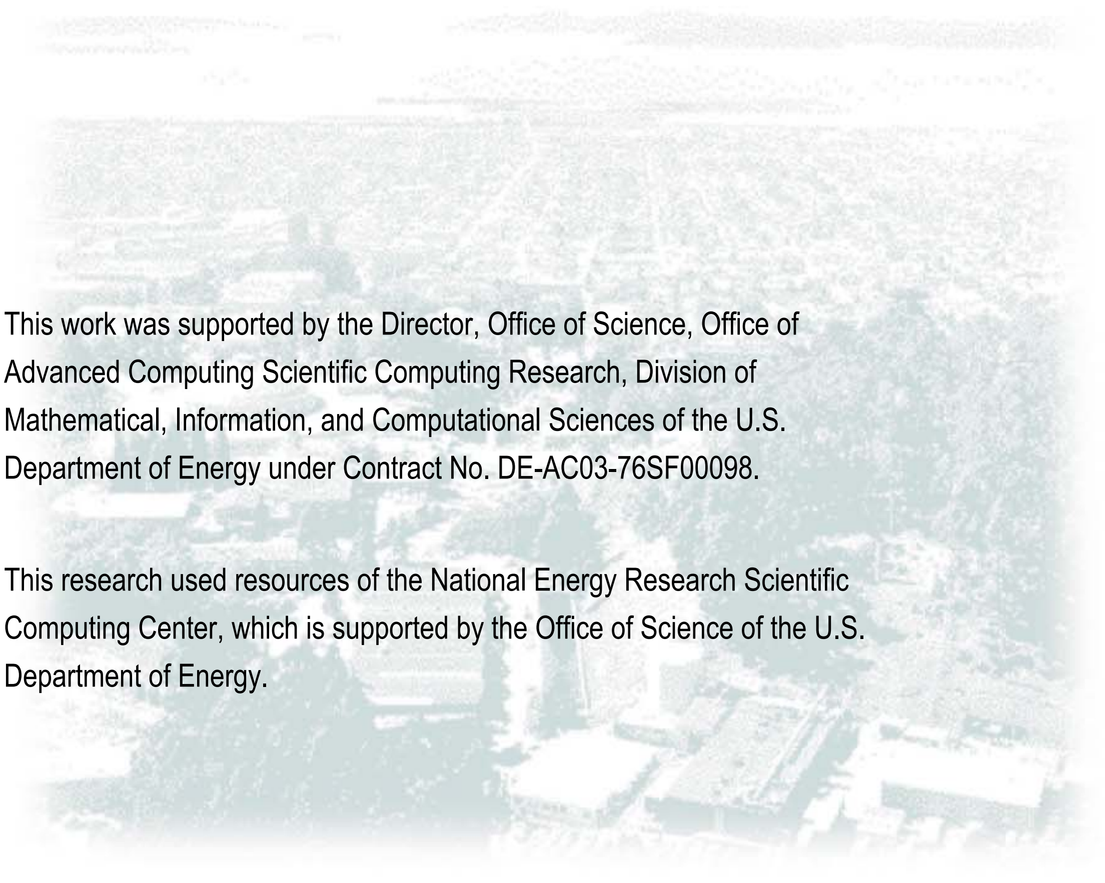
LBNL-XXXXX

ERNEST ORLANDO LAWRENCE
BERKELEY NATIONAL LABORATORY

Parallel Performance of the XL Fortran random_number Intrinsic Function on Seaborg

Richard A. Gerber
User Services Group, NERSC Division

July 2003



This work was supported by the Director, Office of Science, Office of Advanced Computing Scientific Computing Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

Parallel Performance of the XL Fortran `random_number` Intrinsic Function on Seaborg

Richard Gerber

Ernest Orlando Lawrence Berkeley National Laboratory, NERSC Division, User Services Group, One Cyclotron Road, MS: 943R0256, Berkeley, CA 94720

Abstract

The Fortran intrinsic function `random_number` is shown to perform very poorly when simultaneously called from 16 tasks per node on NERSC's IBM SP Seaborg in its default runtime configuration. Setting the runtime option `intrinthds=16` improves runtime performance significantly and gives good results for all possible numbers of tasks per node. It is speculated that the cause of the problem is the creation of an excessive number of threads in the default configuration. It is noted that these threads appear to be created by default, without specifying a "thread-safe" compiler or other user interaction.

Introduction

The poor performance of the BeamBeam3D code [1] used by the SciDAC[*] Accelerator project[†] on the current (July 2003) Seaborg system was caused by problems with the Fortran intrinsic function `random_number`. [2] Appropriate use of a runtime configuration parameter increased code performance significantly, decreasing the code's execution time by a factor of four. This report documents and quantifies the performance of `random_number` on Seaborg, a POWER 3 (375 MHz) IBM SP with 16 CPUs/node.

Most of the use of `random_number` by the NERSC user community is assumed to be in the context of a parallel code using MPI for communication. The `random_number` routine may constitute only a minor component of the algorithm, but may be called many times. In the BeamBeam3D code `random_number` was called for each of millions of particles each time step. For this investigation a small Fortran test code was prepared that calls `random_number` many times. In order to emulate a "typical" user runtime environment MPI was initialized (but otherwise unused).

The most common way to invoke the IBM XL Fortran compiler to create a parallel code is via the `mpxlf90` command; therefore, this command was used to compile source code cited in this report. IBM XL Fortran compiler version 8.1.0.3 was used. Note this is not one of the "thread-safe" IBM compilers that are used to create multithreaded programs.

RANDOM_NUMBER Configuration Options

There are no documented options to control the threading behavior of RANDOM_NUMBER in the XLF 8.1 for AIX manuals. However, the following excerpt is present in a README file in the XLF 8.1 installation directory on Seaborg and in the version 8.1 *XL Fortran for Linux on pSeries User's Guide*.

intrinthds={num_threads}

[*] Scientific Discovery through Advanced Computation, a program of the U.S. Department of Energy's Office of Science.
[†] Advanced Computing for 21st Century Accelerator Science and Technology.

Specifies the number of threads for parallel execution of the MATMUL and RANDOM_NUMBER intrinsic procedures. The default for *num_threads* equals the number of processors online.

As mentioned in the report[2] on the BeamBeam3D code, a web search located the following text at the URL `http://msgs.sp2.net/cgi-bin/get/sp0207/45.html`:

```
APAR: IY35729  COMPID: 5765F7100  REL: 810
ABSTRACT: ADD A RUNTIME OPTION TO CONTROL THE NUMBER OF THREADS

PROBLEM DESCRIPTION:
 ERROR DESCRIPTION:
 random_number() and matmul() are 2 Fortran instrinsic functions
 that are multi-threaded. It has been requested by
 customers to allow users to control the number of
 threads (eg. 1) used in these functions.

PROBLEM CONCLUSION:
A number of customers have experienced the problem that
multi-threaded intrinsic function floods their systems with
uncontrollable number of threads, which can result in
unacceptable run-time performance. In order to
solve this problem, we have to provide a new run-time option to
give them the ability to control the number of threads used in
those functions. Currently only matmul and random_number have
the multi-threaded version.
Syntax:
  intrinthds={num_threads}
```

This runtime option has a significant effect on the performance of `random_number` on NERSC's SP as is show below. A test code was written and used to investigate how execution times are affected by choices for the value of `intrinthds` and the number of MPI tasks used per node.

The Test Code

A test simple test code was created that does little but initialize MPI and then call RANDOM_NUMBER a number of times. Here is the code:

```
program ranno
        implicit none
        include 'mpif.h'

        integer:: i
        integer, parameter:: times=100
        integer, parameter:: idim=10000000
        real, dimension(idim):: realArray1,realArray2
        integer:: ierr
        integer:: totTasks, task_id


        call MPI_INIT(ierr)
        call MPI_COMM_SIZE( MPI_COMM_WORLD, totTasks, ierr)
        call MPI_COMM_RANK( MPI_COMM_WORLD, task_id, ierr)

        do i=1,times
                call RANDOM_NUMBER(realArray1)
                call RANDOM_NUMBER(realArray2)
        end do

        print  *,  "Task  ",task_id,"  of  ",totTasks,"  has  a  value:
",realArray1(100),realArray2(100)

        call MPI_FINALIZE(ierr)

end program ranno
```

The code initializes MPI so performance can be investigated as a function of the number of MPI tasks created on a node. It loops through enough calls to `random_number` to give a non-trivial, yet convenient run time. It then prints out a value in an attempt to keep the compiler from optimizing away the entire loop, which it might do if it recognized that the calculated values are never used or examined. With the array size given, the maximum memory usage on a node with 16 tasks is approximately:

```
10,000,000 entries * 4 bytes/entry * 2 arrays * 16 tasks = 1.28 GB.
```

This is less than the 16 GB of physical memory on the Seaborg node with the least amount of memory. Runtime performance would be severely impacted if the code's memory exceeded physical memory and the node starting memory paging to disk.

The code was compiled with XLF 8.1 and the value of `intrinthds` was set via the `XLFRTEOPTS` environment variable.

```
% mpxlf90 -O3 -qtune=pwr3 -qarch=pwr3 ranno.f
% setenv XLFRTEOPTS intrinthds=N
```

N was varied for each run.

Results

The code was run with 1, 2, 4, 8, 12, 13, 14, 15 and 16 MPI tasks on a single Seaborg node in order to investigate the effect of running various numbers of tasks on a node. Most NERSC user codes will use 16 MPI tasks/node and users are charged for using all 16 CPUs.

The relevant IBM software versions were:

```
  AIX 5.1
  ppe.poe                    3.2.0.13  poe Parallel Operating
  LoadL.full                 3.1.0.13  LoadLeveler
  bos.rte.bind_cmds          5.1.0.35  Binder and Loader Commands
  xlfcmp                      8.1.0.3  XL Fortran Compiler
  xlfrte                      8.1.0.3  XL Fortran Runtime
  xlfrte.aix51                8.1.0.3  XL Fortran Runtime Environment
```

Note that in the test code each task is essentially independent, connected only by MPI initialization. If there were no interaction between tasks, e.g. if each task was run on physically separate computer hardware, the run time should be independent of the number of tasks. However, this is not the case on SMP nodes like those on Seaborg; tasks must compete for access to memory for example.

For each run, the value of `intrinthds` was either unset (termed the "default" configuration) or had a value between 1 and 128. The NERSC `poe+` utility, which parses and concatenates output from IBM's `hpmcount` program, was used to record run times, `sys` times, and `user` times. Use of the `poe+` utility to measure timings for the entire code, as opposed to timing the `random_number` calls only, was appropriate in this case since MPI startup times are much shorter than the execution times reported here and thus had little relative effect. (MPI initialization times have been measured by NERSC and are on the order of .017 seconds/task; less than 1 second for 16 tasks or fewer.)

A striking result is that the program had a highly variable run time in the default configuration when 16 MPI tasks were run on a single node. Execution times varied unpredictably from 60 seconds to more than 30 minutes (jobs exceeding 30 minutes were killed). This result did not depend on the memory resident on a node, the variability was observed on both 16 and 32 GB nodes. Likewise the specific node used did not matter; the variation was observed by running the executable multiple times in succession in a single batch job running on one node.

The following table lists runs times for various combinations of total tasks and values of `intrinthds`. A dedicated compute node was used for all runs. When results were highly variable from run to run, multiple timings are show below. Where single timings are given, a variation of at most a few percent was observed for a minimum of three runs.

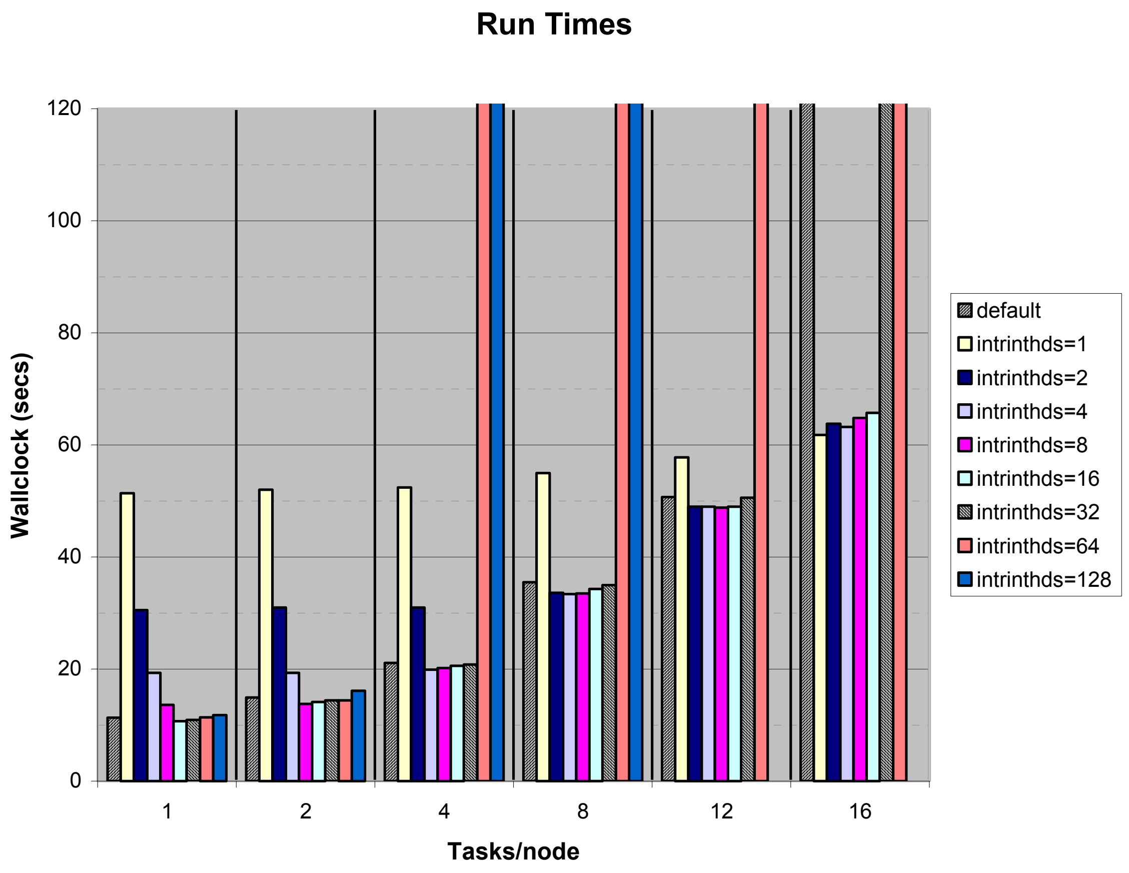
Run Times (seconds)

	Value of intrinthds								
Number of tasks	**default**	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
1	11.3	51.4	30.5	19.3	13.6	10.7	10.9	11.4	11.8
2	14.9	52.0	31.0	19.3	13.8	14.1	14.4	14.4	16.1
4	21.1	52.4	31.0	19.9	20.2	20.6	20.8	93.1, 167, 21.6(1)	>1400
8	35.5	55.0	33.6	33.4	33.5	34.3	35.0	>720, >1800, 583(1)	>1000
12	50.7	57.8	49.0	49.0	48.8	49.0	50.6	>1200	N/A(2)
13	145, 55.1(1)	58.5	52.7	52.1	52.1	52.7	54.5, 112, 54.1, 54.0(1)	>1200	N/A(2)
14	59.6, 641(1)	59.1	54.3	54.6	55.3	56.9	58.0, 207(1)	>1200	N/A(2)
15	62.8, 62.2, 698(1)	62.7	58.7	59.0	58.8	61.4	62.7, 456(1)	>1200	N/A(2)

16	1037, 233, 63.2, 66.0, 67.7, 857, 807.5, >1800, >720[1]	61.8	63.8	63.2	64.8	65.7	69.6, >1500, 66.8[1]	>1800, >1200	N/A[2]

[1] Where results are highly variable, multiple measurements are given.
[2] No runs were made with this combination of parameters.

These results are illustrated in the following graph. Configurations that had highly variable long run times are shown as extending off scale.



`Wallclock` time, time in `user` mode, time in `system` mode, and MFlips/s (Million Floating Point Instructions per Second) as reported by `poe+` for some runs in the default configuration are reported in the following table.

Default configuration measurements from poe+/hpmcount

Number of tasks	Wallclock seconds	user seconds	sys seconds	MFlips/s / task
1	11.3	46.0	17.1	1455
2	14.9	50.4	8.25	1107
4	21.1	52.1	4.65	708
8	35.5	52.3	3.69	463
16	varies; >63	varies; >53	varies; >2.8	Varies

Discussion

The code's performance in the default configuration with 16 tasks per node is clearly unacceptable for use in a production computing environment. The variation from run to run is great and in many cases the program is not observed to finish.

The second table above shows that in the default configuration 1 task accrues 46 seconds of `user` time, but only takes 11 seconds of `wallclock` time to run. This is an indication that the job is running multiple threads that execute on the other 15 CPUs on the node. The finding that multiple threads are being created by default, without using the "thread-safe" compilers or any other user specification, is notable and an unexpected result.

If one examines the timings when `intrinthds=32` in the first table and graph, they are remarkably similar to the timings obtained when `intrinthds` is unset (the "default configuration"). If we were to assume that in the default runtime environment 32 threads were created per task, it would explain the poor performance using 16 tasks/node; a combination that translates to 512 threads/node. From the timings given in this report, there appears to be a threshold somewhere around 512 (or perhaps as low as 256) spawned threads per node beyond which node performance becomes inconsistent and significantly degraded. The `uptime` command was observed to report a system load average of `99.99` during one of the long-running jobs. `LoadLeveler` (the IBM batch software) reported load averages up to `471` on the nodes with jobs that were measured to have long run times. These large values of system load indicate a heavily overburdened system.

Interestingly, a value of `intrinthds=16`, which would appear to be the intended default from a reading of the IBM comments on the web site referenced above, gives very good performance at all possible tasks per node on Seaborg.

These results indicate that heavy use of the `random_number` Fortran intrinsic can produce very poor performance in Seaborg's default configuration. While `random_number` is seldom expected to be the major component of a computational kernel, Seaborg users should be aware of potential problems with this intrinsic. Failure to do so could cause a minor component of the entire code to have a major impact on overall performance.

Although not designed to investigate how to perform the maximum number of `random_number` calls per unit time, the results of this report can be used to draw some tentative answers to that question. If one assumes that a single task will perform N times more `random_number` calls in N times the measured execution time of a single run described in this report, one can construct the following table:

Number of tasks per node	Time needed to execute RANDOM_NUMBER calls relative to the number performed with 16 tasks/node with intrinthds=16
1	2.61
2	1.72

4	1.28
8	1.04
12	1.03
16	1.00

Using 16 tasks/node with `intrinthds=16` performs the most calculations in the least amount of time.

Conclusions

Calling the Fortran `random_number` intrinsic function simultaneously from 16 MPI tasks on a single Seaborg SMP node in the default configuration gives unacceptable and unpredictable runtime performance. NERSC users should be advised that heavy use of `random_number` could adversely affect overall code performance. Setting the runtime parameter `intrinthds=16` gives good performance and should be recommended to NERSC users.

References

---

[1] J. Qiang, M.A. Furman, & R.D. Ryne, (2003), "*Parallel Particle-In-Cell Simulation of Colliding Beams in High Energy Accelerators*," submitted to Supercomputing 2003.

[2] Gerber, Richard A., (2003), "*An Investigation of Reported Anomalies When Running the BeamBeam3D Code on Seaborg*," http://www.nersc.gov/projects/scaling/beambeam3d.html.