# Best Practice Guide - IBM Power 775

Maciej Cytowski, ICM, University of Warsaw

28-11-2013


# Table of Contents

# 1. Introduction

The IBM Power 775 (IH) supercomputing server is a highly scalable system with extreme parallel processing performance and dense, modular packaging. It is based on IBM Power7 architecture and was designed by IBM within the US DARPA's HPCS (High Productivity Computing Systems) Program [1]. This unique supercomputing environment is currently available in a couple of organisations worldwide, e.g.: ICM, University of Warsaw (Poland), Met Office (UK), ECMWF (UK) and the Slovak Academy of Sciences (Slovak Republic). The “Boreasz” system available at ICM, University of Warsaw is a single cabinet system with 2560 IBM Power7 compute cores and a peak performance of 78 TFlop/s. The main purpose of the system is research carried out within the POWIEW project which among others includes scientific areas like numerical weather forecasting and large-scale cosmological simulations. Boreasz is also a Tier-1 system in the PRACE project infrastructure.

Boreasz was installed within the "HPC Infrastructure for Grand Challenges of Science and Engineering" (POWIEW) Project [http://www.wielkiewyzwania.pl], co-financed by the European Regional Development Fund under the Innovative Economy Operational Programme.

**Figure 1. IBM Power 775 system Boreasz available at ICM, University of Warsaw**


This best practice guide provides information about IBM Power 775 architecture machines in order to enable users of these systems to achieve good performance of their applications. The guide covers a wide range of topics, from the detailed description of the hardware, through information about the basic production environment, to information about porting and submitting jobs and tools and strategies for analysing and improving the performance of applications.

This document is a specific best practice guide based on previously published *"Best Practice Guide - IBM Power"* written by J. Engelberts (SURFsara), W. Lioen (SURFsara), H. Stoffers (SURFsara), M. Cestari (CINECA), S.

[1] DARPA High Productivity Computing Systems (HPCS) Program, http://www.darpa.mil/Our_Work/MTO

Giuliani (CINECA), E. Rossi (CINECA), J. Rodriguez (BSC), D. Vicente (BSC) and G. Collet (CEA). The original documents are available online at the PRACE website [http://www.prace-ri.eu].

# 2. System Architecture and Configuration

## 2.1. Processor architecture

The Power7 processors of the Boreasz system have 8 cores each running at a clock speed of 3.83 GHz inside a water-cooled Power 775 cabinet structure. Each core has 64 KB L1 cache consisting of 32 KB instruction cache and 32 KB data cache. Every core has private, very fast 256 KB L2 cache called "L2 Turbo cache". Eight cores on a single processor share a L3 cache of 32 MB. Each core has its own private 4 MB L3 cache, but can also access other L3 sections if needed. Such an L3 cache architecture enables a subset of cores to utilize the entire large shared L3 cache when the remaining cores are not using it. Furthermore, each core has four floating point units. Each floating point unit is able to create a 64-bit fused multiply-add result per clock cycle. This leads to 8 flops per cycle per core. With this number the theoretical performance of Boreasz comes to 8 flops/core x 3.83 Ghz x 32 cores/node x 80 nodes = 78.43 Teraflops.

Power7 processors support Simultaneous Multithreading (SMT) [http://www-03.ibm.com/systems/resources/pwrsysperf_SMT4OnP7.pdf]. SMT is a processor technology that allows up to four separate instruction streams (threads) to run concurrently on the same physical processor, improving overall throughput. To the operating system, each hardware thread is treated as an independent logical processor. This means that, in that case, there are 32 logical cores on one processor.

### Operating system

The operating system of Boreasz is AIX. There are some AIX specific issues that might be confusing for Linux users (e.g. different syntax of some of the standard Unix commands) which are described later in the document. Also, since the instruction set of the Power7 architecture is different than that of standard x86 architecture, binaries can not just be copied from PC or x86 cluster system and run on Power7 machine.

### Endianness

Another typical difference with regular PCs, which sometimes leads to confusion, is the endianness of the Power7 processor and PowerPC architecture in general. In contrast with Intel and AMD processors as found in most PCs, the Power7 is big endian.

Most numbers in computer memory (or registers on the CPU), consist of multiple bytes. On big endian machines the first byte in memory is the most significant one (MSB). Consecutive bytes are less significant. This is also referred to as MSB-LSB. On little endians, the order is reversed, i.e. LSB-MSB.

As an example, given number N = 4356732 (or N = 427A7C in hexadecimal representation) will be stored as:

7C,7A,42,00,00,00,00,00

on little endian systems. On big endian systems the same number is stored in a reversed fashion:

00,00,00,00,00,42,7A,7C.

This means that special care has to be taken with the transfer of binary data files from one type of machine to another. A typical case is when a user prepares (binary) input files on a local PC, which are transferred to the supercomputer for a large-scale calculation or, the other way around, when a user transfers binary output files produced on a HPC platform to its local PC for post processing or visualization. Several tools exist to circumvent this problem. Firstly, users can use ASCII coded (or human readable) files. Secondly, there exist a number of I/O libraries, like NetCDF and HDF, which ensure portability of binary data across different systems and take care of endianness under the hood.

## 2.2. Building Blocks

Each node of Boreasz (IBM Power 775) consists of 4 Power7 processors, or 32 Power7 physical cores. There are eight nodes (called octants) within a single drawer. There are 10 such drawers in the system, which totals up to 80

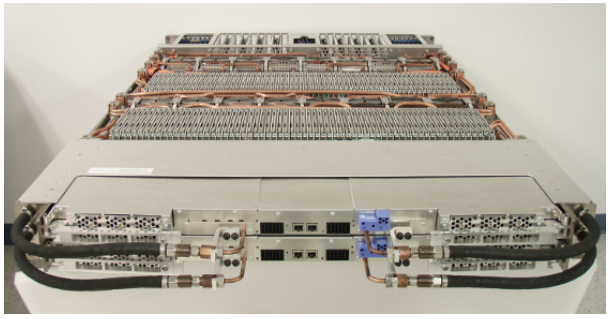
nodes, 320 Power7 processors and 2560 compute cores. Each drawer is equipped with a valve system to enable the use of deionized water for cooling the high frequency processors.

Please note that 4 of the Boreasz nodes are special purpose nodes (i.e. front-end, service and I/O nodes) thus only 76 nodes are available for computational jobs.

# 2.3. Memory Architecture

Nodes of Boreasz are equipped with 128 GB of memory (4 GB per physical core). It should be noted that only the 32 physical cores within each node have direct access to the memory. When more memory would be needed for a single task, programs should be able (or be enabled) to use some kind of communication mechanism, for example the common Message Passing Interface (MPI), or using sockets.

**Figure 2. IBM Power 775 drawer with cooling valves connected to each QCM node and each memory DIMM. The system is 100% water cooled.**



A single node is called an octant (there are eight nodes in one drawer) or a QCM - Quad Chip Module. A QCM contains 4 Power7 chips that are connected in a ceramic module, which gives a 32-way SMP node. The total memory bandwidth of a QCM is 512 GB/s.

# 2.4. Fat-nodes

Boreasz can be very useful for applications based on coarse-grain parallelism whose performance depend on high memory bandwidth. A single node has approximately 1 TFlop/s of compute performance, 128 GB of memory and up to 512 GB/s of memory bandwidth. This gives a very good 0.5 Byte/Flop ratio.

As an example of an application which performance depends on both the CPU performance and memory bandwidth we show the results of VASP test case which was executed on Boreasz and on a reference x86_64 cluster (Sun Constellation system based on AMD Opteron architecture). The test case was a simulation of a supercell crystal GaAs consisting of 80 atoms. In the table below we present walltime measurements on different partitions of two computational systems. We can see that VASP running on Boreasz is much more faster when core to core performance is compared, but also it presents better scalability. The best results obtained for Boreasz system (on 3 nodes) was more than 6 times better than the best result obtained on the x86_64 cluster.

**Table 1. Comparison of VASP test case on IBM Power 775 (Boreasz) and on x86_64 cluster (Halo2) at ICM**

| System | Number of cores | Partition size (TFlop/s) | Walltime |
|---|---|---|---|
| Sun Constellation, x86_64 | 16 (1 node) | 0.14 | 48 297 s |
| Sun Constellation, x86_64 | 32 (2 nodes) | 0.28 | 26 309 s |
| Sun Constellation, x86_64 | 48 (3 nodes) | 0.42 | 19 154 s |
| Sun Constellation, x86_64 | 64 (4 nodes) | 0.56 | 14 423 s |
| IBM Power 775 | 32 (1 node) | 0.98 | 4 800 s |
| IBM Power 775 | 64 (2 nodes) | 1.96 | 2 864 s |
| IBM Power 775 | 96 (3 nodes) | 2.94 | 2 475 s |

## 2.5. Networks and Node Interconnect

The network susbsystem of Boreasz is one of the most efficient solutions currently available on the market. It has a hierarchical structure. The first networking layer is the node interconnect between all 8 octants within each drawer. This layer is based on the so-called Hub Chip Module or Torrent (eight per drawer) which connects each QCM to PCI-e (two 16x and one 8x PCI-e slot). Each Hub is connected to every other Hub in the drawer thrue copper wiring (so-called L-links) which results in low latency and high bandwidth connectivity between eight QCM wiithin drawer. Each Hub is capable of transmitting 1.1 TB per second which includes both network connectivity with other nodes and I/O services. A key feature of the Hub chip is that it is integrated into the POWER7 coherency bus allowing it to participate in the coherency operations on the POWER7 chips. The Host Fabric Interconnect (HFI), which manages communication to and from the network, can extract data from the POWER7 memory or directly from the POWER7 caches. In addition the HFI can inject network data directly into the processor's L3 cache [2].

The second networking layer is the interconnect between each and every drawer in the system. It is based on the so-called Integrated Switch Router (ISR) which routes traffic between Hub chips on different drawers thrue the so-called D-links. There are no external switches or routers outside Power 775 system. When communicating between two drawers the worst case direct route utilizes up to three hops via L-D-L links.

## 2.6. I/O subsystem

Nodes of Boreasz system are diskless. There is one dedicated storage drawer within the rack which is connected with I/O nodes (file system node) of the system via SAS PCIe cards. Total capacity of the storage is arround 130 TB and the bandwidth for accessing data from computational nodes is approximately 10 GB/s. This whole storage space is made available on the frontend and all computational nodes based IBM's parallel I/O GPFS solution.

# 3. System Access

## Logging in

Boreasz is running the AIX operating system. Command line access to such systems can be obtained with SSH. Once users have received a username and a password, they will be able to login.

In order to login to Boreasz (assuming that user already received his/her username and password) user needs to perform following step:

1. Login to Boreasz system: *ssh username@boreasz.hpc.icm.edu.pl*

Currently, three operating systems are popular amongst users, Windows, Mac OS X and Linux. To start with the latter, the SSH client is already installed with a basic installation of the OS. A second advantage is that most of these installations run the X window system. This comes in handy whenever a graphical user interface is required. Examples of such applications are graphical debuggers.

Some 10 years ago, Apple decided to redesign its Mac OS to an own NeXTSTEP/BSD flavor called Darwin. Like with Linux, an SSH client is already installed and X windows is available and easily installed.

For Windows users, some more preparations are necessary. By default, Windows does not come with an SSH client, nor with an X windows server. Several freeware and commercial packages are available and can be found on the web. A commonly used terminal emulator capable of running ssh that can be downloaded from the web is PuTTY (free). Free X windows servers for Windows are e.g. Xming and Xorg from Cygwin suite.

## File transfer

Like login in over a secure socket layer (SSL), also files are transferred over that layer. In Linux and OS X, files can be transferred using SCP. Again, on Windows there is no default SCP client. “SSH Tectia” comes with a file transfer feature, another free tool is WinSCP.

[2] J.Abeles et al. "Performance Guide for HPC Applications on IBM Power 775 Systems"

## PRACE DECI access (in preparation)

The Boreasz system is part of the PRACE DECI infrastructure. This means that in the future logging in can also be done using the GSI-SSH protocol. To make use of this, a user needs a Grid certificate as means of authentication, rather than a username and a password. Also in the future file transfer will be possible using GSIFTP or GridFTP.

# 4. Production Environment

This section describes specific rules and setup with respect to ICM's accounts and users.

## 4.1. Access

Access to ICM's HPC platforms is allowed to owners of an ICM account (i.e. users). Each user is uniquely identified by a username/password pair. Login credentials are strictly personal, meaning that no sharing even between members of the same working group is allowed. Each single user is considered to be personally responsible for any misuse that takes place. Moreover, it is forbidden to have or to use multiple ICM's accounts.

Personal accounts are bound to specific projects or computational grants that the user is involved in. It is allowed for a single user to be involved in more than one computational grant on ICM's HPC systems. Each of these computational grants has a unique group identifier (which is also used for accounting, see below). It is the responsibility of each user to keep track of the list of computational grants that he or she is involved in. Moreover it should be reported immediatly to ICM's help desk (*pomoc@icm.edu.pl*) when a specific user is no longer involved in a given project. This should be done by the Principal Inverstigator of the computational grant.

## 4.2. Accounting

As described above, each users can be assigned to (at least) one computational grant. In order to access the computational resources for computing it is mandatory to specify on which computational grant (project budget) the billing for this specific run is to be charged. This is done by specifying the so called "account_no", in a form that depends on the resource manager itself.

On Boreasz it is mandatory to add the following entry in the LoadLeveler submission script:

```
#@ account_no = <account_no>
```

with <account_no> being one of the computational grants the submitting user is authorized for. If this entry is not included in the batch submission script, the resource manager will reject the request. In order to retrieve the available computational grants that are assigned to a specific user use the groups command:

```
$ groups username
```

Please note that it is the responsibility of each user to assign specific computational jobs to different grants. ICM's accounting system is keeping track and gathers all usage information per user and per computational grant. This information is used during yearly reporting and evaluation of computational grants.

In order to check the usage of budget assigned to a specific project on Boreasz please send an email to: *powiew-admins@icm.edu.pl*.

# 5. Programming Environment / Basic Porting

Please note that TCSH is the default shell on the Boreasz system. All example job scripts and commands in this document are only valid in the TCSH shell.

The programming environment of the IBM Power 775 machine contains a choice of compilers for the main scientific languages (Fortran, C and C++), debuggers to help users in finding bugs and errors in the codes, profilers to help in code optimization and numerical libraries for getting performance in a simple way.

Since this is a massively parallel system, it is mandatory to interact with the parallel environment by adopting a parallel paradigm (MPI and/or OpenMP), using parallel compilers, linking parallel libraries, and so on.

User codes, once written, are usually executed within the batch environment, using the job language of the native scheduler.

# 5.1. Available compilers and usage recommendations

On the IBM Power 775 the native IBM compilers (XL) are available for Fortran, C and C++.

## Compiler invocation

```
C, C++ : xlc, xlC
         xlc_r, xlC_r (Thread safe)


Fortran: xlf, xlf90, xlf95, xlf2003
         xlf_r, xlf90_r, xlf95_r, xlf2003_r (Thread safe)
```

The "_r" invocations instruct the compiler to link and bind object files to thread-safe components and libraries, and produce thread-safe object code for compiler-created data and procedures. We recommend using the "_r" versions.

You can check the compiler version using the -qversion flag:

```
xlc -qversion
xlf -qversion
```

## Compiling parallel MPI programs (distributed memory parallelism)

All wrappers around the compilers start with "mp". On AIX the suffix "_r" can be added behind the wrapper to indicate thread-safe compilation.

```
C, C++ : mpcc, mpCC
         mpcc_r, mpCC_r (Thread safe)


Fortran: mpxlf, mpxlf90, mpxlf95
         mpxlf_r, mpxlf90_r, mpxlf95_r, mpxlf2003_r (Thread safe)
```

## Compiling parallel OpenMP programs (shared memory parallelism)

XL compilers support OpenMP, a shared-memory parallelism constructs. OpenMP provides an easy method for SMP-style parallelization of discrete, small sections of code, such as a "do loop". OpenMP can only be used among the processors of a single node. For use with production scale, multi-node codes, OpenMP threads must be combined with MPI processes.

To compile a C, C++ or Fortran code with OpenMP directives use the thread-safe (_r) version of the compiler and the "-qsmp=omp" directive. It should be noted that the -qsmp=omp option is required for both the compile step and the link step. XL C/C++ and Fortran support OpenMP V3.0.

Example of usage:

```
xlf_r -qsmp=omp -qnosave -o execname filename.f
xlf90_r -qsmp=omp -o exename filename.f
xlc_r -qsmp=omp -o exename filename.c
xlC_r -qsmp=omp -o exename filename.C
```

Please note that for the invocations xlf, f77, fort77 and most importantly xlf_r, all local variables are STATIC (also known as SAVE variables), and are therefore not thread-safe! Please add the flag `-qnosave` to generate thread-safe code with these invocations.

## Compiling parallel hybrid programs (MPI + OpenMP)

OpenMP and MPI can be freely mixed in your code. You must use a “multiprocessor” and “thread-safe” compiler invocation with the -qsmp=omp option, *i.e.*:

```
mpxlf_r -qsmp=omp -qnosave
mpxlf90_r -qsmp=omp
mpcc_r -qsmp=omp
mpCC_r -qsmp=omp
```

## Memory Usage Modes

The IBM XL compilers can support both 32 and 64-bit addressing through the -q32/-q64 options. Note that 32-bit mode is default on Boreasz. You can change the default mode of the compilers by setting an environment variable:

```
export OBJECT_MODE=64
export OBJECT_MODE=32
```

The addressing mode can also be specified directly to the XL compilers with the -q option:

```
xlf -q64
xlf -q32
```

Using the 64-bit option causes all pointers to be 64 bits in length and increases the length of “long” datatypes from 32 to 64 bits. It does not change the default size of any other datatype.

The following points should be kept in mind when choosing the memory usage mode:

- You cannot bind object files that were compiled in 32-bit mode with others compiled in 64-bit mode. You must recompile to ensure that all objects are in the same mode.
- Your link options must reflect the type of objects you are linking. If you compiled 64-bit objects, you must also link these objects with the -q64 option.

It is recommended that users compile in 64-bit mode unless they have compatibility issues within their code.

## Recommended Optimization Options

The recommended optimization options for the XL compilers are

```
-O3 -qstrict -qarch=pwr7 -qtune=pwr7
```

A more aggressive optimization is possible, for example using the following options:

```
-O3, -qhot, -qipa=level=2 ...
```

Instead of `-O3` higher levels `-O4` or `-O5` can be tried. But, in some cases, these may cause problems so it is advisable to start with a weaker optimization level. For the full range of compiler and linker options see the appropriate man page.

## Recommended Debugging/Profiling Option

Debugging requires the -g option and profiling additionally requires the -p or -pg (recommended) option (also at link time). In both cases it is recommended to use the -qfullpath option.

| | |
|---|---|
| -g | Generates debug information for debugging tools. |
| -p | Sets up the object files produced by the compiler for profiling. |
| -pg | like -p, but it produces more extensive statistics. |
| -qfullpath | Records the full or absolute path names of source and include files in object files compiled with debugging information (when you use the -g option). |

Here are some examples:

```
xlc_r -g -qfullpath -O0 hello.c
xlf_r -g -pg -qfullpath -O3 -qstrict -qarch=auto -qtune=auto hello.f90
```

## Further Documentation

A collection of IBM documentation files, including programming and reference guides:

- IBM XL C/C++ 11.1 [http://www-01.ibm.com/support/docview.wss?uid=swg27018970&aid=1]
- XL Fortran 14.1 [http://www-01.ibm.com/support/docview.wss?uid=swg27024803&aid=1]

# 5.2. Available (vendor optimized) numerical libraries

The vendor optimized numerical library for this platform are ESSL, PESSL and MASS: optimized numerical libraries from IBM. There is also a set of libraries compiled by ICM staff and made available on the system in */opt* directory.

## (P)ESSL: (Parallel) Engineering and Scientific Subroutine Library from IBM

Taken from the ESSL website [http://www.ibm.com/systems/p/software/essl]

ESSL is a collection of subroutines providing a wide range of performance-tuned mathematical functions for many common scientific and engineering applications. The mathematical subroutines are divided into nine computational areas:

- Linear Algebra Subprograms

- Matrix Operations
- Linear Algebraic Equations
- Eigensystem Analysis
- Fourier Transforms, Convolutions, Correlations and Related Computations
- Sorting and Searching
- Interpolation
- Numerical Quadrature
- Random Number Generation

ESSL provides two run-time libraries; both libraries support both 32-bit and 64-bit environment applications:

- The ESSL Serial Library provides thread-safe versions of the ESSL subroutines for use on all processors. You may use this library to develop your own multithreaded applications.
- The ESSL Symmetric Multi-Processing (SMP) Library provides thread-safe versions of the ESSL subroutines for use on all SMP processors. In addition, some of these subroutines are also multithreaded, meaning, they support the shared memory parallel processing programming model. You do not have to change your existing application programs that call ESSL to take advantage of the increased performance of using the SMP processors; you can simply re-link your existing programs.

Both libraries are designed to provide high levels of performance for numerically intensive computing jobs and both provide mathematically equivalent results. The ESSL subroutines can be called from application programs written in Fortran, C, and C++.

Parallel ESSL is a scalable mathematical subroutine library for standalone clusters or clusters of servers connected via a switch and running AIX and Linux. Parallel ESSL supports the Single Program Multiple Data (SPMD) programming model using the Message Passing Interface (MPI) library. The Parallel ESSL SMP libraries support parallel processing applications on clusters of Power Systems servers and blades connected via a LAN supporting IP or with an InfiniBand switch.

Parallel ESSL provides subroutines in the following computational areas:

- Level 2 Parallel Basic Linear Algebra Subprograms (PBLAS)
- Level 3 PBLAS
- Linear Algebraic Equations
- Eigensystem Analysis and Singular Value Analysis
- Fourier Transforms
- Random Number Generation

For communication, Parallel ESSL includes the Basic Linear Algebra Communications Subprograms (BLACS), which use MPI. For computations, Parallel ESSL uses the ESSL subroutines (ESSL is a pre-requisite).

The Parallel ESSL subroutines can be called from 32-bit and 64-bit application programs written in Fortran, C, and C++ running the AIX and Linux operating systems.

The Parallel ESSL SMP Libraries are provided for use with the IBM Parallel Environment MPI library. You can run single or multithreaded US or IP applications on all types of nodes. However, you cannot simultaneously call Parallel ESSL from multiple threads.

## Compiling and linking your code with the ESSL library

To compile your code (C or Fortran) with the ESSL library you only need to add the appropriate “-l” option:

```
xlf_r prog.f -lessl
```

If you are accessing ESSL from a Fortran program, you can compile and link using the commands shown in the table below.

| ESSL Library | Environment | Fortran Compile Command |
|---|---|---|
| Serial | 32-bit integer, 32-bit pointer | xlf_r -O -q32 -qnosave xyz.f -lessl |
| | 32-bit integer, 64-bit pointer | xlf_r -O -q64 -qnosave xyz.f -lessl |
| | 64-bit integer, 64-bit pointer | xlf_r -O -q64 -qnosave -qintsize=8 xyz.f -lessl6464 |
| SMP | 32-bit integer, 32-bit pointer | xlf_r -O -q32 -qnosave xyz.f -lesslsmp -lxlsmp |
| | 32-bit integer, 64-bit pointer | xlf_r -O -q64 -qnosave xyz.f -lesslsmp -lxlsmp |
| | 64-bit integer, 64-bit pointer | xlf_r -O -q64 -qnosave -qintsize=8 xyz.f -lesslsmp6464 -lxlsmp |

## Further Documentation

- IBM Reference manual for ESSL [http://www-05.ibm.com/e-business/linkweb/publications/servlet/pbi.wss?CTY=US&FNC=SRX&PBL=SA22-7904-05]

## Important notice

Please note that not all BLAS, LAPACK and FFTW functionality is available in (P)ESSL. Original BLAS, LAPACK and FFTW libraries are also available on the system in the /opt directory.

## Additional libraries available on Boreasz

Other numerical libraries can be available on the system; you can verify it by using the command:

```
module avail
```

At the time of writing of this documentation the list of additional libraries was the following:

- Linear algebra: BLAS, BLACS, LAPACK, SCALAPACK
- Solvers: PETSc, Hypre
- Fourier transform: FFTW 2, FFTW 3
- Multiprecision libraries: MPFR, GMP
- Domain decomposition: Metis, Parmetis, Zoltan
- Random Number Generators: SPRNG
- Others: GSL

To load a specific library (for example GSL - GNU Scientific Library) use the following command:

```
module load gsl
```

This sets three environment variables LIBNAME_DIR, LIBNAME_LIB and LIBNAME_INCLUDE. The first one is the base directory of the library. The second one is the path to library which can be used when linking by typing *-L$LIBNAME_LIB*. The third one is the path to library include files which can be used during the compilation as *-I$LIBNAME_INCLUDE*. Of course in this example LIBNAME should be replaced by library name, for instance GSL (i.e. GSL_DIR, GSL_LIB, GSL_INCLUDE).

*Compiling and linking example*

```
module load gsl
xlf_r prog.f -L$GSL_LIB -lgsl
```

# 5.3. MPI

The distributed memory parallelism can be easily exploited by your C/C++ or Fortran codes by means of the MPI (Message Passing Interface) library. On the IBM Power 775 system, a vendor specific version of MPI is available.

## Running Parallel MPI programs interactively

MPI programs on the Power 775 system normally run under the “Parallel Operating Environment” (POE). Programs compiled with the “mp” compiler wrappers automatically invoke POE. Moreover users can use the well known “mpiexec” command which also invokes POE transparently for the user.

There are two different ways to run a parallel program:

**1. Running POE**

```
poe <prog> <program options> <poe options>
```

**Example 1 - explicit POE:**

```
poe ./myexe myparameters -procs 4
```

**Example 2 - implicit POE:**

```
setenv MP_PROCS 4
./myexe myparameters
```

**2. Running mpiexec**

```
mpiexec <mpiexec options> <prog> <program options>
```

**Example 1:**

```
mpiexec -n 4 ./myexe myparameters
```

For more information about poe and mpiexec options and environment variables see the man-pages.

## Running Parallel MPI programs in a batch environment

When the phrase

```
#@ job_type = parallel
```

is present in a job script, the native scheduler of the system (LoadLeveler) generates an environment suitable for parallel jobs. The number of processes is determined from the value of `#@node`, `#@tasks_per_node` or `#@total_tasks` clauses included in the job script.

*Note:* Under control of LoadLeveler, poe ignores the `MP_PROCS` environment variable and the `-procs` flag as well as some other explicit settings. Mpiexec has exactly the same behaviour, e.g. it ignores `-n` flag.

This example is for a job running 128 processes on 4 nodes:

```
#@ node = 4
#@ tasks_per_node = 32
```

This example is for a job, running 8 processes in total.

```
#@ total_tasks = 8
```

# 5.4. OpenMP

For programs compiled with “-qsmp=omp”, the actual number of threads that will be used when executed is defined at the execution time by environmental variable `OMP_NUM_THREADS`. However, the number of threads should be less or equal to the number of hardware threads available on the node (i.e. 128). The default number (if OMP_NUM_THREADS is not set) is the number of available hardware threads on the node (i.e. 128).

## Running parallel OpenMP programs

After the number of threads is set in the environment variable `OMP_NUM_THREADS` the program will run with as many threads as indicated. For example 128 threads:

```
setenv OMP_NUM_THREADS 128
./exename
```

A fragment of a job script for running an OpenMP programs is presented below:

```
...
#@ job_type = serial
#@ queue
setenv OMP_NUM_THREADS 128
./exename
```

Note that you do not have to use poe for pure OpenMP codes that are intended to run on a single node.

# 5.5. MPI-OpenMP hybrid parallelism

As indicated before, a hybrid parallel program needs to be built with any of the “mp” compiler wrappers in combination with the “-qsmp=omp” compiler flag.

## Running hybrid parallel programs

To run the program `hello` on two nodes with 1 MPI process per node and 128 OpenMP threads per node, the following commands need to be executed:

```
setenv OMP_NUM_THREADS 128
poe ./hello -nodes 2 -tasks_per_node 1
```

Here is a fragment of a LoadLeveler script to run the code on two nodes with 2 MPI tasks and 128 OpenMP threads per node.

```
...
#@ job_type = parallel
#@ node = 2
#@ tasks_per_node = 1
#@ queue
setenv OMP_NUM_THREADS 128
./hello
```

# 5.6. Batch system / job command language

A batch job is the typical way users run production applications on HPC machines. The user submits a batch script to the batch system (LoadLeveler). This script specifies, at the very least, how many nodes and cores the job will use, how long the job will run, and the name of the application to run. The job will advance in the queue until the resources are ready, then it will be launched on the compute nodes. The output of the job will be available when the job has completed. On request, the user can be notified with an email for every step of his job within the batch environment.

LoadLeveler is the native batch scheduler for the IBM Power 775 machine.

In order to submit a batch job, a LoadLeveler script file must be written with directives for the scheduler followed by the commands to be executed. The script file has then to be submitted using the llsubmit command.

## The basic LoadLeveler commands

| | |
|---|---|
| llsubmit job.cmd | Submit the job described in the "job.cmd" file (see below for the scripts syntax) |
| llq | Return information about all the jobs waiting and running in the LoadLeveler queues |
| llq -u $USER | Return information about your jobs only in the queues |
| llq -l <jobID> | Return detailed information about the specific job |
| llq -s <jobID> | Return information about why the job remains queued |

| | |
|---|---|
| llcancel <jobID> | Cancel a job from the queues, either it is waiting or running |
| llstatus | Return information about the status of the machine |

On Boreasz system users need to specify a class (queue) name. For a list of available classes and resources limits see the documentation available in */opt/info/queuing_system*.

# LoadLeveler script file syntax

In order to submit a batch job, you need to create a file, *e.g.* job.cmd, which contains

- LoadLeveler directives specifying how much wall-time, processors, resources you wish to allocate to your job.
- shell commands and programs which you wish to execute. The file paths can be absolute or relative to the directory from which you submit the job.

Once created, this file must be submitted with the llsubmit command:

```
> llsubmit job.cmd
```

It will then be queued into a “class”, depending on the directives specified in the script file.

Note that if your job tries to use more resources (memory or time) than requested, it will be killed by the scheduler. On the other hand, if you require more resources than needed, you risk to wait longer before your job is taken into execution. For these reasons, it is important to design your jobs in such a way that they request the right amount of resources.

The first part of the script file contains directives for LoadLeveler specifying the resources needed by the job, in particular:

| | |
|---|---|
| #@ wall_clock_limit = <hh:mm:ss> | Sets the maximum elapsed time for the job. (ex: 24:00:00) |
| #@ network.MPI = mode | Network settings. Recommended settings: sn_all,not_shared,US,HIGH. |
| #@ job_type = parallel<br><br>#@ job_type = serial | “parallel” informs LoadLeveler to setup a parallel (MPI) environment for running the job;<br><br>“serial” is used for serial job or parallel (pure OpenMP only). |
| #@ total_tasks = <Ntask> | Sets the number of MPI tasks. Do not use for pure OpenMP jobs |
| #@ output = file.out | File name for standard output for executed job. |
| #@ error = file.err | File name for standard error for executed job. |
| #@ class = <class name> | Asks for a specific class of the scheduler (for PRACE DECI jobs use the "prace" class) |
| #@ account_no = | Informs LoadLeveler about the accounting number to be charged for this job (place your computational grant identifier here) |
| #@ queue | concludes the LoadLeveler directives |

# Example job scripts

*Serial*

This is a serial (one task) job, requesting 1 hour of wall-time. The job file, the exec file (myjob) and the input file (input) are in the same directory where the job is submitted from. This directory needs to reside on a file system that is mounted on the execution nodes:

```
#@ job_name = myjob
#@ output = myjob.$(jobid).out
#@ error = myjob.$(jobid).err
#@ wall_clock_limit = 1:00:00
#@ network.MPI = sn_all,not_shared,US,HIGH
#@ job_type = serial
#@ account_no = <your_account_number>
#@ class= <job_class>
#@ queue

./myjob < input > output
```

*Pure MPI*

This is a template for a parallel (pure MPI) job on 1 node, 32 cores, requesting 20 minutes of wall-time:

```
#@ job_name = myjob
#@ output = myjob.$(jobid).out
#@ error = myjob.$(jobid).err
#@ wall_clock_limit = 0:20:00
#@ network.MPI = sn_all,not_shared,US,HIGH
#@ job_type = parallel
#@ node = 1
#@ tasks_per_node = 32
#@ account_no = <your_account_number>
#@ class= <job_class>
#@ queue

mpiexec ./myMPI < input > output
```

*Pure OpenMP*

Parallel (shared memory) job: 1 task and 32 threads on 32 cpus on 1 node (smp), requesting 20 mins of wall-time.

**NOTE:** the job_type must be set to “serial”, since it refers to a single task:

```
#@ job_name = myjob
#@ output = myjob.$(jobid).out
#@ error = myjob.$(jobid).err
#@ wall_clock_limit = 0:20:00
#@ network.MPI = sn_all,not_shared,US,HIGH
#@ job_type = serial
#@ node = 1
#@ tasks_per_node = 1
#@ account_no = <your_account_number>
#@ class= <job_class>
#@ queue
setenv OMP_NUM_THREADS 32
echo “Using $OMP_NUM_THREADS threads”
./myOMP < input > output
```

*Hybrid mode MPI/OpenMP*

Hybrid jobs are MPI job where each MPI task is splitted into several threads.

In this example we have a 16 tasks job (MPI) on 2 nodes (8 MPI tasks per node) and 4 threads per task (OpenMP), requesting 20 minutes of wall-time.

```
#@ job_name = myjob
#@ output = myjob.$(jobid).out
#@ error = myjob.$(jobid).err
#@ wall_clock_limit = 0:20:00
#@ network.MPI = sn_all,not_shared,US,HIGH
#@ job_type = parallel
#@ node = 2
#@ tasks_per_node = 8
#@ account_no = <your_account_number>
#@ class= <job_class>
#@ queue
setenv OMP_NUM_THREADS 4
echo “Using 16 MPI tasks, each splitted into $OMP_NUM_THREADS threads”
mpiexec ./myHybrid < input > output
```

*ST and SMT mode*

The POWER7 processor introduces 4-way SMT which allows 4 threads to execute simultaneously on the same core. SMT4 makes one physical core appear as 4 logical cores to the operating system and applications.

When threads execute on cores they can stall waiting for a resource or data. SMT allows multiple threads to share the core thus improving the utilization of the core resources in the presence of stalls. POWER7 cores can operate in three threading modes:

- ST – 1 thread executing on the core at any given time
- SMT2 – 2 threads executing on the core concurrently
- SMT4 – 3 or 4 threads executing on the core concurrently

The default SMT mode on the Boreasz system is SMT4 and it cannot be changed by users. Therefore users should try to prepare their applications with the use of mixed MPI and multhitreading programming model and execute their programms with the use of multiple threads per core.

## Multi-step jobs

Every scheduler has the possibility to chain multiple jobs. In the following, we describe the details to execute a multi-step job with LoadLeveler.

In a typical multi-step job, the various steps are terminated by the #@queue statement. However, you need to tell LoadLeveler not to execute all the steps at the same time (this is the standard behavior), but to wait for the completion of the previous one (except for the first). This is done with the #@dependency keyword.

This is an example where the script to be executed in each step is similar and is reported in the second part of the script after the last #@queue statement. In order to differentiate the different steps you can use the $LOADL_STEP_NAME environmental variable. Be careful: you need to specify #@node and #@tasks_per_node for each step.

```
#@ job_name = STEPjob
```

```
#@ error = $(job_name).$(jobid).$(stepid).err
#@ output = $(job_name).$(jobid).$(stepid).out
#@ wall_clock_limit = 00:10:00
#@ network.MPI = sn_all,not_shared,US,HIGH
#@ job_type = parallel
#@ account_no = <your_account_number>
#@ class= <job_class>
#@ node=1
#@ tasks_per_node=32
#@ step_name=step_0
#@ queue
#@ node=2
#@ tasks_per_node=32
#@ step_name=step_1
#@ dependency = step_0 == 0
#@ queue

switch ( $LOADL_STEP_NAME )
  case step_0:
    setenv OMP_NUM_THREADS 1
    mpiexec ./step1.x
    breaksw
  case step_1:
    setenv OMP_NUM_THREADS 1
    mpiexec ./step2.x
    breaksw
endsw
```

### Important notice

Please always read the information that is printed out on the screen during login. Furthermore, please use the documentation files in */opt/info*.

# 6. Tuning Applications

In this Section several ways to tune applications will be presented, including compiler options and advanced options for parallel usage and programming. Besides these tuning options, affinity is important, because the key to performance often lies herein. In order to better understand the affinity topics in this Section, we briefly summarize the relevant architectural features of a Power 775 node.

A Power 775 node is often referred to as QCM (Quad-Chip Module), since it consists of four Power7 processor chips. In IBM terminology it is also referred as “octant” since there are eight such nodes in a single CEC. Although this is very misleading, for historical reasons the AIX system and the LoadLeveler queueing system identify a single node as a set of four MCMs (by definition of a Multi-Chip Module Power 775 node should rather be referred to as a single MCM). Therefore a single MCM for AIX and LoadLeveler is equal to a single Power7 processor together with the RAM memory assigned to it.

Each core is capable of four-way simultaneous multithreading (SMT). A node has 32 cores and can run 128 hardware threads. Both in LoadLeveler and in the AIX, the hardware threads are called cpus, i.e.: core0 = cpu0 + cpu1 + cpu2 + cpu3, core1 = cpu4 + cpu5 + cpu6 + cpu7, ..., core31 = cpu124 + cpu125 + cpu126 + cpu127. These of course are virtual (or logical) cpus, however, the word “virtual” or “logical” is commonly dropped.

Processor affinity takes advantage of the fact that some remnants of a process may remain in one processor's state (in particular, in its cache) from the last time the process ran. Scheduling it to run on the same processor next time could result in the process running more efficiently by reducing performance-degrading situations such as cache misses. This minimizes thread migration and context-switching cost among cores. It also improves the data locality and reduces the cache-coherency traffic among the cores (or processors).

Although the MCMs are fully connected, MCMs are NUMA nodes (NUMA domains): intra node memory access to/from another MCM is slower. MCM0 = cpu0, ..., cpu31; MCM1 = cpu32, ..., cpu63; ...; MCM3 = cpu96, ..., cpu127.

*Note:* the current version of Section 6 is written for the AIX operating system, IBM Parallel Environment (a.o. for MPI), IBM XL compilers (most relevant for OpenMP) and LoadLeveler (because of the tight integration with IBM PE and the OpenMP implementation of the IBM XL compilers).

# 6.1. Advanced / aggressive compiler flags

This section will cover some compiler flags needed for tuning an application.

Before going into IBM (hardware) specific compiler options, let's first have a look at the general optimizations. IBM specific options will be explained in the next section.

The general optimizations include:

- Function inlining
- Dead code elimination
- Array dimension padding
- Structure splitting and field reordering

These optimizations can be performed with the flag -O*n*, where *n* is the level of optimization. In the end, these optimizations lead to a smaller and more efficient code.

In the following tables you can see the different levels of optimization for IBM XL:

| IBM XL option | Description |
|---|---|
| -O0 | Performs only quick local optimizations such as constant folding and elimination of local common subexpressions. |
| -O2 | Performs optimizations that the compiler developers considered the best combination for compilation speed and runtime performance. The optimizations may change from product release to release. |
| -O3 | Performs some memory and compile-time intensive optimizations in addition to those executed with -O2. The -O3 specific optimizations have the potential to alter the semantics of a program. The compiler guards against these optimizations at -O2 and the option -qstrict is provided at -O3 to turn off these aggressive optimizations. Specifying -O3 implies -qhot=level=0. |
| -O4 | This option is the same as -O3, but also:<br>• sets the -qarch and -qtune options to the architecture of the compiling machine.<br>• sets the -qcache option most appropriate to the characteristics of the compiling machine.<br>• sets the -qipa option.<br>• sets the -qhot option to level=1. |
| -O5 | Equivalent to -O4 -qipa=level=2 |

There are also some aggressive compiler flags, such as:

| IBM XL option | Description |
|---|---|
| -qhot | Instructs the compiler to perform high-order loop analysis and transformations during optimization. |
| -qipa | Interprocedural analysis (IPA) enables the compiler to optimize across different files (whole-program analysis), and can result in significant performance improvements. |
| -qinline | Inline functions are expanded in any context in which it is called. This avoids the normal performance overhead associated with the branching for a function call, and it allows functions to be included in basic blocks. |

In addition, there are some specific flags available for IBM Power7 architecture that should be always used during compilation:

| IBM XL option | Description | |
|---|---|---|
| -qarch=pwr7 | Specifies the architecture system for which the executable is optimized. | |
| -qtune=pwr7 | Specifies additional performance optimizations. | |

IBM specific processor features include Altivec/VSX, a floating point and integer SIMD instruction set, which allows to exploit the SIMD and parallel processing capabilities of the Power7 processor. It can be switched on in the XL compiler suite by compiling with special options. The recommended command lines are:

For C:

```
xlc -qarch=pwr7 -qtune=pwr7 -O3 -qhot -qaltivec
```

For Fortran:

```
xlf -qarch=pwr7 -qtune=pwr7 -O3 -qhot
```

# 6.2. Advanced MPI usage

## 6.2.1. Tuning / environment variables

The text for this Section has mostly been taken from “Performance Guide For HPC Applications On IBM Power 775 System” by Jim Abeles *et al.* [3].

For better performance, the following environment variables can be used on top of the LoadLeveler or PE affinity settings (see Section 6.2.3 and Section 6.3.1). Please note that the influence of those settings for particular application's performance may differ and should be always measured with tests before production runs.

```
setenv MP_FIFO_MTU 4K
setenv MP_RFIFO_SIZE 16777216
setenv MP_EUIDEVELOP min
```

[3] op. cit.

```
setenv MP_USE_BULK_XFER yes
setenv MP_PULSE 0
setenv MP_BUFFER_MEM 512M
setenv MP_EUILIB us
```

Large computer systems make use of remote direct memory access (RDMA). RDMA supports zero-copy networking by enabling the network adapter to transfer data directly to or from application memory, eliminating the need to copy data between application memory and the data buffers in the operating system. Such transfers require no work to be done by CPUs, caches, or context switches, and transfers continue in parallel with other system operations. When an application performs an RDMA Read or Write request, the application data is delivered directly to the network, reducing latency and enabling fast message transfer. Specific RDMA variables:

```
setenv MP_RDMA_MTU 4K
setenv MP_BULK_MIN_MSG_SIZE 64k
setenv MP_RC_MAX_QP 8192
```

**MP_FIFO_MTU=4K**

Unless this is set, FIFO/UD uses a default 2K packet. FIFO bandwidths can be improved by setting this variable to “4K” to enable 4KB MTU for FIFO packets. Note that the switch chassis MTU must be enabled for 4KB MTU for this to work.

**MP_RFIFO_SIZE=16777216**

The default size of the receive FIFO used by each MPI task is 4MB. Larger jobs are recommended to use the maximum receive FIFO size by setting `MP_RFIFO_SIZE=16777216`.

**MP_EUIDEVELOP=min**

The MPI layer will perform checking on the correctness of parameters according to the value of `MP_EUIDEVLOP`. As these checks can have a significant impact on latency, when not developing applications it is recommended that `MP_EUIDEVELOP=min` be set to minimize the checking done at the message passing interface layer.

**MP_USE_BULK_XFER=yes**

Setting `MP_USE_BULK_XFER=yes` will enable RDMA. The benefit of RDMA depends on the application and its use of buffers. For example, applications that tend to re-use the same address space for sending and receiving data will do best, as they avoid the overhead of repeatedly registering new areas of memory for RDMA.

RDMA mode will use more memory than pure FIFO mode. Note that this can be curtailed by setting MP_RC_MAX_QP to limit the number of RDMA QPs that are created.

**MP_PULSE=0**

POE and the Partition Manager use a pulse detection mechanism to periodically check each remote node to ensure that it is actively communicating with the home node. You specify the time interval (or pulse interval), with the MP_PULSE environment variable. During an execution of a POE job, POE and the Partition Manager daemons check at the interval you specify that each node is running. When a node failure is detected, POE terminates the job on all remaining nodes and issues an error message. The default value for MP_PULSE is 600 (600 seconds or 10 minutes). Settings MP_PULSE=0 disable the pulse mechanism.

**MP_BUFFER_MEM=512M**

Setting MP_BUF_MEM can increase the memory available to the protocols beyond the defaults. Setting MP_BUF_MEM can address (“MPCI_MSG: ATTENTION: Due to memory limitation eager limit is reduced to X”) cases in which the user-requested MP_EAGER limit cannot be satisfied by the protocols.

**MP_RDMA_MTU=4K**

Unless this is set, RDMA uses the default of 2K packet. RDMA bandwidths can be improved by setting this variable to “4K” to enable 4KB MTU for RDMA packets. Note that the switch chassis MTU must be enabled for 4KB MTU for this to work. In cases where network latency dominates, and for certain message sizes, keeping the MTU size at 2K will provide better performance.

The minimum size message used for RDMA is defined to be the maximum of `MP_BULK_MIN_MSG_SIZE` and `MP_EAGER_LIMIT`. So if `MP_EAGER_LIMIT` is defined to be higher than `MP_BULK_MIN_MSG_SIZE`, the smallest RDMA message will be limited by the eager limit.

**`MP_RC_MAX_QP=8192`**

This variable defines the maximum number of RDMA QPs that will be opened by a given MPI task. Depending on the size of the job and the number of tasks per node, it may be desirable to limit the number of QPs used for RDMA. By default, when the limit of RDMA QPs is reached, future connections will all use FIFO/UD mode for message passing.

## 6.2.2. Mapping tasks on node topology

Since all nodes are fully connected, all nodes are created equal and performance-wise there is no need to map tasks to specific nodes.

## 6.2.3. Task affinity

The key to MPI performance often lies in processor affinity of MPI tasks. The simplest way to achieve processor affinity for MPI tasks is using the `task_affinity` LoadLeveler keyword:

```
#@ tasks_per_node = 4
#
#@ rset = rset_mcm_affinity
#@ mcm_affinity_options = mcm_distribute mcm_mem_pref
#@task_affinity = core(x)
```

or

```
#@ tasks_per_node = 4
#
#@ rset = rset_mcm_affinity
#@ mcm_affinity_options = mcm_distribute mcm_mem_pref
#@task_affinity = cpu(x)
```

The `core` keyword specifies that each task in the job is bound to run on as many processor cores as specified by x. The `cpu` keyword indicates that each task of the job is constrained to run on as many cpus as defined by x.

If you want more control over the task placement, you can use the launch tool. It is also very simple to use.

The target cpus for the MPI tasks are selected through the environment variables `TARGET_CPU_LIST`. The value “-1” denotes automatic detection otherwise list like “0 4 8 12 16 20 ... 124” or “0,7-3,11-14,23-27:2,46-37:3” should be used.

To obtain the affinitization provided by `launch` simply replace program invocations such as

```
...
#@ queue
mpiexec ./MpiProgram ...
```

by

```
...
#@ queue
module load launch
mpiexec launch ./MpiProgram ...
```

### 6.2.4. Adapter affinity

On Boreasz system all jobs should use the following adapter affinity settings:

```
#@ network.MPI = sn_all,not_shared,US,HIGH
```

## 6.3. Advanced OpenMP usage

### 6.3.1. Tuning, Environment variables and Thread affinity

There appears to be no LoadLeveler support for pure OpenMP affinity. The simplest way to achieve processor affinity for OpenMP threads is to use environment variables. Furthermore, the OpenMP standard does not prescribe environment variables for this. Consequently, every compiler vendor uses its own environment variables. For the IBM XL compilers you can use `XLSMPOPTS` `startproc` and `stride` sub options:

```
export OMP_NUM_THREADS=32
export XLSMPOPTS=startproc=0:stride=4
```

Processor numbers refer to virtual or logical CPUs 0..127. Therefore, the specification `startproc=0:stride=4` binds the OpenMP threads to the 32 physical cores.

The example above illustrates the most important use of the `XLSMOPTS` environment variable. Besides `startproc` and `stride`, `XLSMOPTS` has more options. Please check IBMs web site [http://publib.boulder.ibm.com/infocenter/comphelp/v101v121/index.jsp?topic=/com.ibm.xlf121.aix.doc/proguide/xlsmpopts.html] for more detailed information.

## 6.4. Hybrid programming

### 6.4.1. Optimal tasks / threads strategy

As already indicated each node of the Power 775 system contains 4 MCMs. For a hybrid run, the user is strongly advised to use a multiple of 4 MPI tasks per node: each task uses its own MCM, or in case of a multiple of 4, an equal number of tasks runs on each MCM.

### 6.4.2. Task and thread affinity

The simplest way to achieve processor affinity for hybrid MPI tasks / OpenMP threads is using the `hybrid_launch` tool.

The target cpus for the MPI tasks / OpenMP threads are selected through the environment variables `TARGET_CPU_LIST`. The value “-1” denotes automatic detection otherwise list like “0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62” or “0,7-3,11-14,23-27:2,46-37:3” should be used.

To obtain the affinitization provided by `hybrid_launch` simply replace program invocations such as

```
...
#@ queue
mpiexec ./HybridProgram ...
```

by

```
...
#@ queue
module load hybrid_launch
mpiexec hybrid_launch ./HybridProgram ...
```

# 6.5. Memory optimization

## 6.5.1. Memory affinity (MPI/OpenMP/Hybrid)

In the current LoadLeveler implementation, memory affinitization (binding) only appears to work for pure MPI program. The following sequence of LoadLeveler keywords results in every MPI tasks being bound to core and the corresponding memory to be bound to the MCM that contains that core.

```
#@ rset = rset_mcm_affinity
#@ mcm_affinity_options = mcm_distribute mcm_mem_req mcm_sni_none
#@ task_affinity = core
```

Please note that above settings should not be used together with `launch` or `hybrid_launch` tools.