and link with `-pg`
  - 2 run your application
  - 3 `gprof executable gmon.out >gmon.txt`
- The file `gmon.out` now contains various statistics, including a flat profile of your program's CPU usage.

# Simple profiling (2)

Sample output by instrumenting `bzip2`

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
15.7	4.68	4.68	90316838	0.00	0.00	.mainGtU [15]
14.6	9.03	4.35				.__mcount [16]
11.8	12.56	3.53	89	39.66	39.66	.generateMTFValues [1
10.0	15.54	2.98	983074	0.00	0.01	.mainQSort3 [11]
8.0	17.92	2.38	3103482	0.00	0.00	.fallbackQSort3 [18]
7.5	20.16	2.24	14	160.00	360.00	.fallbackSort [14]
7.3	22.33	2.17	89	24.38	127.53	.mainSort [10]
6.1	24.16	1.83	24286	0.08	0.09	.copy_input_until_sto
5.8	25.88	1.72	89	19.33	28.24	.sendMTFValues [19]
5.0	27.36	1.48	3481525	0.00	0.00	.mainSimpleSort@AF10_
1.8	27.89	0.53	44674410	0.00	0.00	.bsW [21]
1.4	28.31	0.42	4976261	0.00	0.00	.fallbackSimpleSort [
1.4	28.72	0.41	7149921	0.00	0.00	.add_pair_to_block [2
1.0	29.01	0.29	29913	0.01	0.84	.handle_compress [6]

## Advanced I/O

GPFS and parallel I/O

MPI I/O

Fortran asynchronous I/O

POSIX C I/O

► TOC

## GPFS and parallel I/O

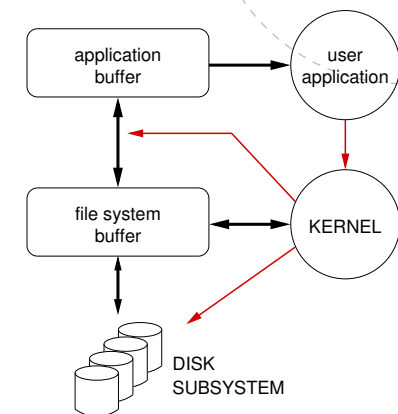
- An important feature of a HPC system is the capability of moving large amounts of data from or to memory, across nodes and from or to permanent storage
- In this respect a high quality and high performance global file system is essential
- GPFS is a robust **parallel** FS geared at high BW I/O, used extensively in HPC and in the database industry
- Disk access is  $\approx 1000$  times slower than memory access, hence key factor for performance are
  - spreading (striping) files across many disk units
  - using memory to cache files
  - hiding latencies in software

## GPFS and parallel I/O (2)

- High transfer rates is achieved by distributing files in blocks round robin across a large number of disk units, up to thousands of disks
- On njord, the GPFS *block size* and stripe unit is 1 MB
- In addition to multiple disks servicing file I/O, multiple threads might read, write or update (R+W) a file simultaneously
- GPFS use multiple I/O servers (4 dedicated nodes on njord), working in parallel for performance, maintaining file and file metadata consistency.
- High performance comes at a cost. Although GPFS can handle directories with millions of files, it is usually the best to use fewer and larger files, and to access files in larger chunks.

## File buffering

- The kernel does read-aheads and write-behinds of file blocks
- The kernel does heuristics on I/O to discover sequential and strided forward and backward reads.
- The disadvantage is memory copying of all data
- Can bypass with `DIRECT_IO` – can be useful with large I/O (MB-sized), utilizing application I/O patterns



## MPI I/O

- An efficient MPI-program often requires fine-grained distribution of data across processors for efficient and scalable execution
- As shown in the previous section, the file system layer works best with large and contiguous chunks of data to work on
- An efficient and highly parallel HPC application need to move data swiftly between the two worlds!
- A few “traditional” approaches
  - all tasks read the input, all tasks write output to individual files
  - all tasks read the input, data is shipped to task 0 for output
  - some of the tasks works as a I/O server subsystem, shipping data to and from the “compute” tasks (yes, reinventing the wheel . . .)

## MPI I/O (2)

- MPI-2 MPI-IO is designed to, as efficiently as possible, move data between disk and application memory
- MPI-IO utilize knowledge on application data structures and task topology to communicate I/O patterns to the (vendor) file system layer
- MPI-IO can do individual, collective, synchronous or asynchronous I/O from all or a subset of the tasks, with MPI datatypes.
- Functionality like Fortran direct access I/O or double buffering to overlap computations and I/O can be implemented quite straight forward

## Using MPI I/O

- All tasks must collectively open the file for I/O
- Next each task has to set its *file view*, the part of the file it is going to read or write.
- Then each task might do collective or individual I/O
- At the end, the file must be collectively closed
- It is possible to specify I/O usage hints when opening the file, to further optimize I/O. Available hints are vendor-specific, and are usually documented on [MPI\\_File\\_open\(3\)](#). See *Supported file hints*

## MPI I/O example

- Define a MPI vector type of integer elements, `MPI_INT`
- The vector element size is `INTS_PER_BLK = 16 int's`.
- A total of `FILESIZE / (sizeof(int) * INTS_PER_BLK)` vector elements in the file.
- A stride of `INTS_PER_BLK * numprocs` int's on file between a processor's consecutive vector elements.
- Each task performs a collective read, a *gather*, of its vector elements
- On completion, bytes on the file are stored *round-robin* across processors, in chunks of `INTS_PER_BLK` int's.

## MPI I/O example (2)

```
#include <stdlib.h>
#include <mpi.h>

#define FILESIZE      4096
#define INTS_PER_BLK  16

int main(int argc, char **argv)
{
    int *mydata, myrank, numprocs, numints;
    MPI_File fh;
    MPI_Datatype filetype;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

    mydata = (int *) malloc(FILESIZE/numprocs);
    numints = FILESIZE/(sizeof(int)*numprocs);
```

## MPI I/O example (3)

```
MPI_File_open(MPI_COMM_WORLD, "/work/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

MPI_Type_vector(numints/INTS_PER_BLK, INTS_PER_BLK,
               INTS_PER_BLK*numprocs, MPI_INT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh, sizeof(int)*INTS_PER_BLK*myrank,
                 MPI_INT, filetype, "native", MPI_INFO_NULL);

MPI_File_read_all(fh, mydata, numints, MPI_INT,
                 MPI_STATUS_IGNORE);

MPI_File_close(&fh);
MPI_Type_free(&filetype);
free(mydata);
MPI_Finalize();
return 0;
}
```

## Fortran asynchronous I/O

- IBM XL Fortran supports Fortran 2003 asynchronous I/O
- Provides the F2003 `ASYNCHRONOUS='yes'` and the IBM extension `ASYNCH='yes'` open specifiers
- A simple mechanism to overlap communications and computations
- May not be used with preconnected or multiply connected units
- Implemented by an `ID=intval` read/write specifier and a new `wait(intval)` statement

## Sample asynchronous I/O

```
real(kind=8), allocatable :: a(:, :)
...
open(10, status='old', access='direct', asynch='yes', recl=...

! read in the first columns
read(10, rec=1) a(1, columns_1)

do i = 2, ntiles
    read(10, id=idval, rec=i) a(1, columns_i)
    ! <compute on part i-1>

    wait(id=idval)
end do

! <compute on part ntiles>
```

## POSIX C I/O

- Using plain libc I/O (descriptor or file pointer) with non-overlapping data and appropriate GPFs buffer sizes might be as least as efficient as MPI I/O
- Might also do asynchronous I/O, with the C POSIX AIO API
- Using AIO requires including `<aio.h>`
- Note AIX provides legacy non-POSIX asynchronous I/O, POSIX I/O is the default with `<aio.h>`
- AIO use ordinary C `open(2)`, `aio_read(3)` or `aio_write(3)` to move data and `aio_fsync(3)` to sync outstanding requests
- AIO is efficient on AIX because it is frequently used with AIX database servers (Oracle)
- Refer to the appropriate man pages

## Efficient use of the queueing system

[Job class overview](#)

[LoadLeveler keywords](#)

[Consumable resources](#)

[LoadLeveler variables](#)

[Job report](#)

[Sample jobscript](#)

[▶ TOC](#)

## Job class overview

class	min-max nodes	max nodes / job	max runtime	description
forecast	1-180	160	unlimited	top priority class dedicated to forecast jobs
bigmem	1-6	4	7 days	high priority 115GB memory class
large	4-180	128	21 days	high priority class for jobs of 64 processors or more
normal	1-52	42	21 days	default class
express	1-186	4	1 hour	high priority class for debugging and test runs
small	1/2	1/2	14 days	low priority class for serial or small SMP jobs
optimist	1-186	48	unlimited	checkpoint-restart jobs

## LoadLeveler keywords

- `#@ job_name`  
the name of the job
- `#@ account_no`  
the account number to charge for the job
- `#@ job_type`  
specify `serial` for serial and SMP (OpenMP) programs, and `parallel` for MPI programs
- `#@ node_usage`  
specifies whether the job shares nodes with other jobs. Default set to `not_shared` except for the `small` class
- `#@ notification`  
when to notify user by mail. Default is mail sent when job ends

## LoadLeveler keywords (2)

```
#@ node
  the number of nodes requested by a job step
#@ tasks_per_node
  the number of tasks per node
#@ task_geometry
  allows you to group tasks (MPI ranks) of a parallel job step to run
  together on the same node, e.g.
  #@ task_geometry = {(0,1) (2) (3,4,5) (6)}
#@ output
  the name of the file to use as standard output (stdout)
#@ error
  the name of the file to use as standard error (stderr)
```

## LoadLeveler keywords (3)

```
#@ wall_clock_limit
  sets the time limit for the job
#@ resources
  specifies the resources consumed by each task of a job step
#@ requirements
  specifies requirements which a machine in the LoadLeveler cluster
  must meet to execute a job step, e.g.
  #@ requirements = (Feature == "SMT")
#@ queue
  marks the ending of the job step and places it in the queue
```

## Consumable resources

### Single-threaded programs

Use `ConsumableCpus(1)`. For the `ConsumableMemory` divide the available memory on a compute node between the number of tasks

```
#@ job_type = parallel
#@ node = 2
#@ tasks_per_node = 16
#@ resources = ConsumableCpus(1) ConsumableMemory(832mb)
#@ network.MPI = sn_all,,us
```

## Consumable resources (2)

### Shared memory (OpenMP) jobs

Set the `ConsumableCpus` to the number of threads (must be equal to the `OMP_NUM_THREADS`) and the `ConsumableMemory` to the available memory in the job class

```
#@ job_type = serial
#@ resources = ConsumableCpus(16) ConsumableMemory(13gb)
#@ environment = OMP_NUM_THREADS=16
```

## Consumable resources (3)

### Hybrid MPI/OpenMP job

Set the `ConsumableCpus` to the number of threads per MPI task and divide the available memory between the number of MPI tasks. E.g. running 8 MPI tasks, 2 tasks per node, with 8 threads each

```
#@ job_type = parallel
#@ node = 4
#@ tasks_per_node = 2
#@ resources = ConsumableCpus (8) ConsumableMemory (6656mb)
#@ environment = OMP_NUM_THREADS=8
```

## LoadLeveler variables

LoadLeveler has several variables that can be used in the keyword statements. This include the keywords themselves when defined.

```
#@ job_name = myjob
#@ error = $(job_name).$(jobid).err
#@ output = $(job_name).$(jobid).out
```

This will give standard error and outfiles:

```
myjob.411656.err
myjob.411656.out
```

## LoadLeveler variables (2)

LoadLeveler also define a set of environment variables that can be used in the command section of the script.

```
#@ job_name = myjob
...
# Create and move to my working directory
w=$WORKDIR/$USER/$LOADL_JOB_NAME.$LOADL_STEP_ID
mkdir -p $w
cd $w
$HOME/a.out
```

This will create and run the executable in the directory:

```
/work/vegarde/myjob.f05n02io.518795.0
```

## Job report

Adding the `-w` option to `llq` provides cpu and memory statistics for a job

```
$ llq -w f05n02io.407278.0
===== Job Step f05n02io.407278.0 =====
f06n05:
  Resource: CPU
            snapshot: 6
            total: 101514422
  Resource: Real Memory
            snapshot: 20
            high water: 1408112
  Resource: Virtual Memory
            snapshot: 5499
            high water: 1407909
  Resource: Large Page Memory
            snapshot: 0
```

Notice, the high water memory usage is reported in 4KB pages

## Job report (2)

Adding the `llq -w` at the end in a job script will append the cpu and memory statistics at job completion to the standard output file

```
...
# Run my job
$HOME/a.out
llq -w $LOADL_STEP_ID
exit 0
```

## Job report (3)

Loadleveler will by default send a mail notification when the job step completes. This will include

- start and exit time of the job
- the exit code of the job

If the available node memory is exceeded the job will be killed and the notification mail will include the following error message

```
LoadL_starter: WLM absolute real memory
                limit (13312 MB) exceeded
```

## Sample jobscript

```
#@ job_name = hybrid_job
#@ job_type = parallel
#@ node = 3
#@ tasks_per_node = 8
#@ class = normal
#@ ConsumableCpus(2) ConsumableMemory(1664mb)
#@ error = $(job_name).$(jobid).err
#@ output = $(job_name).$(jobid).out
#@ queue

export OMP_NUM_THREADS=2; export MP_LABELIO=yes

# Create (if necessary) and move to my working directory
w=$WORKDIR/$USER/test
if [ ! -d $w ]; then mkdir -p $w; fi
cd $w

$HOME/a.out

llq -w $LOADL_STEP_ID
```

## Sample jobscript (2), multiple job steps

```
#@ job_name = preprocess
#@ step_name = step0
#@ job_type = serial
#@ class = small
#@ executable = prepro
#@ ConsumableCpus(1) ConsumableMemory(500mb)
#@ queue

#@ job_name = work
#@ step_name = step1
#@ job_type = parallel
#@ node = 2
#@ tasks_per_node = 16
#@ class = normal
#@ executable = mpi_work
#@ ConsumableCpus(1) ConsumableMemory(832mb)
#@ dependency = (step0 == 0)
#@ queue
```

# Documentation

## Compiler PDF manuals

- [getstart.pdf](#) Setting up an configuring your environment, compiling, linking and troubleshooting
- [compiler.pdf](#) Compiler reference – options, pragmas, macros and (parallel processing) built-ins
- [langref.pdf](#) About the programming language support, standards compliance and extensions
- [proguid.pdf](#) Advanced programming topics, as porting, interlanguage calls, library development, optimization and parallelization

For more information see the 'Njord User Guide' at

<http://docs.notur.no/ntnu/njord-ibm-power-5>