KNL Performance Comparison: R and SPRINT

Adrian Jackson
a.jackson@epcc.ed.ac.uk

March 2017

1. Introduction

R (https://www.r-project.org/) is a statistical programming language widely used for data analytics and statistics. It is an interpreted language, like Python, so consists of a programming language and an interpreter that runs users' programs.

R has a number of different mechanisms for shared memory parallelism (parallelizing a program within a single ARCHER node), generally based around loop/thread parallelism. There are also distributed memory parallel packages for R. For this benchmarking we investigated the performance of the foreach shared memory parallel functionality (which distributes loop iterations to different threads) and the SPRINT distributed memory parallel package (http://www.r-sprint.org/).

2. Compilation, Setup and Input

Compilation

There are a number of pre-requisites required to build R on a system like ARCHER. We started by installing:

- bzip2
- xz
- curl
- pcre

When building R it is important to configure the maths libraries it uses correctly to give good performance. There are two options on ARCHER, Cray's libsci or Intel's MKL. For these benchmarks we used Intel's MKL and configured the build to use these like this (note, the same configuration can be used for the KNL and IvyBridge nodes on ARCHER):

```
export LIBSCI="-L${MKLROOT}/lib/intel64 -lmkl_intel_lp64 -
lmkl_sequential -lmkl_core -lmkl_blacs_intelmpi_lp64 -lpthread -lm -
ldl"
export LAPACK_LIBS="$LIBSCI"
export BLAS_LIBS="$LIBSCI"
export CRAYPE_LINK_TYPE=dynamic
```

On the main ARCHER system we then configured ARCHER using the following setup:

```
./configure --prefix=/work/d67/d67/adrianj7/R-mkl --with-blas --with-
lapack -with-x=no CC=cc FC=ftn F77=ifort CXX=CC CFLAGS="-
I/work/d67/d67/adrianj7/R-mkl/bzip2/include -
I/work/d67/d67/adrianj7/R-mkl/xz/include -I/work/d67/d67/adrianj7/R-
mkl/pcre/8.35/include -I/work/d67/d67/adrianj7/R-
mkl/curl/7.53.0/include -I/work/d67/d67/adrianj7/R-
mkl/readline/6.3/include -
I/opt/intel/composer_xe_2015.2.164/mkl/include" CPPFLAGS="-
I/work/d67/d67/adrianj7/R-mkl/bzip2/include -
I/work/d67/d67/adrianj7/R-mkl/xz/include -I/work/d67/d67/adrianj7/R-
mkl/pcre/8.35/include -I/work/d67/d67/adrianj7/R-
mkl/curl/7.53.0/include -I/work/d67/d67/adrianj7/R-
mkl/readline/6.3/include -
I/opt/intel/composer_xe_2015.2.164/mkl/include" LDFLAGS="-
L/work/d67/d67/adrianj7/R-mkl/bzip2/lib -L/work/d67/d67/adrianj7/R-
mkl/xz/lib -L/work/d67/d67/adrianj7/R-mkl/pcre/8.35/lib -
L/work/d67/d67/adrianj7/R-mkl/curl/7.53.0/lib -
L/work/d67/d67/adrianj7/R-mkl/readline/6.3/lib"
```

On the ARCHER KNL system we configured R as follows:

```
./configure --prefix=/work/knl-users/adrianj/R --with-blas --with-
lapack -with-x=no CC=cc FC=ftn F77=ftn  CXX=CC CFLAGS="-I/work/knl-
users/adrianj/R/xz/include -I/work/knl-
users/adrianj/R/pcre/8.35/include -I/work/knl-
users/adrianj/R/curl/7.53.0/include -I/work/knl-
users/adrianj/R/readline/6.3/include -
I/opt/cray/pe/libsci/16.09.1/GNU/5.1/mic_knl" CPPFLAGS="-I/work/knl-
users/adrianj/R/xz/include -I/work/knl-
users/adrianj/R/pcre/8.35/include -I/work/knl-
users/adrianj/R/curl/7.53.0/include -I/work/knl-
users/adrianj/R/readline/6.3/include -
I/opt/cray/pe/libsci/16.09.1/GNU/5.1/mic_knl" LDFLAGS="-L/work/knl-
users/adrianj/R/bzip2/lib -L/work/knl-users/adrianj/R/xz/lib -
L/work/knl-users/adrianj/R/pcre/8.35/lib -L/work/knl-
users/adrianj/R/curl/7.53.0/lib -L/work/knl-
users/adrianj/R/readline/6.3/lib"
```

We then ran make and built R.

As R is an interpreted language, with a full runtime system, once the runtime system has been built the additional packages we require need to be installed in it. This involves running the R runtime system (`R`) and running the following commands.

```
options(download.file.method='wget')
install.packages('rlecuyer')
install.packages('boot')
install.packages('e1071')
install.packages('rbenchmark')
install.packages('randomForest')
install.packages('doMC')
options(useHTTPS=FALSE)
source("https://bioconductor.org/biocLite.R")
biocLite("ShortRead")
install.packages('sprint', dependencies=TRUE,
repos='http://cran.rstudio.com/', INSTALL_opts ='--no-test-load')
```

For the ARCHER KNL system the build of R failed because part of the build process involves running the programs that have been compiled to install additional packages. However, as the programs have been compiled to run on the KNL hardware they fail on the KNL login nodes because they don't have the same instruction set as the KNL processors themselves.

This meant we had to build R on the ARCHER KNL nodes without the `craype-mic-knl` module loaded, and therefore without the correct KNL instruction set selected. We compiled with the `craype-ivybridge` module loaded. This means that the R executable was not built for the vector instructions on the KNL processors.

We recognized that this might create performance issues for R on the KNL system so we also built R on a standalone KNL system, available at EPCC and called ninja, and copied that version of R to the ARCHER KNL system to allow a comparison between the version built without the KNL instruction set and the one built natively on the KNL

Setup

We ran on a number of standard ARCHER IvyBridge nodes, and the quad flat (aoe:quad_0) KNL nodes. We did some initial testing of MCDRAM performance for the benchmark we are using in the evaluation and it had no performance impact so we have not used MCDRAM for our tests.

We used up to 64 cores per KNL node, with no hyperthreading on either system. Hyperthreading was investigated and did not improve performance so is not presented here.

The performance data we present is the fastest version of two independent runs for each benchmark. There was no significant variation between the runtime of the two different runs so error bars outlining the variability of performance for each data point are not included.

Input

For the test we are using the randomForest functionality in R, with a test cases representing 100000 genes and 62 cases. We implemented versions of this script using both the SPRINT functionality and the `foreach` functionality available in R. The main functionality from these input scripts are provided in Appendix A.

For these benchmarks we have skipped the randomForest tuning step and hard coded a number of trees and the number of variables available for splitting at each tree node (`mtry`). This enables us to quickly evaluate the randomForest performance using a consistent setup.

3. Performance Data

Figure 1 shows the core for core performance of R when using SPRINT or `foreach` on KNL and IvyBridge processors. We are comparing a number of different variations in this graph:

- R parallelisation strategies: `foreach` vs SPRINT
- Compilers on the main ARCHER system: Intel vs GNU
- Version compiled directly on KNL (ninja) vs compiled on the ARCHER login nodes
- Processors: IvyBridge vs KNL

Note, where a compiler is not mentioned, the Intel compiler was used.

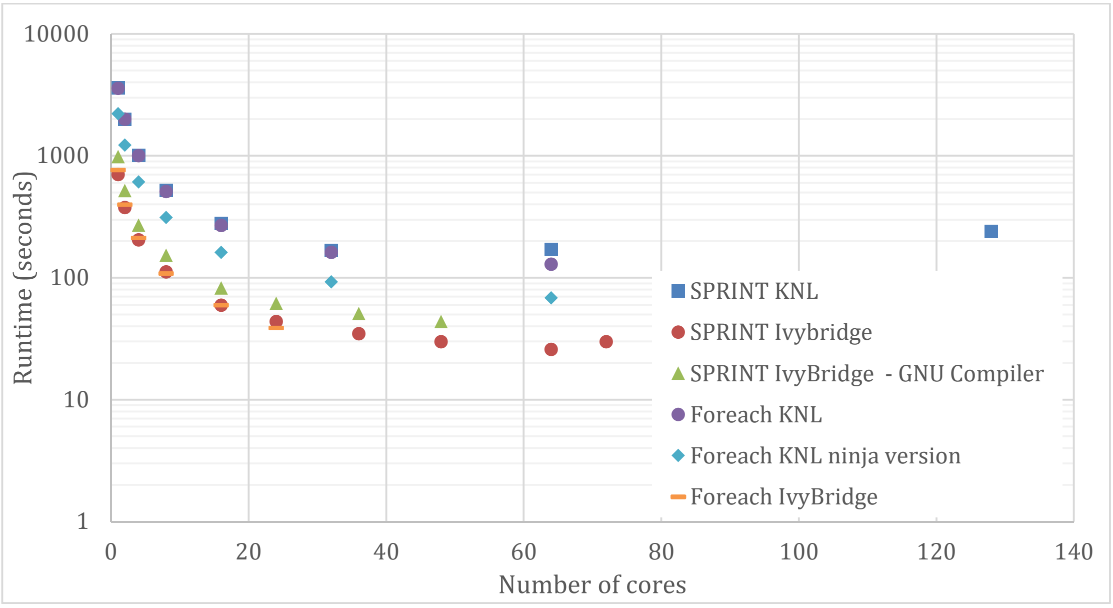

Figure 1: R parallel performance (SPRINT and foreach) on IvyBridge and KNL processors

We can see on a core for core comparison both the SPRINT and `foreach` implementations are slower on KNL than they are on IvyBridge. We can see that `foreach` and SPRINT on KNL offer similar performance, although at higher core counts `foreach` is a little faster (172 seconds vs 130 seconds at 64 cores on the KNL).

It is also evident from the graph that the version of R compiled on our native KNL system (ninja), where we can compile directly on the KNL processor and therefore use the correct KNL compiler flags, significantly outperforms the version we compiled on ARCHER. We can see targeting the correct instruction set for the KNL is important in terms of performance. Indeed at 64 cores the ninja version is nearly 2x faster than the ARCHER KNL version (see the “Foreach KNL” points vs the “Foreach KNL ninja version” points in Figure 1).

This highlights the importance of correct compilation on a system like the ARCHER KNL testbed, and identifies a problem that we need to solve for the ARCHER system, either by enabling compilation directly on the KNL nodes, or by providing some KNL based login nodes for compilation.

We could not do a similar performance comparison for SPRINT as SPRINT requires MPI libraries, and ninja does not have the Cray MPI libraries used on the ARCHER KNL system. However, a direct comparison of `foreach` vs SPRINT on our standalone system (ninja) suggests that the same performance impact is being experienced by the SPRINT version of the randomForest benchmark on the ARCHER KNL system.

We can also see, from Figure 1, that the Intel compilers provides better performance on IvyBridge processors than the GNU compilers.

Figure 2 presents a node for node comparison, rather than a core for core comparison, of the SPRINT performance on the ARCHER and ARCHER KNL systems. As we have multiple data points per node (i.e. 1, 2, 4, 8, 16, 24 cores are all 1 node) we can see a range of performances for some of the node numbers.

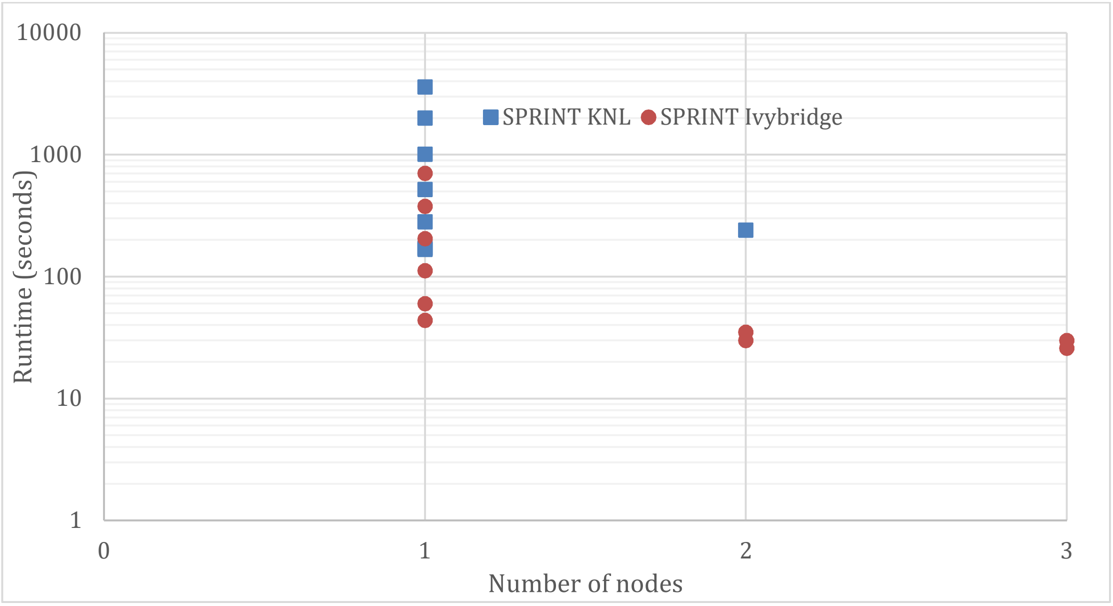

Figure 2: Runtime of SPRINT on IvyBridge and KNL processors vs number of nodes used

We can see from Figure 2 that the KNL system is not providing as good performance, on a node for node basis, as the IvyBridge system, and it is not scaling either. However, the scaling issues are likely to be because this test case isn’t large enough to require all the processes we are using (128 MPI processes for the 2 node KNL data point). Indeed, the IvyBridge performance is not scaling well when going from 1 to 2 to 3 nodes (3 nodes represent 72 MPI processes).

It is also likely that the poor KNL performance is, at least partly, due to the compilation issue we observed with the `foreach` case (i.e. ARCHER build vs ninja build).

Finally, Figures 3 and 4 demonstrate performance improvements that can be achieved if underpopulating on KNL processors by making sure threads or processes are properly distributed across the processor rather than being clustered on the first N cores.

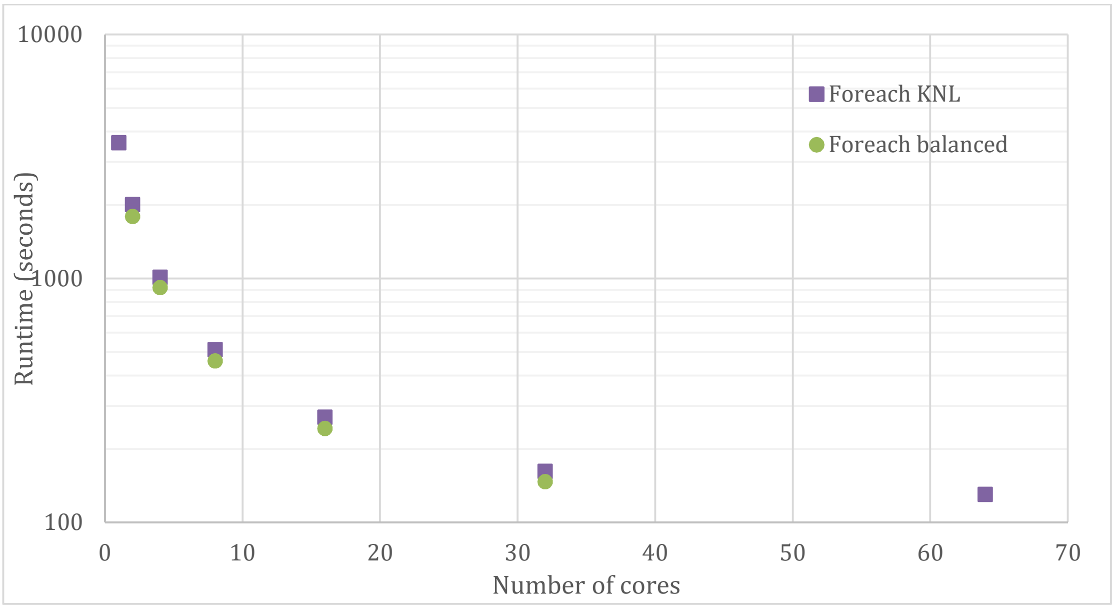

Figure 3: Comparing default KNL performance to the performance when foreach threads are manually distributed across the cores as evenly as possible

For the `foreach` case, where R is creating a set of threads to do the parallel work, we can use the `aprun` flag `-cc` to ensure threads are distributed across cores, i.e. if we are running 16 threads, we can distribute them across the 64 cores like this:

```
aprun -n 16 -cc 0,4,8,12,16,20,24,28,32,36,40,44,48,52,56,60 ./R
```

Such an approach gives between a 10-20% performance improvement over the default thread mapping `aprun` will perform. Note, if you were doing this on a different system, not a Cray, you could use the `taskset` utility to achieve the same benefits.

With the SPRINT version, where we have MPI processes rather than threads, we can use the `aprun` depth flag (`-d`) to specify how many cores to allocate to each MPI process. For instance, we could use this `aprun` line to run 16 SPRINT processes on the 64 cores of the KNL processor:

```
      aprun -n 16 -d 4 ./R
```

As shown in Figure 4, this approach gives similar performance benefits (10-20%) for the SPRINT version. Obviously, such an approach is not applicable if you are using all 64 cores on the KNL processor, and it also does not give any benefit in our cases when using less than 4 cores (as we aren't able to saturate the hardware of a single quadrant with such a small number of processes or threads).

The final data point on the balanced line in Figure 4 is where we are underpopulating on 2 nodes, running 64 MPI processes, 32 on each node, with a depth of 2. This does provide a significant speedup for the 2 node case, although it still isn't quick enough to make it worth using the resources of 2 nodes for the simulation.

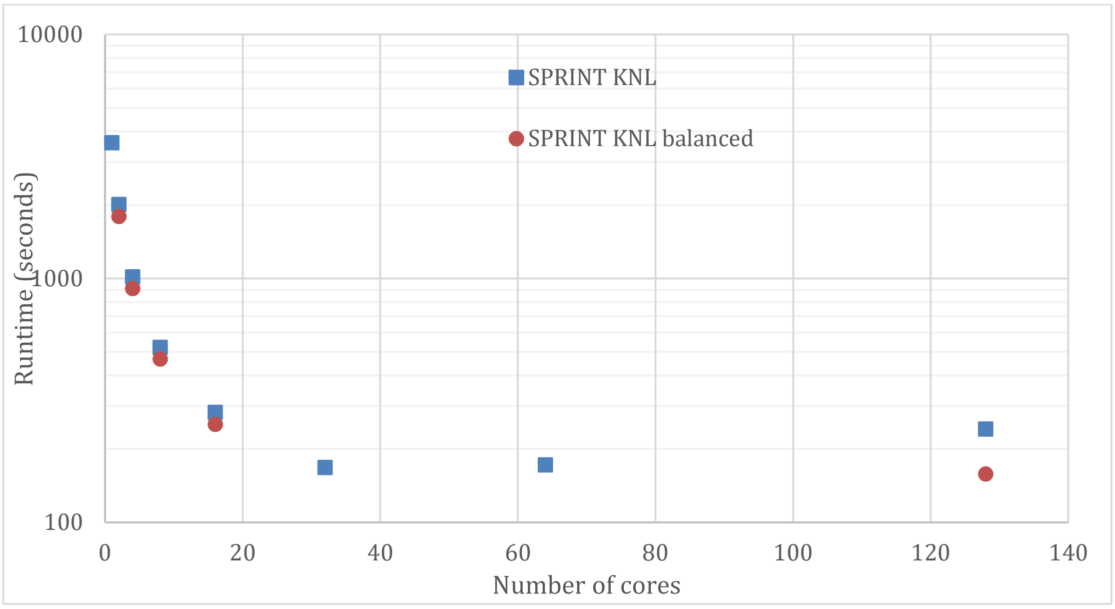

Figure 4: Comparing default KNL SPRINT performance to when MPI processes are distributed as evenly as possible across cores

4. Summary and Conclusions

We have shown that R can be parallelised, and scale well with both shared memory (`foreach`) and distributed memory (SPRINT) approaches on ARCHER. We have identified that compilation on KNL needs careful attention and configuration to get the best performance on the KNL processor, and some method of compiling applications that require it to be carried out directly on the KNL would be useful.

We have also shown that process and thread placement can impact performance if you aren't filling up nodes (if you're underpopulating). Currently R on KNL processors, at least for the randomForest benchmark we used in this benchmarking, is slower than on IvyBridge processors.

5. Acknowledgements

This work was supported by Intel through EPCC's IPCC project (Intel Parallel Computing Centre). Fiona Reid and Emmanouil Farsarakis from EPCC kindly reviewed and improved the report.

Appendix A

The SPRINT random forest script:

```
best.mtry <- 477
ntrees = 8192

transformedInputData = t(inputdata)

before_bench <- Sys.time()
rf.train <- prandomForest(x=transformedInputData,
                          y=sampleclasses,
                          ntree=ntrees,
                          importance=TRUE,
                          proximity=TRUE,
                          mtry=best.mtry)
```

```
after_bench <- Sys.time()

pterminate()
```

The foreach random forest script:

```
library(randomForest)

library(doMC)
registerDoMC(noOfThreads)

best.mtry <- 477
ntrees = 8192

rf.train <- foreach(ntreeloc=ntrees, .combine=combine,
.multicombine=TRUE,.packages='randomForest') %dopar% {
randomForest(x=transformedInputData,
                              y=sampleclasses,
                              ntree=ntreeloc,
                              importance=TRUE,
                              proximity=TRUE,
                              mtry=best.mtry)
}
```

Appendix B

Timings for the graphs in this report. Figures 1 and 2 are based on this data.
ARCHER IvyBridge Timings

		SPRINT		foreach
Number of Nodes Used	Number of Cores Used	GNU Compiler	Intel Compiler	Intel Compiler
1	1	991	705	769
1	2	523	379	401
1	4	272	205	213
1	8	154	112	109
1	16	83	60	60
1	24	62	44	39
2	36	51	35	
2	48	44	30	
3	64		26	
3	72		30	

ARCHER KNL Timings

		SPRINT	foreach	
Number of Nodes Used	Number of Cores Used	Intel Compiler	Intel Compiler	Ninja version
1	1	3606	3598	2228
1	2	2012	2011	1229
1	4	1015	1012	615
1	8	521	511	314
1	16	282	270	162

1	32	168	162	93
1	64	172	130	69
2	128	241		

Figures 3 and 4 are based on this data from the ARCHER KNL system:

		SPRINT		foreach	
Number of Nodes Used	Number of Cores Used	Normal distribution	Balanced distribution	Normal distribution	Balanced distribution
1	1	3606		3598	
1	2	2012	1799	2011	1799
1	4	1015	911	1012	917
1	8	521	468	511	460
1	16	282	252	270	243
1	32	168		162	147
1	64	172		130	
2	128	241	158		