Best Practice Guide SuperMUC v1.0

Nikos Anastopoulos, NTUA/GRNET, Greece

Petri Nikunen, CSC, Finland

Volker Weinberg (Editor), LRZ, Germany <weinberg@lrz.de>

02-May-2013

Table of Contents

1. Introduction 5
2. System architecture & configuration 6
2.1. General informations about SuperMUC 6
2.1.1. SuperMUC 6
2.1.2. System purpose and target users 6
2.1.3. System overview 6
2.1.4. Energy efficiency 7
2.1.5. Why "warm" water cooling? 7
2.2. System configuration 7
2.2.1. System parameters 7
2.2.2. System software 10
2.2.3. Storage systems 10
2.2.4. Processor architecture 11
2.2.5. Operating environment 15
2.2.6. Filesystems 16
2.2.7. Transfer of files between SuperMIG and SuperMUC 19
2.2.8. Transferring files from/to other systems 19
2.2.9. Quota 19
3. System access 20
3.1. Application for an account / project 20
3.2. Contact, support and servicedesk 20
3.2.1. Selfservice portal 21
3.2.2. Simple submission site 21
3.2.3. Servicedesk phone 21
3.2.4. Persons 21
3.3. Login 22
3.3.1. Login to SuperMUC/SuperMIG 22
3.3.2. Login with secure shell 22
3.3.3. Login via grid services using GSI-SSH 23
3.4. Password policies 23
3.5. Changing password or login-shell, viewing accounting data 24
3.6. Budget and quotas 24
3.7. Moving data from/to SuperMUC 24
3.7.1. SCP 24
3.7.2. SFTP 25
3.7.3. GridFTP 25
3.8. Access to SVN server 25
3.8.1. Using SVN with a https svn server 25
3.8.2. Using SVN+SSH repository access 25
3.9. Supported services to and from SuperMUC 26
3.9.1. Incoming (to SuperMUC) 26
3.9.2. Outgoing (from SuperMUC) 26
4. Production environment 26
4.1. Accounting, budgets and quota 26
4.2. Software packages 27
4.3. Module environment 27
4.4. LRZ tools 27
4.5. LRZ library 28
4.6. Node allocation policy 29
4.6.1. Jobs size: totaltasks, number of nodes and tasks per node 29
4.7. Interactive parallel jobs 30
4.7.1. IBM MPI 30
4.7.2. Intel MPI 30
4.7.3. Limited resources for job class "test" 30
4.8. Batch-jobs with LoadLeveler 31

4.9. Job command file 32
4.10. Examples for job command files for SuperMIG (Fat Nodes) 32
4.10.1. Parallel MPI job (IBM MPI) 32
4.10.2. Parallel MPI job (Intel MPI) 32
4.10.3. Hybrid MPI job (IBM MPI) 33
4.10.4. Hybrid MPI/OpenMP job (Intel MPI) 33
4.10.5. Pure OpenMP job (on fat node island) 34
4.11. Examples for job command files for SuperMUC (Thin Nodes) 34
4.11.1. Parallel MPI Job (IBM MPI) 34
4.11.2. Parallel MPI Job (Intel MPI) 35
4.11.3. Hybrid MPI Job (IBM MPI) 35
4.11.4. Hybrid MPI/OpenMP job (Intel MPI) 36
4.11.5. Pure OpenMP job (on thin node island) 37
4.12. Job command file keywords 37
4.12.1. Job name 37
4.12.2. Job type 37
4.12.3. Job class 38
4.12.4. Limits 39
4.12.5. Parallel jobs 39
4.12.6. Restart 41
4.12.7. Files and directories 41
4.12.8. Executable and arguments 41
4.12.9. Job steps 42
4.12.10. Variables 43
4.12.11. Notification 44
4.13. Working with energy aware jobs on SuperMUC 44
4.14. Querying the status of a job 45
4.14.1. Fields in llq's listing 46
4.15. Using llrun to start interactive or batch jobs 47
4.16. Why isn't my job running? 48
4.17. Further information 48
5. Programming environment 49
5.1. Development tools and libraries 49
5.1.1. Overview 49
5.1.2. Eclipse (integrated development environment) 49
5.2. Compilers and parallel programming 50
5.2.1. Intel compilers and performance libraries 50
5.2.2. GNU compilers 50
5.2.3. PGI compilers 50
5.2.4. MPI 50
5.2.5. OpenMP 50
5.3. An overview of compiler functionality 50
5.3.1. Optimization options for x86_64 processors 51
5.3.2. Options for code transformations, aliasing and interprocedural optimization 51
5.3.3. Linkage options 52
5.3.4. Source format and preprocessing 53
5.3.5. Options for data and I/O 53
5.3.6. Diagnostics, runtime checking and debugging 54
5.3.7. Parallelization and vectorization options 55
5.3.8. Compiler directives for the Intel compiler 56
5.3.9. Threading Tools 57
5.4. Parallelization models supported on SuperMUC 57
5.4.1. MPI 57
5.4.2. OpenMP 58
5.5. Supported MPI implementations 58
5.5.1. IBM MPI 58
5.5.2. Intel MPI 59
5.6. OpenMP - Parallel programming on shared memory systems 61

5.6.1. An abstract description of OpenMP 61
5.6.2. Overview of OpenMP functionality 63
5.6.3. Remarks on the usage of OpenMP 63
5.6.4. References, examples and documentation 66
5.7. Vectorization 66
5.7.1. Overview 66
5.7.2. Automatic vectorization by compiler 67
5.7.3. Cilk Plus language extensions for C and C++ 69
5.7.4. Vectorization-friendly features of Fortran 69
5.7.5. References 70
6. Performance analysis 70
6.1. Available performance analysis tools 70
6.1.1. Loading the tools 70
6.1.2. Information 70
6.1.3. Timing & profiling 70
6.1.4. Using hardware perfomance counters 71
6.1.5. MPI and OpenMP profiling and tracing 71
6.1.6. Memory Leaks 72
6.2. Westmere-EX/Sandy-Bridge-EP specific performance counters 73
6.2.1. Performance Monitoring 73
6.2.2. Using performance counters to detect performance problems 73
6.2.3. Complete event list 78
7. Tuning 78
7.1. Westmere/Sandy-Bridge specific single-core optimization 78
7.1.1. Processor-specific optimization 78
7.1.2. Vectorization 78
7.2. Advanced OpenMP tuning 82
7.2.1. Iteration scheduling 82
7.2.2. Thread affinity 83
7.2.3. Synchronization 85
7.2.4. General tuning 85
7.3. Advanced MPI tuning 86
7.3.1. IBM MPI 86
7.3.2. Intel MPI 96
7.4. Hybrid programming 97
7.4.1. General 97
7.4.2. Optimal MPI processes/OpenMP threads configuration 98
7.4.3. Enforcing processor/memory affinity 99
8. Debugging 100
8.1. Debuggers with graphical interface (GUI) 100
8.2. Table of available debuggers and info 100
9. Further documentation 100
9.1. IBM Parallel Operating Environment: POE 100
9.2. IBM LoadLeveler 100
9.3. IBM Central 101
9.4. Intel compiler, libraries and tools 101
9.5. Intel optimization and tuning 101
9.6. Intel 64 and IA-32 architectures software developer's manuals 101

1. Introduction

The Tier-0 system SuperMUC is hosted by the Gauss Centre for Supercomputing (GCS) at the Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften (Leibniz Supercomputing Centre of the Bavarian Academy of Science and Humanities) in Garching near Munich. In June 2012 SuperMUC was Europe's fastest supercomputer and the number 4 on the world-wide TOP500 list. Already starting in August 2011 a migration system nicknamed SuperMIG was provided to port applications to the new programming environment. The migration system will be integrated as one fat-node island into the SuperMUC system.

Figure 1. SuperMUC hosted by LRZ

This best practice guide provides information about SuperMUC in order to enable users of the system 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 including how to login and the accounting procedure as well as information about porting and submitting jobs, up to tools and strategies on how to analyze and improve the performance of applications.

The guide includes contributions from LRZ, Nikos Anastopoulos (NTUA/GRNET) and Petri Nikunen (CSC).

SuperMUC is a x86-based system. A generic x86 best practice guide can be found on the PRACE best pratice guides webpage. [http://www.prace-ri.eu/Best-Practice-Guides]

This guide is written within PRACE, the Partnership for Advanced Computing in Europe. We try to synchronize the content of this guide in regular intervals with LRZ online documentation. Most up-to-date documentation about SuperMUC is available online: LRZ SuperMUC online documentation [http://www.lrz.de/services/compute/supermuc/].

2. System architecture & configuration

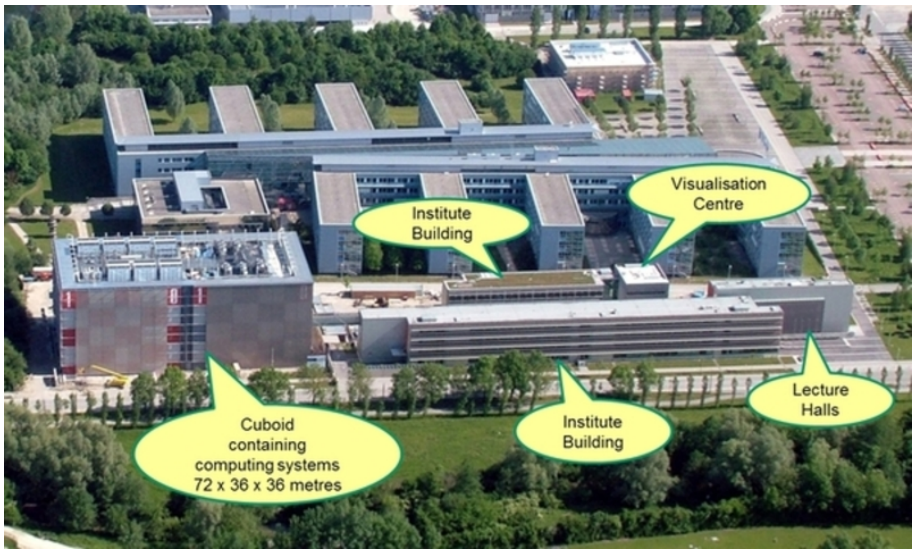
2.1. General informations about SuperMUC

2.1.1. SuperMUC

SuperMUC is the name of the new supercomputer at Leibniz-Rechenzentrum (Leibniz Supercomputing Centre) in Garching near Munich (the MUC suffix is borrowed from the Munich airport code). With more than 155.000 cores and a peak performance of 3 Petaflop/s (=10^15 Floating Point Operations per second) in June 2012 SuperMUC is one of the fastest supercomputers in the world.

SuperMUC is housed in the recently expanded compute cuboid of LRZ.

Figure 2. LRZ buildings (Photo: E. Graf, TU München)



2.1.2. System purpose and target users

SuperMUC strengthens the position of Germany's Gauss Centre for Supercomputing [http://www.gauss-centre.eu/] in Europe by delivering outstanding compute power and integrating it into the European High Performance Computing ecosystem. With the operation of SuperMUC, LRZ will act as an European Centre for Supercomputing and will be Tier-0 centre of PRACE, the Partnership for Advanced Computing in Europe [http://www.prace-ri.eu/].

SuperMUC is available to all European researchers to expand the frontiers of science and engineering.

Since August 2011 a migration system (nicknamed SuperMIG) enables porting applications to the new programming environment. SuperMUC will be fully operational in July 2012.

2.1.3. System overview

- 155,656 processor cores in 9400 compute nodes,
- >300 TB RAM,
- Infiniband FDR10 interconnect,
- 4 PB of NAS-based permanent disk storage,
- 10 PB of GPFS-based temporary disk storage,
- >30 PB of tape archive capacity,
- powerful visualization systems,

- highest energy-efficiency.

2.1.4. Energy efficiency

SuperMUC uses a new, revolutionary form of warm water cooling developed by IBM. Active components like processors and memory are directly cooled with water that can have an inlet temperature of up to 40 degrees Celsius. The "High Temperature Liquid Cooling" together with very innovative system software promises to cut the energy consumption of the system. In addition, all LRZ buildings will be heated re-using this energy.

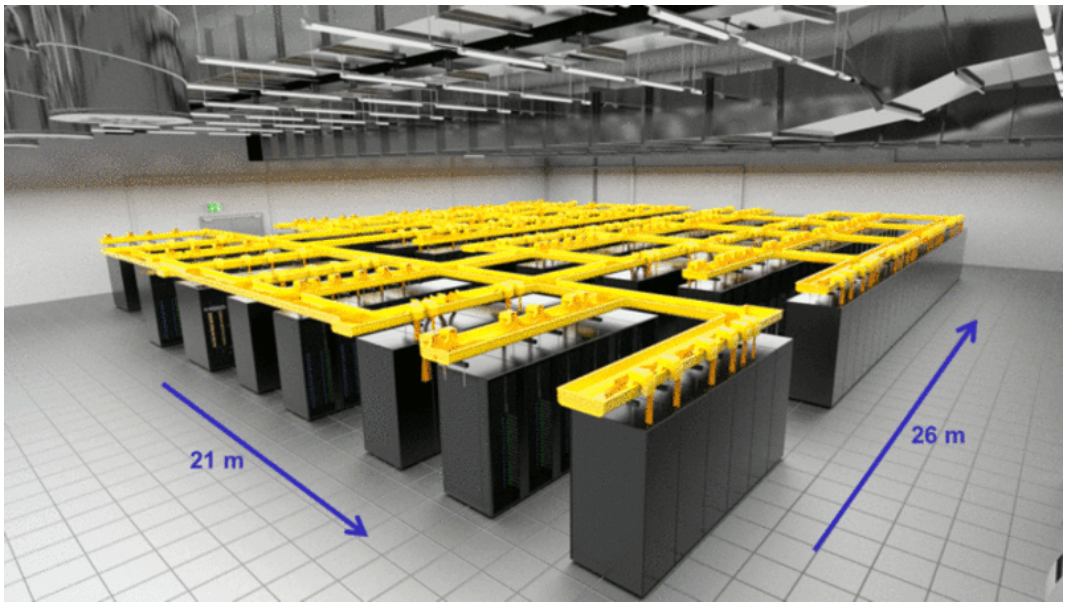
2.1.5. Why "warm" water cooling?

Typically water used in data centers has an inlet temperature of approx 16 degrees Celsius and, after leaving the system, an outlet temperature of approx. 20 degrees Celsius. To make water with 16 degrees Celsius requires complex and energy-hungry cooling equipment. At the same time there is hardly any use for the warmed-up water as it is too cold to be used in any technical processes.

SuperMUC allows an increased inlet temperature. It is easily possible to provide water having up to 40 degrees Celsius using simple "free-cooling" equipment as outside temperatures in Germany hardly ever exceed 35 degrees Celsius. At the same time the outlet water can be made quite hot (up to 70 degrees Celsius) and re-used in other technical processes - for example to heat buildings or in other technical processes.

By reducing the number of cooling components and using free cooling LRZ expects to save several millions of Euros in cooling costs over the 5-year lifetime of the system.

Figure 3. SuperMUC in the computer room



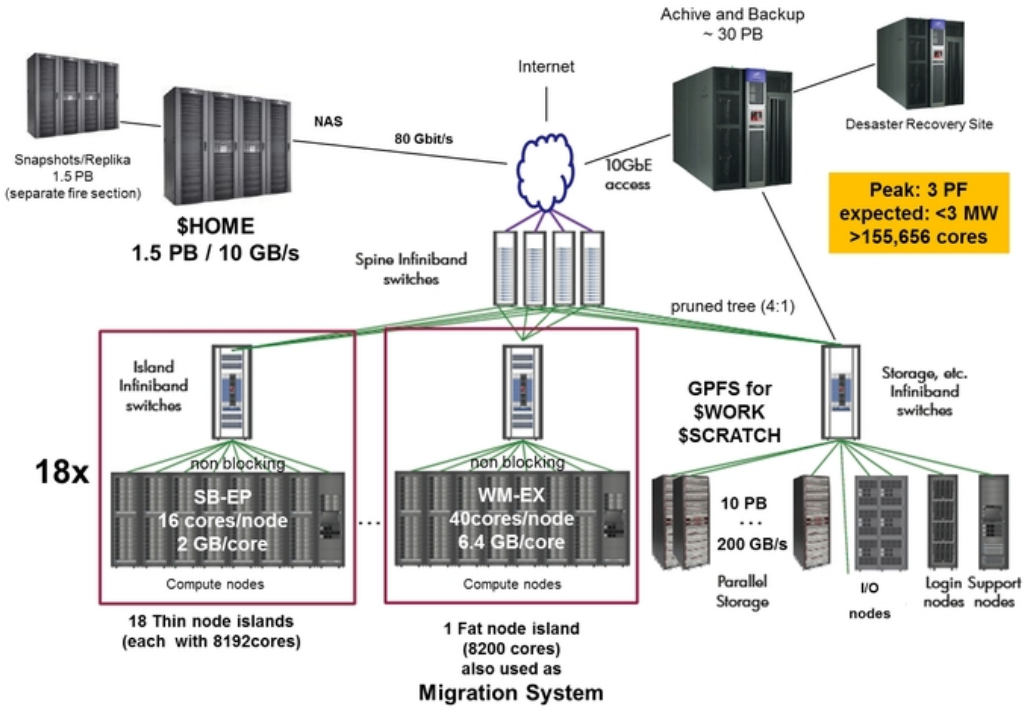
2.2. System configuration

2.2.1. System parameters

LRZ's target for the architecture is a combination of a large number of moderately powerful compute nodes, with a peak performance of several hundred GFlop/s each, and a small number of fat compute nodes with a large shared memory. The network interconnect between the nodes allows for perfectly linear scaling of parallel applications up to the level of more than 10,000 tasks.

SuperMUC consists of 18 Thin Node Islands and one Fat Node Island which is at first also used as the Migration System SuperMIG. Each Island contains more than 8,192 cores. All compute nodes within an individual Island are connected via a fully non-blocking Infiniband network (FDR10 for the Thin nodes / QDR for the Fat Nodes). Above the Island level, the high speed interconnect enables a bi-directional bi-section bandwidth ratio of 4:1 (intra-Island / inter-Island).

Figure 4. Schematic view of SuperMUC



Details of SuperMUC are given in the following table.

Item	Thin Node Islands	Fat Node Island	Migration system
System	IBM System x iDataPlex	BladeCenter HX5	BladeCenter HX5
Processor Types (Thin + Fat)	Sandy Bridge-EP Intel Xeon E5-2680 8C	Westmere-EX Intel Xeon E7-4870 10C	Westmere-EX Intel Xeon E7-4870 10C
Number of Islands (Thin + Fat)	18	1	1
Nodes per Island	512	205	205
Processors per Node	2	4	4
Cores per Processor	8	10	10
Cores per Node	16	40	40
Logical CPUs per Node (Hyperthreading)	32	80	80
Nodes per Island	512	205	205
Total Number of nodes	9216	205	205
total Number of cores	147,456	8200	8200
Peak Performance [PFlop/s]	3.185	0.078	0.078
Linpack Performance [PFlop/s]	2.897	0.065	0.065
Total size of memory [TByte]	288	52	52
Memory per Core [GByte] (typically available for applications)	2(~1.5)	6.4(~6.0)	6.4

Item	Thin Node Islands	Fat Node Island	Migration system
Size of shared Memory per node [GByte]	32	256	256
Bandwidth to Memory per noder [Gbyte/s]	102.4	136.4	136.4
Latency to local memory [ns (cylces)]	~ 50 (~135)	~70 (~170)	~70 (~170)
Latency to remote memory [ns (cylces)]	~ 90 (~240)	~120 (~200)	~120 (~200)
Level 3 Cache Size (shared) [Mbyte]	20	24	24
Level 2 Cache Size [kByte]	256	256	256
Level 1 Cache Size [kByte], Associativity	32@ 8 way	32	32
Level 3 Cache Bandwidth and Latency (shared) [byte/cycle]	1 x 32 @ 31 cycles	1 x 32	1 x 32
Level 2 Cache Bandwidth and Latency [byte/cycle] Latency is much longer, if data are also in L1 or L2 of other core.	1 x 32 @ 12 cycles	1 x 32	1 x 32
Level 1 Cache Bandwidth and Latency [byte/cycle] Latency is much longer, if data are also in L1 or L2 of other core.	2 x 16 @ 4 cycles	2 x 16	2 x 16
Level 3 Cache line Size [Byte]	64	64	64
Expected electrical power consumption of total system [MW]	< 3		< 0.21
Network Technology	Infiniband FDR10		Infiniband QDR
Intra-Island Topology	non-blocking Tree		non-blocking Tree
Inter-Island Tolopogy	Pruned Tree 4:1		n.a.
Bisection bandwidth of Interconnect [TByte/s]	35.6		n.a.
Filesystem for SCRATCH and WORK	IBM GPFS		NetApp NAS
File System for HOME	NetApp NAS		NetApp NAS
Size of parallel storage [Pbyte]	10		n.a.
Size of NAS user storage [PByte]	1.5 (+ 1.5 for replication)		1
Aggregated bandwidth to/from GPFS [GByte/s]	200		n.a.
Aggregated bandwidth to/from NAS storage [GByte/s]	10		n.a.

Item	Thin Node Islands	Fat Node Island	Migration system
Login Servers for users	5		2
Service and management Servers	12		4
Batchsystem	IBM Loadleveler		
Archive and Backup Software	IBM TSM		
Planed Capacity of Archive and Backup Storage [PByte]	> 30		

2.2.2. System software

SuperMUC uses following software components:

- Suse Linux Enterprise Server [http://www.suse.com/] (SLES)
- System management: xCat from IBM [http://www.ibm.com/systems/software/xcat/index.html]
- Batch processing: Loadleveler from IBM [http://www-03.ibm.com/systems/software/loadleveler/]

From the user side a wide range of compilers, tools and commercial and free applications is provided. Many scientists also build and run their own software.

2.2.3. Storage systems

SuperMUC has a powerful I/O-subsystem which helps to process large amounts of data generated by simulations.

2.2.3.1. Home file systems

Permanent storage for data and programs is provided by a 16-node NAS cluster from Netapp. This primary cluster has a capacity of 2 Petabytes and has demonstrated an aggregated throughput of more than 10 GB/s using NFSv3. Netapp's Ontap 8 "Cluster-mode" [http://media.netapp.com/documents/tr-4067.pdf] provides a single namespace for several hundred project volumes on the system. Users can access multiple snapshots of data in their home directories.

Data is regularly replicated to a separate 4-node Netapp cluster with another 2 PB of storage for recovery purposes. Replication uses Snapmirror-technology and runs with up to 2 GB/s in this setup.

Storage hardware consists of >3400 SATA-Disks with 2 TB each protected by double-parity RAID and integrated checksums.

2.2.3.2. Work and scratch areas

For highest-performance checkpoint I/O, IBM's General Parallel File System (GPFS) [http://www-03.ibm.com/systems/software/gpfs/] with 10 PB of capacity and an aggregated throughput of 200 GB/s is available. Disk storage subsystems were built by DDN [http://www.ddn.com/products/sfa12k].

2.2.3.3. Tape backup and archives

LRZ's tape backup and archive systems based on TSM (Tivoli Storage Manager) from IBM [http://www-01.ibm.com/software/tivoli/products/storage-mgr/] are used for archiving and backup. They have been extended to provide more than 30 Petabytes of capacity to the users of SuperMUC. Digital long-term archives help to preserve results of scientific work on SuperMUC. User archives are also transferred to a disaster recovery site.

2.2.4. Processor architecture

The migration system and fat node island of SuperMUC are equipped with Westmere-EX processors. The thin node island is equipped with Sandy-Bridge-EP processors. The following sections discuss the architectural features for each of these processor models.

2.2.4.1. Westmere-EX processor

Westmere-EX is the server variant of the 32nm shrink of Intel's Nehalem microarchitecture. The first Westmere-based processors were launched in 2010. The most important features of the processor microarchitecture are listed below:

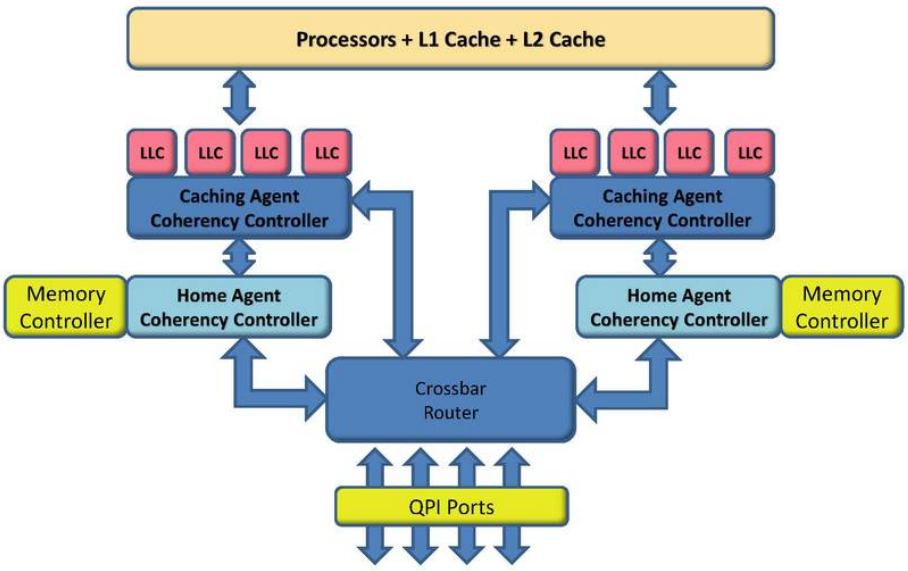
- *Hyper-Threading technology (HTT)*: Westmere incorporates an improved implementation of Intel's HTT. HTT implements two hardware contexts (threads) on top of a single processor core, and multiplexes dynamically their execution. The two contexts appear to the operating system as different CPUs, but in reality they share the vast majority of processor resources (caches, execution units, branch predictors, etc.). HTT aims at increasing the throughput of a conventional superscalar processor. Its rationale is to exploit the empty pipeline slots of the one thread that occur e.g. due to low instruction-level parallelism or cache misses, by scheduling instructions of the second thread.

- *Quick-Path Interconnect (QPI)* along with *Integrated Memory Controllers (IMC)*: QPI is a high-bandwidth, low-latency point-to-point system interconnect which provides fast and scalable data transfers between sockets. It enables highly scalable configurations for systems with varying number of sockets. QPI assumes that the processor sockets will have memory controllers integrated into them (and not in the chipset), which is the case in Westmere-EX processor. Westmere-EX features 4 DDR3 channels per socket.

  In this memory architecture, each socket has its local memory module, memory bus and IMC. This provides separate and potentially uncontended paths to main memory, which enables scalability. On the other hand, since memory is located in multiple sites on the platform, accessing it incurs varying latency, with local accesses being faster than remote ones. This gives Non-Uniform Memory Access (NUMA) characteristics to the architecture.

- *Uncore*: The portion of the processor that ties together the cores with the "outside world" is called the *Uncore*. It includes all the chip components outside the CPU core: IMCs, socket-to-socket interconnects, multi-core memory coherence controller, large shared cache, and the chip-level clocking, power-delivery and debug mechanisms that tie the various on-die components together. The following picture presents a block diagram for the Westmere-EX Uncore.

Figure 5. Westmere-EX uncore



- Other architectural features:

- Second-level branch predictor and translation lookaside buffer (TLB).
- 64 KB L1 cache/core (32 KB L1 Data + 32 KB L1 Instruction) and 256 KB L2 cache/core.
- 4–12 MB L3 cache, shared by all cores of the socket. It implements the *inclusive property*, meaning that a cache line residing in L1/L2, *must* be present in the L3 cache, too.
- Macro- and Micro-fusion techniques, that reduce the number of micro-instructions decoded/executed/retired.
- Loop Stream Detector optimized for power efficiency.
- Enlarged buffers (Re-Order Buffer, Reservation Station) for improved out-of-order execution (increased instruction-level parallelism).
- A new set of instructions that gives over 3x the encryption/decryption rate of AES algorithm.
- Integrated graphics added into the processor package (only in dual core Arrandale and Clarkdale models).
- 2nd generation Intel Virtualization Technology, along with improved virtualization latency and new virtualization capabilities.
- Support for "Huge Pages" of 1 GB in size.
- Reduced latency and greater scalability for native atomic instructions (LOCK prefixed, XCHG).

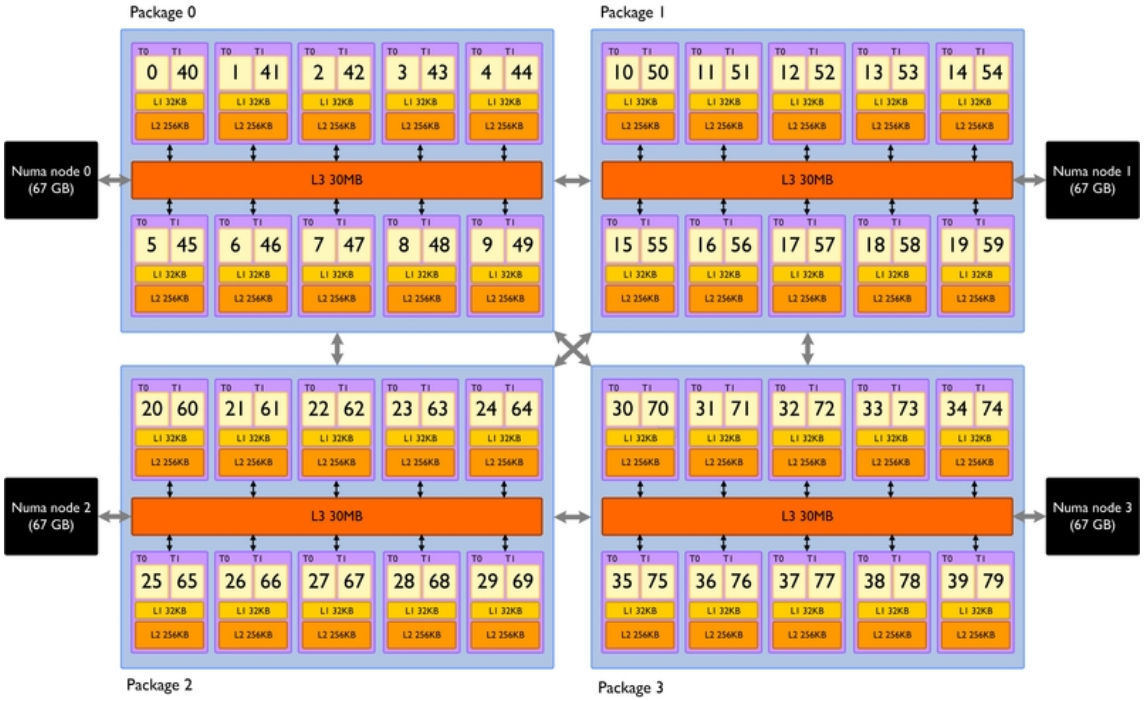
Specifically for the SuperMUC Fat Nodes, the processor configuration is shown in the following table:

Number of Sockets (NUMA nodes) per Node	4
Number of Cores per Socket	10
Hyperthreading	enabled
Threads per Core	2
Number of Threads per Socket	20
Peak Frequency per Core	2.4 Ghz
Peak Performance per Core	9.6 GFlop/s
Peak Performance per Node	384 GFlop/s
L3 Cache per Socket (shared)	30 MB
L2 Cache per core	256 KB
L1 Cache per core	32 KB

The following figure shows the block diagram of a Fat Node. It comprises of 4 sockets, each of which contains 10 dual-threaded cores, giving 80 logical processors in total. Each socket is adjacent to a local memory controller and memory bus, which offers a separate path to main memory for that socket's processors. This is a Non-Uniform Memory Access (NUMA) organization, which guarantees uncontended and fast memory access when threads are spread across different sockets and their data are allocated in the local memory modules.

The figure depicts the CPU numbers that corresponds to each logical processor in a SuperMUC Fat Node, as seen by the operating system itself. These numbers can be used by the programmer to specify a desired affinity mapping at a low level.

Figure 6. SuperMUC fat node processor topology



2.2.4.2. Sandy Bridge-EP processor

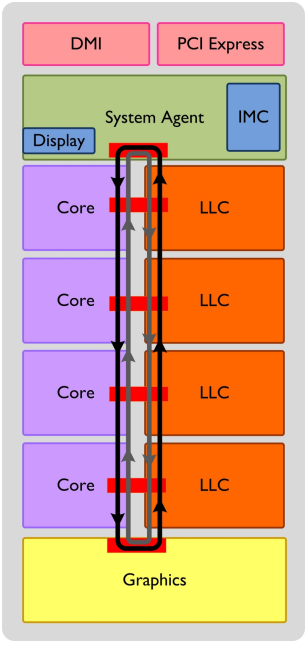
Sandy-Bridge is the codename for a new Intel microarchitecture, first released in products in 2011. It introduces Intel's first monolithic CPU and graphics solution for maximum performance.

The most important innovations of the new microarchitecture are listed below:

- *Advanced Vector Extensions (AVX)* 256-bit instruction set: This is an extension to the SSE 128-bit instruction set that provides new features, new instructions and a new coding scheme. The width of SIMD registers is extended from 128 to 256 bits, meaning that they can hold a vector of 4 double-precision floating point elements, or a vector of 8 single-precision floating point elements. It introduces also a three-operand SIMD instruction format, where the destination register is distinct from the two source operands and thus preserves both source operands (i.e., in the form c = a + b). Finally, AVX adds 12 new SIMD instructions.

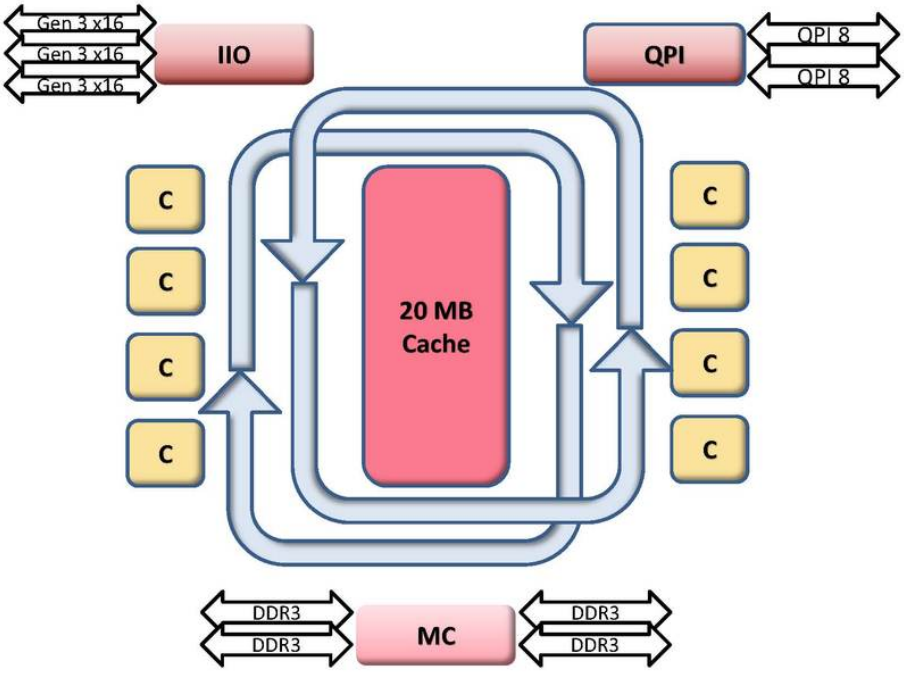
- *Scalable ring on-die interconnect*: Sandy-Bridge incorporates a new, ring-based interconnect between cores, graphics, last-level cache and System Agent domain. The ring is composed of 4 rings (32 bit bidirectional data ring, request ring, acknowledge ring and snoop ring) and is fully pipelined at core frequency. Access on the ring always picks the shortest path between the communication endpoints, thus minimizing latency. The Last Level Cache (LLC) in Sandy-Bridge is segmented so that each core has a portion (bank) attached closer to it. Data accesses on a bank that is "local" to a core are satisfied immediately without getting on the ring. When the core requests data residing on neighbouring banks, then the request goes out on the ring and it takes one or more hops (proportional to the number of cores) to fetch the data. Finally, the ring features distributed arbitration, sophisticated coherence protocol, and it scales to servers with large number of processors. The ring is shown in the following figure.

Figure 7. Sandy-Bridge-EP ring interconnect



- *Significant bandwidth improvements over prior generation in various uncore components*: Specifically, the LLC bandwidth was increased by roughly 800% (cache component), the on-die interconnect by 900% (ring component), the DDR3 memory bandwidth by 200% (integrated MC component), the socket-to-socket bandwidth by 250% (QPI component), and the PCIe bandwidth by 300% (IIO component). The aforementioned uncore components are depicted in the following picture.

Figure 8. Sandy-Bridge-EP uncore



- All the key microarchitectural features of Westmere were adopted by Sandy-Bridge, and in some cases improved. These include HTT, IMC, QPI, the Uncore, and other architectural features.

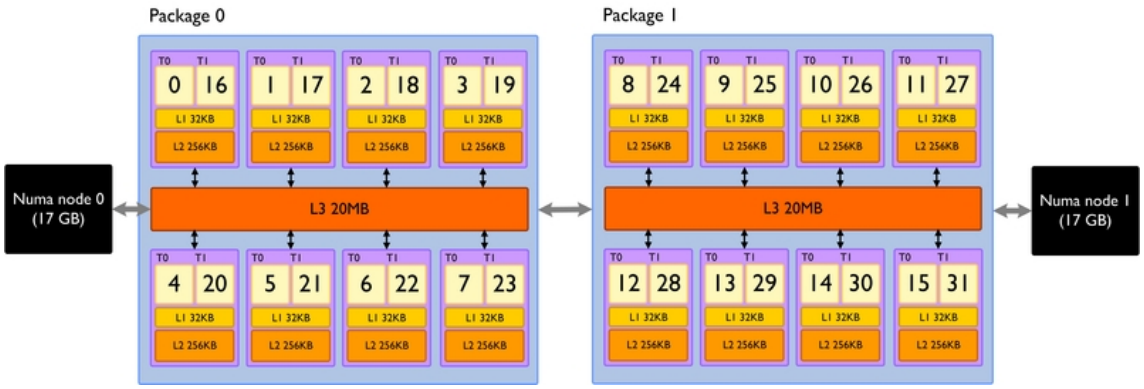
- Other new features:
  - Shared L3 cache includes the processor graphics (LGA 1155).
  - Two load/store operations per CPU cycle for each memory channel.
  - Decoded micro-operation cache ("DSP") that serves as a Level-0 cache for about 1500 micro-operations, and gives higher bandwidth, lower latency and less power.
  - Even larger buffers for improved out-of-order execution (Re-Order Buffer, Reservation Station) and a new Physical Register File.
  - Improved performance for transcendental mathematics, AES encryption (AES instruction set), and SHA-1 hashing.
  - Optimized Intel Turbo Boost technology.
  - First generation of Intel HD graphics.

Specifically for the SuperMUC Thin Nodes, the processor configuration is shown in the following table:

Number of Sockets (NUMA nodes) per Node	2
Number of Cores per Socket	8
Hyperthreading	enabled
Threads per Core	2
Number of Threads per Socket	16
Peak Frequency per Core	2.7 Ghz
Peak Performance per Core	21.6 Gflop/s
Peak Performance per Node	345.6 Gflop/s
L3 Cache per Socket (shared)	20 MB
L2 Cache per core	256 KB
L1 Cache per core	32 KB

The following figure depicts the CPU numbers that correspond to each logical processor in a SuperMUC Thin Node, as seen by the operating system itself.

Figure 9. SuperMUC thin node processor topology



2.2.5. Operating environment

2.2.5.1. Operating environment

All nodes of SuperMUC run SUSE Linux Enterprise Server (SLES) as a GNU/Linux based operating environment.

For batch queuing IBM Tivoli Workload Scheduler [http://www.lrz.de/services/compute/supermuc/loadleveler/] is installed.

2.2.6. Filesystems

2.2.6.1. Overview

File system	Environment Variable for Access	Purpose	Implementation Overall size Bandwidth	Backup and Snapshots	Intended Lifetime and Cleanup Strategy	Quota size (per project)
Shared Filesystem for SuperMIG and SuperMUC						
`/home/hpc`	`$HOME`	store the user's source, input data, and small and important result files. Globally accessible from login and compute nodes.	NAS-Filer 1.5 PB 10 GB/s	YES: backup to tape and snapshots	Project duration.	default: 100 GB per project [b]
Only on SuperMIG						
on SuperMIG: `/scratch` `/scratch_mpiio` [a]	`$SCRATCH` `$SCRATCH_MPIIO` legacy: `$OPT_TMP`	temporary huge files (restart files, files to be pre- / post-processed). Globally accessible from login and compute nodes.	NAS-Filer 200 TB ~800 MB/s	*NO*	*Automatic deletion after approx. 2 weeks.*	fixed: 25 TB per project
`/work` `/work_mpiio` [a]	`$WORK` `$WORK_MPIIO`	huge result files. Globally accessible from login and compute nodes.	NAS-Filer 250 TB, ~ 800 MB/s 1 TB	*NO*	Project duration (beware of technical problems, archive important data to tape)!	default: 1 TB per project [b]
Only on SuperMUC						
`/gpfs/scratch`	`$SCRATCH` legacy: `$OPT_TMP`	temporary huge files (restart files, files to be pre- / post-processed). Globally accessible from login and compute nodes.	GPFS 10 PB up to 200 GB/s (shared between scratch and work)	*NO*	*Automatic deletion after approx. 2 weeks.*	fixed: 25 TB per project
on SuperMUC:`/gpfs/work`	`$WORK`	huge result files. Globally accessible from login and compute nodes.	GPFS 10 PB up to 200 GB/s (shared between scratch and work)	*NO*	Project duration (beware of technical problems, archive important data to tape!)	default: 1 TB per project [b]

File system	Environment Variable for Access	Purpose	Implementation Overall size Bandwidth	Backup and Snapshots	Intended Lifetime and Cleanup Strategy	Quota size (per project)
Different on SuperMIG/MUC and Compute/Login Nodes						
various	`$TMPDIR`	temporary filesystem for system command use. different on login and compute nodes.	`$SCRATCH` `/$USER` on login nodes, `$SCRATCH` `/tmp` on compute nodes	*NO*	May be deleted after job end or logout.	
`/tmp`	*direct use of /tmp is strongly discouraged on compute nodes (may impact system usability)*					

[a] `/work` and `/work_mpiio` share the same physical disk space but are differently mounted. The variable `MP_SINGLE_THREAD` must be set to "no" (`export MP_SINGLE_THREAD=no`).

[b] Can be increased on request.

2.2.6.2. User's responsibility for saving important data

Having (parallel) filesystems of several hundreds of Terabytes (`/scratch` and `/work`), it is technically impossible (or too expensive) to backup these data automatically. Although the disks are protected by RAID mechanisms, other severe incidents might destroy the data. Therefore it is within the responsibility of the user to transfer data to more safe places (e.g. `$HOME`) and to archive them to tapes. Due to the long off-line times for dump and restoring of data, LRZ might not be able to recover data from any type of file outage/inconsistency of the *scratch or work* filesystems. The alias name `$WORK` and the intended storage period until the end of your project should not be misguided as an indication for the data safeness.

There is no automatic backup for `/scratch` and `/work`. Beside automatic deletion, severe technical problems might destroy your data. It is your obligation to copy, transfer or archive the files you want to keep!

2.2.6.3. Limitations and advantages of the GPFS file system

On SuperMUC, a single GPFS file space has been established, within which the `$SCRATCH` and `$WORK` portions described in the above table live. This file system is tuned for *high bandwidth*, but it is not optimal for handling large quantities of *small* files located in a single directory with parallel accesses. In particular, generating more than ca. 1000 files per directory at approximately the same time from either a parallel program or from simultaneously running jobs will probably cause your application(s) to experience I/O errors (due to timeouts) and crashes. If you require this usage pattern, please generate a directory hierarchy with at most a few hundred files per subdirectory.

2.2.6.3.1. Optimizing concurrent access to files and directories

The general rule is to avoid having hundreds or thousands of tasks trying to modify the same directory at the same time with certain operations.

Doing lookups, creation or deletion of different files in the directory works fine with the Fine Grain Directory Locking (FGDL) protocols that have been the default for some time now. But the operations: `readdir (ls)`, `rename (mv)`, and `mkdir` all in the same directory, or all tasks trying to create the same file, will serialize all the tasks and become a very big bottleneck when there are more than a handful of nodes. The nodes will thrash the directory blocks back and forth via disk operations as they try to share lookups and then need exclusive locks to modify the directory. So these operations are done best by designating one task to do them before all the other tasks start working with the file objects.

In the case of all tasks opening a common file, instead of every task doing a create (or open with the create flag) the code using MPI should do something like:

```
!# serial creation
```

```
barrier
if (task==0) then
     do i=0,nprocs-1
        create file(i)  // with optional truncate option
     enddo
endif
!# all files created now
barrier
!#parallel usage
if (task !=0) then
   open file(i)
   write(...)
endif
```

The tasks can then proceed to modifying their own portions of the common file, with best results if their regions do not overlap on a granularity smaller than the GPFS blocksize. For fine grain updates that are smaller than the blocksize, the MPI-IO package is advised since it will use MPI to ship around the small updates to nodes that manage different regions of the file.

2.2.6.4. Temporary filesystems $SCRATCH

Please use the environment variable `$SCRATCH` to access the temporary filesystems. This variable points to the location where the underlying file system will deliver optimal I/O-Performance. Do not use `/tmp` for storing temporary files! (The filesystem where `/tmp` resides on is very small and slow. Files will be regularly deleted by sysadmin).

2.2.6.5. Coping with high watermark deletion in $SCRATCH

The high watermark deletion mechanism may *remove files which are only a few days old* if the file system is used heavily. In order to cope with this situation, please note:

- The normal `tar -x` command preserves the modification time of the original file and not the time when the archive has been unpacked. Therefore, files which have been unpacked from an older archive are one of the first candidates to be deleted. To prevent this, use `tar -xm` to unpack your files, which will give them the actual date.
- Please use the TSM system to archive/retrieve files from/to `$SCRATCH` to/from the tape archive.
- Please *always* use `$WORK` or `$SCRATCH` for files which are considerably larger than 1 GB.
- Please remove any files which are not needed any more as soon as possible. The high watermark deletion procedure is then less likely to be triggered.
- More information about the filling of the file systems and about the oldest files will be made available on a web site in the near future.

2.2.6.6. Snapshots, backup, archiving, and restoring

For all files in `$HOME` backup copies are kept and made available in the special (read-only) subdirectory `$HOME/.snapshot/`. Please note that the `.snapshot` directories are not visible by a simple `ls` command. A file can be restored by simply copying the file from the appropriate snapshot directory to its original or other location.

For using the TSM tape archiving and backup it is necessary to login to the archiving nodes `super-muc-tsm.lrz.de`. The regular login nodes do not support TSM usage. Conversely, the archiving nodes should not be used for any purpose than TSM data handling.

For details see the HPC Backup and Archiving [http://www.lrz.de/services/compute/backup/] document on how to handle backups and the TSM tape system.

2.2.7. Transfer of files between SuperMIG and SuperMUC

While the HOME directories are transparently shared between these two systems, access to data on `$WORK` or `$SCRATCH` goes to different physical storage on migration system and SuperMUC, respectively. In order to enable migration of data from migration system to SuperMUC, read-only mounts of the migration system file spaces are available on the SuperMUC login nodes (but not the compute nodes) under the name `$WORK_MIGRATION` and `$SCRATCH_MIGRATION`. This is a one-way only facility.

```
cp $WORK_MIGRATION/myfile $WORK/myfile
```

The other way around e.g., copying files directly from SuperMUC to SuperMIG is not possible, since the filesystems are mounted read-only. However, `scp` between SuperMUC and SuperMIG is available from SuperMUC (however not from SuperMIG).

```
scp $WORK/myfile supzero.lrz.de:$WORK_MIGRATION/myfile
```

We expect that the migration system will be integrated into the main system in 2013; at that point the GPFS will also become available there (and the NAS filespaces for `$SCRATCH` and `$WORK` will be removed soon afterwards).

2.2.8. Transferring files from/to other systems

Due to our security regulations, transferring files typically will *only* work if the IP address of the *remote* system has been entered into LRZ routing database. Additionally, it may be necessary to specify specific ports to be opened for this IP address. Please apply for such an entry via an update in the project proposal [http://www.lrz.de/services/compute/supermuc/projectproposal/] form.

- *secure copy: scp*

  For secure file transfer, the command scp can be used.

  However due to security considerations, transfer of files is only possible from the remote site to SuperMUC.

  This method is most straightforward but time-consuming for large files.

- *GridFTP*

  GridFTP is the way to go if you want to transfer files, big or small, to or from an LRZ machine. LRZ offers three kinds of interfaces to meet your transfer needs:

  - a graphical point-and-click interface through Globus Online [http://www.grid.lrz.de/en/mware/globus/client/go_with_lrz_resources.html],
  - a more traditional command line interface through gtransfer [http://www.grid.lrz.de/en/mware/globus/client/gtransfer.html],
  - the command line client included by default in the Globus toolkit, globus-url-copy [http://www.grid.lrz.de/en/mware/globus/client/guc.html], especially suitable for advanced users.

  All of them are very easy to use - you select the style that best fits your personal taste.

2.2.9. Quota

2.2.9.1. General information on quota

To see your quota use:

```
module load lrztools
budget_and_quota
```

2.2.9.2. Quota/volume limits in $HOME and $WORK/$SCRATCH on SuperMIG

The storage for $HOME or $WORK is limited. Each project is assigned a separate volume on the NAS-Filer which will be mounted to /home/hpc/<project_name> and will contain the home-directories of the project's users. The maximum size of the volumes is limited. The command to get information about your quota and the disk space used is:

```
sdf $HOME  or  sdf -g $HOME  for output in GByte
sdf $WORK
sdf $SCRATCH
```

The disk space in $HOME is not only occupied by your current data but also by snapshots ("backup copies") from the last 10 days. Typically your file space consists of 150 GB quota + an additional 150 GB for the snapshots. If you change and delete lots of files in your home directory during 10 days so that the amount of changes is larger than 150 GB in 10 days the additional space is not sufficient and snapshots will also take up space from the "real" quota until they are automatically deleted.

It might help if you do not place any temporary files in your home directory ($HOME) but use the large parallel project filesystem ($WORK) or the parallel temporary file system $SCRATCH.

2.2.9.3. Quota limits in $WORK on SuperMUC

The storage is limited on the level of projects. Each project is a file set. Individual user quotas or sub-quotas are not possible. To directly see the quotas use:

```
mlsquota -j work_projectID fs1
```

where projectID is your projectID/Unixgroup ID.

3. System access

3.1. Application for an account / project

Persons who do not have an SuperMUC account need to apply for a project and need to undergo a review process. Please use the Online Proposal Form [http://www.lrz.de/services/compute/supermuc/projectproposal/] and apply for resources on SuperMUC. Except for test project requests with limited resources, it will take approximately two months to process a project request. Questions concerning the usage should be directed to the LRZ HPC support team [http://www.lrz.de/services/compute/support/], preferably via the Servicedesk Portal [https://servicedesk.lrz.de/?service=Hochleistungsrechnen%20und%20Grid-Supercomputing%20%28HLRB%29?=en] .

3.2. Contact, support and servicedesk

For questions from special application areas, you may also directly contact members of the support team listed in the table below. However, we always recommend to submit an incident via our Servicedesk-Portal, via the simple Submission Site or by calling the Servicedesk. This will guarantee that your request will be processed on short order and all actions can be tracked.

Please always consult http://www.lrz.de/aktuell/ [http://www.lrz.de/aktuell/] before reporting an incident.

3.2.1. Selfservice portal

Allows you to submit incidents, view old incidents and add additional information. Furthermore you can place orders according to the LRZ service catalog. Important: This portal only works if you use MS Internet Explorer or Mozilla Firefox with enabled javascript.

Link to the Portal. [https://servicedesk.lrz.de/selfservice/Default.aspx?service=Hochleistungsrechnen%20und%20Grid]

3.2.2. Simple submission site

Allows you to submit incidents with or without your LRZ username. The portal supports all browsers and mobile browsers.

Link to the Portal. [https://servicedesk.lrz.de/plainsubmit/?service=Hochleistungsrechnen%20und%20Grid]

3.2.3. Servicedesk phone

Phone: +49 89 35831-8800

3.2.4. Persons

Phone-Nr.: +49 89 35831-Ext

3.2.4.1. High Performance Computing

Name	Phone-Ext	E-Mail	Area of support
Dr. Matthias Brehm	-8773	`Brehm(at)lrz(dot)de`	Group Leader, application support Fortran, MPI, Optimization, tools, user affairs. Problem and incident management.
Dr. Reinhold Bader	-8825	`Bader(at)lrz(dot)de`	Group Leader HPC: Operation, system software, tools, performance libraries, Fortran. Change and configuration management of application software. Problem and incident management.
Dr. Momme Allalen	-8816	`Allalen(at)lrz(dot)de`	Application support: chemistry; optimization and scaling of HPC codes.
Dr. Alexander Block	-8817	`Block(at)lrz(dot)de`	Application support: fluid dynamics and FE, optimization and scaling of HPC codes.
Dr. Nicolay J. Hammer	-8872	`Hammer(at)lrz(dot)de`	Intel Compilers, MKL, etc.
Dr. Ferdinand Jamitzky	-8727	`Jamitzky(at)lrz(dot)de`	Java, Scripting, Problem Management , Life Sciences.
Dr. Helmut Satzger	-8864	`Satzger(at)lrz(dot)de`	Life Sciences.
Dr. Volker Weinberg	-8863	`Weinberg(at)lrz(dot)de`	New programming languages and paradigms.
Carmen Navarrete	-8869	`Navarrete(at)lrz(dot)de`	C, C++ , tools.
Dr. Wolfram Hesse	-8845	`Hesse(at)lrz(dot)de`	Accounting, Monitoring.
Carla Guillen	-8867	`Guillen(at)lrz(dot)de`	User administration and project management; HPC Statistics. Eclipse.

Name	Phone-Ext	E-Mail	Area of support
Cornelia Wendler	-8857	Wendler(at)lrz(dot)de	User administration and project management; HPC Statistics.

3.2.4.2. Remote visualization and virtual reality

Name	Phone-Ext	E-Mail	Area of support
Jutta Dreer	-8741	Dreer(at)lrz(dot)de	Software and user support for virtual reality environments.
Dr. Ferdinand Jamitzky	-8727	Jamitzky(at)lrz(dot)de	Software and user support for remote visualization support.
Dr. Helmut Satzger	-8764	Satzger(at)lrz(dot)de	System Configuration. Software and user support for remote visualization support.

Please see also the Remote Visualization [http://www.lrz.de/services/compute/visualisation/] and Virtual Reality [http://www.lrz.de/services/v2c_de/] webpages.

3.2.4.3. Grid computing

Please consult the Grid Team's support pages. [http://www.grid.lrz.de/en/team.html]

3.3. Login

3.3.1. Login to SuperMUC/SuperMIG

Before you can login to SuperMUC you have to set your own password. Login with your userID and the start password (that we deliver to your project manager) into the ID Portal [https://idportal.lrz.de/r/entry.pl?Sprache=en] using your account and the start password that we have delivered to your project manager.

SuperMUC/SuperMIG uses front-end nodes or login nodes for interactive access and the submission of batch jobs.

The front-end nodes have an identical environment, but multiple sessions of one user may reside on different nodes which must be taken into account e.g., when killing processes.

Two mechanisms are provided for logging in to the system; both incorporate security features to prevent appropriation of sensitive information by a third party.

Note: The SuperMUC firewall permits only *incoming* SSH-connections.

3.3.2. Login with secure shell

Access via SSH (Secure Shell) is described in detail in the LRZ document about SSH. [http://www.lrz.de/services/compute/ssh/]

If you want access without every time typing in your password but with public/private SSH-keys please note:

LRZ Security Policies demand that the user's private SSH keys are locked and guarded with a non-empty passphrase, therefore it is not allowed to use an empty passphrase during the private key generation . We consider an empty passphrase as a violation of our security policies. Users disregarding this policy will be barred from further usage of LRZ systems.

From the UNIX command line on one's own workstation the login to an LRZ account xxyyyyzz is performed via:

System part	Login	Architecture	Number of login nodes behind the round-robin address
SuperMUC / thin nodes	`ssh -X xxyyyyzz@supermuc.lrz.de`	Intel Sandy Bridge EP	5
Migration system, SuperMUC / fat nodes	`ssh -X xxyyyyzz@supzero.lrz.de`	Intel Westmere EX	2
Nodes on SuperMUC with connection to the archive system	`ssh -X xxyyyyzz@supermuc-tsm.lrz.de`	Intel Sandy Bridge EP	2

Please also bear in mind the following *notes*:

- The IP address of your front-end machine must be associated with a valid DNS entry, and must be known to us, otherwise your SSH request will *not be routed*. Additional entries or changes can be submitted via a modification request in the using Online Proposal Form [http://www.lrz.de/services/compute/supermuc/projectproposal/].
- The LRZ domain name is mandatory if accessing from outside the Munich Scientific Network.
- The `-X` option is responsible for tunneling of the X11 protocol, it may be omitted if no X11 clients are required.

The Secure Shell RSA public key are given in the following link. Please add these to `~/.ssh/known_hosts` on your own local machine before logging in for the first time.

- SSH Public Keys [http://www.lrz.de/services/compute/supermuc/access_and_login/supermuc_ssh-rsa.txt]

Note: ssh from SuperMUC to the outside world is disabled.

3.3.3. Login via grid services using GSI-SSH

An alternative way of accessing the SuperMUC is to use GSI-SSH, which is a component of the Globus toolkit and provides

- terminal access to your account,
- a single sign-on environment (no password required to access other machines),
- easy access to a number of additional functionalities, including secure and parallel file transfer.

The prerequisites for using it are

- a Grid certificate installed on your machine and acknowledged by LRZ, as described on the LRZ Grid Portal [http://www.grid.lrz.de/en/gacc/certreq.html]. Please note that TUM, LMU, and LRZ members can use the new and easy short lived credential service (SLCS) [https://slcs.pca.dfn.de/gridshib-ca/] of the DFN as an alternative: it allows you to immediately obtain a certificate for Grid usage,
- an installation of a GSI-SSH client on your own workstation, either the command line tool *gsissh* or the multi platform Java tool Gsissh-Term, as described on the LRZ Grid Portal [http://www.grid.lrz.de/en/mware/globus/client/gsissh_access.html].

3.4. Password policies

Passwords must be changed at least once in 12 months. We are aware that this measure imposes some overhead on users, but believe that it is necessary on security reasons, having implemented it based on guidelines of BSI (federal agency for information security) and the IT security standard ISO/IEC 27001. You are able to determine the actual invalidation date for your password by logging into the ID portal [https://idportal.lrz.de] and selecting

the menu item "Person -> view" or "Account -> view authorizations". In order to prevent being surprised by a password becoming invalid, you will be notified of the need to change your password via e-mail. Even if you miss the deadline for the password update, this only implies a temporary suspension of your account - you will still be able to log in to the ID portal and make the password change.

For more details see the complete (German) text of the authentication regulations [http://www.lrz.de/wir/regelwerk/passwortrichtlinien.pdf].

Changing the password is also necessary after it has been newly issued, or reset to a starting value by a project manager or LRZ staff. This assures that actual authentication is done with a password known only to the account owner.

3.5. Changing password or login-shell, viewing accounting data

The direct use of the `passwd` and `chsh` commands to change passwords and login shells, respectively, has been disabled.

Please use the LRZ ID portal [https://idportal.lrz-muenchen.de/] instead:

- Log in to the web interface using your account and password.
- To toggle between English and German use the little flags.
- For changing your password, select the entry *"Self Services/modify password"*. In the main window you are then prompted for your *old* password once and for the *new* password (needs to have between 6 and 20 characters) twice.
- For changing your login shell select the entry *"Self Services/change login shell"*. For the platform *"SuperMUC"* select the new login shell from the drop-down menu.
- The ID portal also offers functionality to view your user account data. Please contact us if you need changes.

3.6. Budget and quotas

CPU time budget and file system quotas are displayed at login or at the start of a batch job.

However you can query them with the following commands:

```
module load lrztools
budget_and_quota
```

3.7. Moving data from/to SuperMUC

FTP access to the high performance systems from outside is disabled for security reasons, you have to use scp, sftp or GridFTP.

Note: Transfers of files with scp or sftp from SuperMUC to the outside world can only be initiated from the outside, e.g. you cannot copy files from SuperMUC to the outside, but you can fetch files from SuperMUC from the outside.

3.7.1. SCP

Example:

```
scp localfile UserID@supzero.lrz.de:remotefile
scp  UserID@supzero.lrz.de:remotefile localfile
```

3.7.2. SFTP

SSH File Transfer Protocol (also Secret File Transfer Protocol, Secure FTP, or SFTP) is a network protocol that provides file access, file transfer, and file management functionality over any reliable data stream.

Example:

```
sftp UserID@supzero.lrz.de
put localfile remotefile
get remotefile localfile
```

3.7.3. GridFTP

The only way to initiate transfers from SuperMUC to the outside is using GridFTP. However, this requires a certificate.

For details see: http://www.grid.lrz.de/en/mware/globus/client/gridftpclient.html

3.8. Access to SVN server

The SuperMUC firewall permits only incoming SSH-connections. You can use port forwarding to establish a connection between the subversion server and SuperMUC, i.e., you may use one of the the following procedures.

You will be prompted for your SuperMUC password (or your ssh passphrase). If you are unlucky the port selected by you (e.g. 10022) is already used by someone else - in this case you will see an error message printed out **in advance of** the motd; you then need to change your port to a different value. You might need to delete the localhost entry from `~/.ssh/known_hosts` if ssh complains about the host key. If you need to change ssh ports, you will probably also need to invoke "`svn switch --relocate ...`" on your SVN sandboxes because the port number will be encoded in the stored location.

3.8.1. Using SVN with a https svn server

1. To establish the port forwarding for the **ssl/tls** port issue the following command to connect from your workstation you normally use to SSH to the system SuperMUC (or SuperMIG accordingly):

```
ssh -l <LoginName> -R <arbitraryPortNumber>:<svnServer>:443
                                          supermuc.lrz.de
Example:
ssh -l hk00xyz -R 10443:pmviewer.svn.sourceforge.net:443 supzero.lrz.de
```

2. After successful login to supermuc you may then access your repository via:

```
svn <svnCommand> https://<remoteLoginName>@localhost:
                    <ForwardedPortNumber>/<svnDirectoryPath>
Example:
svn list https://mySVNUser@localhost:10443/svnroot/pmviewer
svn co   https://mySVNUser@localhost:10443/svnroot/pmviewer pmviewer
```

3.8.2. Using SVN+SSH repository access

1. To establish the port forwarding for the **ssh** port issue the following command to connect from your workstation you normally use to SSH to the system SuperMUC (or SuperMIG accordingly):

```
ssh -l <LoginName> -R <arbitraryPortNumber>:<machine-withSVNrepo.>:22
                                                    supermuc.lrz.de
```

```
Example:
ssh -l hk00xyz -R 10022:mySVNmachine.myhost.de:22 supermuc.lrz.de
Repository:
mySVNmachine.myhost.de:/my/svn/repo
```

2. After successful login to supermuc you have to set up a new protocol in your `~/.subversion/config` file. Therefore you enter the following last line to the tunnel section in the config file:

```
[tunnels]
### Configure svn protocol tunnel schemes here.  By default, only
### the 'ssh' scheme is defined.  You can define other schemes to
### be used with 'svn+scheme://hostname/path' URLs.  A scheme
### ...
myssh = ssh -p 10022
```

Now you may use the **svn+ssh** command as usual, with the exception that the newly defined **myssh** protocol is used instead of the standard **ssh** protocol:

```
svn <svnCommand> svn+myssh://<remoteLoginName>@localhost/<svnDirectoryPath>

Example:
svn list svn+myssh://mySVNUser@localhost/my/svn/repo
svn co   svn+myssh://mySVNUser@localhost/my/svn/repo
```

3.9. Supported services to and from SuperMUC

3.9.1. Incoming (to SuperMUC)

- ssh
- GRAM
- gsissh
- gridftp
- Ephemeral Ports for gridftp

3.9.2. Outgoing (from SuperMUC)

- GRAM
- gridftp
- myproxy
- Ephemeral Ports for gridftp

If you need information about the port, please contact LRZ.

4. Production environment

4.1. Accounting, budgets and quota

CPU time budget and file system quotas are dislayed at login or at the start of a batch job.

However you can query them with the following commands:

```
module load lrztools
budget_and_quota
```

4.2. Software packages

For an overview of software packages that are available on LRZ systems, see: http://www.lrz.de/services/software/.

4.3. Module environment

LRZ uses the *environment module* approach to manage the user environment for different software, library or compiler versions. The distinct advantage of the modules approach is that the user is no longer required to explicitly specify paths for different executable versions, and try to keep the MANPATH and related environment variables coordinated. With the modules approach, users simply "load" and "unload" modules to control their environment. Type

```
module avail
```

to list all the modules which are available to be loaded. Notice that most of them indicate associated version numbers. Modules make it easy to switch between versions of a package. Specifying a module name without a version number will select the default production version.

- More information on the modules package [http://www.lrz.de/services/software/utilities/modules/]

Many modules contain the URL of the LRZ documentation for the specific software package. Type

```
module show <modulename>
```

and look for the variable which ends with `_WWW`.

4.4. LRZ tools

To get access to the LRZ Tools insert the following command

```
module load lrztools
```

After that the following commands are available

Comand	**Purpose, details and usage**
Information	
`budget_and_quota`	Displays the CPU time budget and the file system quotas.
Placement of processes and threads	
`cpuset_query`	Returns information about the topology, number of core and CPU on which a process can run.
`get_cpuset_mask`	Returns a string of 0's and 1's indicating on which (logical) CPUs a process can run.
`where_do_i_run`	Returns the CPU-ID on which the command was run.
`placementtest-mpi.ibm`	Returns information how processes and threads are placed on nodes and CPUs. Example:

Comand	Purpose, details and usage

`placementtest-mpi.intel`
 	
`placementtest-omp`
	`MP_POCS=5`
	`MP_NODES=2`
	`OMP_NUM_THREADS=10`
	`MP_TASK_AFFINITY=`
	`core:$OMP_NUM_THREADS`
	`mpiexec -n 5`
	`placement-test.ibm`

Batchjobs	
`lljobs`	list jobs, per default prints jobs of user of current and past month, for details see: `lljobs -h`
`llu`	Shortcut to show only own jobs in the queue.
`llx`	A better and more comprehensive overview of jobs in the queues, including various sort options. For details see: `llx -h`
`llrun`	Submit or interactively run a parallel executable, for details see: `llrun -h`

`autonomous`	Embarrassingly parallel commands or jobs can be run as one LoadLeveler job; usage: `autonomous [<options>] [-f] file`, for details see: `autonomous -h`, example: copy 32 files in parallel to one directory
	
	`cat >cmdfile <<EOD`
	`cp file1 DIR`
	`cp file2 DIR`
	`...`
	`cp file32 DIR`
	`EOD`
	
	`autonomous cmdfile`

`myjobs`	Display all job owned by user in the queues. Can be used as input to build other commands. E.g. `llcancel` `` `myjobs` `` will cancel all job owned by user.
`get_cmd_out`	Sort output produced with option `MP_LABELIO`.
Programming Environment	
`I_MPI`	Displays sorted list of the current settings of the Intel MPI Environment.
`MP`	Displays sorted list of the current settings of the IBM Parallel Environment.

4.5. LRZ library

To load the LRZ library use

```
module load lrztools
```

It contains useful subroutines and functions. Compile with:

```
mpicc/mpif90 ... $LIBLRZ_INC ... $LIBLRZ_LIB
```

C function	Fortran subroutine	Purpose, details and usage
`int getpid (void)`	`INTEGER GETPID`	Returns the process ID.
`int gettid (void)`	`INTEGER GETTID`	Returns the thread ID.
`int where_do_i_run(void);`	`INTEGER WHERE_DO_I_RUN()`	Returns the physical CPU ID where the task/thread is running.
`double dwalltime(), double dcputime()`	`REAL(KIND=8) DWALLTIME(), REAL(KIND=8) DCPUTIME()`	Returns the wallclock time/cputime spent between first and current call to the routine.
`void place_task_(int cpu[],int *n);`	`INTEGER CPU(N) PLACE_TASK(CPU,N)`	Sets the mask, that the current task will run on the physical CPUs contained in the array CPU.
`void(place_all_tasks(int *debug)`	`LOGICAL DEBUG PLACE_ALL_TASKS(DEBUG)`	Places the tasks and threads on particular CPUs, whether by default algorithm or by using the environment variable CPU_MAPPING.
`void place_task_(int cpu[],int *n);`	`INTEGER CPU(N) PLACE_TASK(CPU,N)`	Sets the mask, that the current task will run on the physical CPUs contained in the array CPU.
`void placementinfo()`	`PLACEMENTINFO()`	Outputs information about the placement of tasks and threads.

4.6. Node allocation policy

- Only complete nodes are provided to a given job for dedicated use.
- Accounting is performed by using: `AllocatedNodes*Walltime*(number_of_core_in_node)`.
- Try to not waste cores of a node, i.e. try to use all cores. However, in some special cases, e.g. large memory requirements, etc. , you may not be able to use all cores.
- `TASKS_PER_NODE` must be less or equal 40 for SuperMIG.
- `TASKS_PER_NODE` must be less or equal 16 for SuperMUC (if using SMT/HyperThreading it must be less or equal 32).

4.6.1. Jobs size: totaltasks, number of nodes and tasks per node

Typically there are two ways of specifying the job size (e.g. for 40 cores per node):

- *Total tasks and number of cores*
  - This is the more general case.
  - If your number of tasks is not evenly dividable by number of nodes.
  - You must compute the number of node by yourself (i.e., ceil(number of tasks/40)).
  - LoadLeveler decides how to distrubute the tasks to nodes.
- *Tasks per node and number of nodes*
  - Use this if your number of tasks is 40 (or a bit less),
  - if you can run with various number of nodes by specify min and max.

Hints:

- Do not mix both ways!
- Always specify number of nodes!
- Always specify the number of islands on SuperMUC for jobclass "large".
- Do not waste cores of a node (remember: 16 or 40 cores per node)!

4.7. Interactive parallel jobs

4.7.1. IBM MPI

Parallel executables which are built with IBM's MPI can be run interactively by invoking them with one of the following methods

- `poe ./executable -procs <nnn> -nodes <NNN>`
- `mpiexec -n <nnn> ./executable -nodes <NNN>`
- `./executable -procs <nnn> -nodes <NNN>`
- `MP_PROCS=nnn MP_NODES=NNN poe ./executable`
- `MP_PROCS=nnn MP_NODES=NNN mpiexec ./executable`
- `MP_PROCS=nnn MP_NODES=NNN ./executable`

The executable is not executed on the frontend nodes but on the compute nodes. The resource manager of LoadLeveler is used to allocate free nodes. Therefore it is equivalent to submitting a Job Command File to LoadLeveler via `llsubmit`.

The LoadLeveler keyword part can also be used to allocate nodes

```
cat > LL_FILE <<EOD
#@ job_type = parallel
#@ node = 2
#@ total_tasks = 78 ## other example #@ tasks_per_node= 39
#@ queue
EOD
poe ./executable -rmfile LL_FILE
```

4.7.2. Intel MPI

You can invoke executables compiled with Intel MPI via `poe`. However, the run runtime libraries of IBM MPI are then used. Problems, particularly with hybrid (OpenMP-MPI) execution may occur. We recommend to write an LoadLeveler Job Command File and submit it to the class "test".

4.7.3. Limited resources for job class "test"

POE jobs take their resources from the nodes of the job class "test". If you get the the following message:

```
ERROR: 0031-165 Job has been cancelled, killed, or schedd or
resource is unavailable. Not enough resources to start now. Global
MAX_TOP_DOGS limit of 1 reached.
```

then all nodes are busy. You have to wait and try again, or you have to submit your program to the test or general queue as a batchjob. You can use the examples given below by just replacing *class=general* by *class=test*.

If you use less than 40 cores, set "`MP_NODES=1`" to save resources for your co-workers, or use llrun, which makes in most cases the right decision for node and core allocation

4.8. Batch-jobs with LoadLeveler

The login node is intended only for editing and compiling your parallel programs. Interactive usage of "`poe/mpirun`" is not allowed. To run test or production jobs, submit them to the LoadLeveler batch system, which will find and allocate the resources required for your job (i.e., the compute nodes to run your job on).

The most important Loadleveler commands are:

llsubmit	Submit a job script for execution.
llq	Check the status of your job(s).
llhold	Place a hold on a job.
llcancel	Cancel a job.
llclass	Query information about job classes.

The `-H` flag provides extended help information.

Build your job command file `job.cmd` by using a text editor to create a script file.

```
# This job command file is called job.cmd
#@ input = job.in
#@ output = job.out
#@ error = job.err
#@ job_type = parallel
#@ class = general
#@ node = 8
#@ total_tasks=128
#@ network.MPI = sn_all,not_shared,us
#@ ... other LoadLeveler keywords (see below)
#@ queue
echo "JOB is run"
```

To submit the job command file that you created in step 1, use the llsubmit command:

```
llsubmit job.cmd
```

LoadLeveler responds by issuing a message similar to:

```
submit: The job "supermuc.22" has been submitted.
```

Where `supermuc` is the name of the machine to which the job was submitted and 22 is the job identifier (ID). To display the status of the job you just submitted, use the `llq` command. This command returns information about all jobs in the LoadLeveler queue:

```
llq supermuc.22
```

To place a temporary hold on a job in a queue, use the `llhold` command. This command only takes effect if jobs are in the Idle or NotQueued state:

```
llhold supermuc.22
```

To release the hold, use the `llhold` command with the `-r` option:

```
llhold -r supermuc.22
```

To cancel a job, use the `llcancel` command:

```
llcancel supermuc.22
```

4.9. Job command file

A Job Command File describes the job to be submitted to the LoadLeveler Job Manager using the llsubmit command. It can contain multiple steps, each designated by the #@queue statement. Lines starting with '#@' are statements that are interpreted by LoadLeveler's parser. The job command file itself can be the script to be executed by each step of the job.

Note that the llsubmit command itself does not form the job using command line arguments.

4.10. Examples for job command files for SuperMIG (Fat Nodes)

4.10.1. Parallel MPI job (IBM MPI)

```
#!/bin/bash
#
#@ job_type = parallel
#@ class = general
#@ node = 4
#@ total_tasks=156
## other example
##@ tasks_per_node = 39
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ job_name = mytest
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
mpiexec -n 156 ./myprog.exe
# other example
# poe ./myprog.exe
```

4.10.2. Parallel MPI job (Intel MPI)

```
#!/bin/bash
# DO NOT USE environment = COPY_ALL
#@ job_type = MPICH
#@ class = general
#@ node = 4
#@ total_tasks=156
## other example
```

```
##@ tasks_per_node = 39
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
#setup of environment
module unload mpi.ibm
module load mpi.intel
mpiexec -n 156 ./myprog.exe
```

4.10.3. Hybrid MPI job (IBM MPI)

```
#!/bin/bash
#
#@ job_type = parallel
#@ class = general
#@ node = 4
#@ total_tasks=12
## other example
##@ tasks_per_node = 3
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ job_name = mytest
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
export MP_SINGLE_THREAD=no
export OMP_NUM_THREADS=10
# Pinning
export MP_TASK_AFFINITY=core:$OMP_NUM_THREADS
mpiexec -n 12./myprog.exe
# other example
# poe ./myprog.exe
```

4.10.4. Hybrid MPI/OpenMP job (Intel MPI)

```
#!/bin/bash
# DO NOT USE environment = COPY_ALL
#@ job_type = MPICH
#@ class = general
#@ node = 4
#@ total_tasks=12
## other example
```

```
##@ tasks_per_node = 3
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ job_name = mytest
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
#setup of environment
module unload mpi.ibm
module load mpi.intel
export OMP_NUM_THREADS=10
#optional:
#module load mpi_pinning/hybrid_blocked
mpiexec -n 12 ./myprog.exe
```

4.10.5. Pure OpenMP job (on fat node island)

```
#!/bin/bash
#@ wall_clock_limit = 01:20:00,01:19:30
## with softlimit
#@ job_name = mytest
#@ job_type = parallel
#@ class = general
#@ node = 1
#@ total_tasks = 1
## OR
#@ tasks_per_node = 1
#@ node_usage = not_shared
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh

export OMP_NUM_THREADS=40
export KMP_AFFINITY="granularity=core,compact,1"
./myprog.exe
```

4.11. Examples for job command files for SuperMUC (Thin Nodes)

4.11.1. Parallel MPI Job (IBM MPI)

```
#!/bin/bash
##
```

```
## optional: energy policy tags
##
#
##@ job_type = parallel
#@ class = general
#@ node = 200 ###@ island_count= not needed for #### class general
#@ total_tasks= 3200 ## other example
##@ tasks_per_node = 16
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ job_name = mytest
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
poe ./myprog.exe
```

4.11.2. Parallel MPI Job (Intel MPI)

```
#!/bin/bash
##
## optional: energy policy tags
##
# DO NOT USE environment = COPY_ALL
#@ job_type = MPICH
#@ class = large
#@ node = 1000 ### schedule the job to 2 to 4 islands #@ island_count=2,4
#@ total_tasks= 16000 ## other example ##@ tasks_per_node = 16
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ job_name = mytest
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
#setup of environment
module unload mpi.ibm
module load mpi.intel
mpiexec -n 16000 ./myprog.exe
```

4.11.3. Hybrid MPI Job (IBM MPI)

```
#!/bin/bash
##
```

```
## optional: energy policy tags
##@ job_type = parallel
#@ class = large
#@ node = 1000
#@ island_count=2 #@ total_tasks=16000
### other example
##@ tasks_per_node = 16
##@ island_count = 1,18
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ job_name = mytest
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
export MP_SINGLE_THREAD=no
export OMP_NUM_THREADS=2
# Pinning
export MP_TASK_AFFINITY=core:$OMP_NUM_THREADS
mpiexec -n 16000./myprog.exe
```

4.11.4. Hybrid MPI/OpenMP job (Intel MPI)

```
#!/bin/bash
##
## optional: energy policy tags
##
# DO NOT USE environment = COPY_ALL
#@ job_type = MPICH
#@ class = large
#@ node = 1000 #@ island_count=2,4
#@ total_tasks=16000
### other example
##@ tasks_per_node = 16
##@ island_count = 1,18
#@ wall_clock_limit = 1:20:30
##                    1 h 20 min 30 secs
#@ job_name = mytest
#@ network.MPI = sn_all,not_shared,us
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
#setup of environment
module unload mpi.ibm
module load mpi.intel
```

```
export OMP_NUM_THREADS=2
#optional:
#module load mpi_pinning/hybrid_blocked
mpiexec -n 16000 ./myprog.exe
```

4.11.5. Pure OpenMP job (on thin node island)

```
#!/bin/bash
##
## optional: energy policy tags
##
#@ wall_clock_limit = 01:20:00,01:19:30
## with softlimit
#@ job_name = mytest
#@ job_type = parallel
#@ class = general
#@ node = 1
#@ total_tasks = 1
## OR #@ tasks_per_node = 1
#@ node_usage = not_shared
#@ initialdir = $(home)/mydir
#@ output = job$(jobid).out
#@ error = job$(jobid).err
#@ notification=always
#@ notify_user=erika.mustermann@xyz.de
#@ queue
. /etc/profile
. /etc/profile.d/modules.sh
export OMP_NUM_THREADS=16
export KMP_AFFINITY="granularity=core,compact,1"
./myprog.exe
```

4.12. Job command file keywords

4.12.1. Job name

```
#@ job_name = myjobname
```

- Specifies the name of the job. This keyword must be specified in the first job step. If it is specified in other job steps in the job command file, it is ignored.
- The `job_name` only appears in the long reports of the `llq`, `llstatus`, and `llsummary` commands, and in mail related to the job.

4.12.2. Job type

```
#@ job_type=serial | parallel | MPICH
```

Serial (Presently not available at SuperMUC)

- single task job,
- for single task, multiple threads jobs, please see the OpenMP job example above,
- many parallel keywords cannot be specified with a serial `job_type`.

Parallel

- means a POE job,
- multiple tasks, multiple nodes,
- at SuperMUC only complete nodes (40 cores) are given to the users. Nodes are the smallest allocatable units for parallel jobs.

MPICH

- Similar to a parallel job, but handled differently by LoadLeveler internally.

4.12.3. Job class

`#@ class` = class_name

Valid classes are:

Class Name	Purpose	Max. islands	Max. nodes (cores, SMT threads)	Wall Clock Limit	Run limit per user
Job classes on SuperMIG					
`test`	test and interactive use	1	4 (160 cores)	2 h	1
`general`	general purpose Job	1	52 (2080 cores)	48 h	1
`special`	restricted use (by LRZ, IBM or on request particular users)	1	200 (8000 cores)	unlimited	1
Job classes on SuperMUC					
`test`	test and interactive use	1	32 (512, 1024)	2 h	1
`general`	general purpose Job, within one Island	1	512 (8192, 16384)	48 h	tbd
`large`	large Jobs, spanning more than one Island. You must specify the LL keyword: `#@island_count=...`	4	2048 (32768, 32768 [a])	48 h	tbd
`special`	restricted use (by LRZ, IBM or on request by particular users). You must specify the LL keyword: `#@is-`	18	9216 (147456, 147456 [a])	unlimited	1

Class Name	Purpose	Max. islands	Max. nodes (cores, SMT threads)	Wall Clock Limit	Run limit per user
	land_ count=...				

[a] For large core counts it is not recommended to use SMT threads/Hyperthreading.

Hint: use "llclass -l" to see all limits and definitions.

4.12.4. Limits

#@ wall_clock_limit = hardlimit[,softlimit]

Specifies the hard limit, soft limit, or both for the elapsed time for which a job can run. Limits are specified with the format hh:mm:ss (hours:minutes:seconds).

4.12.5. Parallel jobs

4.12.5.1. Keywords for island, node and core allocation

#@ island_count = <min[,max]>

#@ island_count = number

If a job requires more than 512 nodes, this keyword must be specified. Each Island on SuperMUC contains 512 nodes. The scheduler attempts to start the job on min islands, but will start the job on up to max islands if the job does currently not fit on min islands (e.g., because there are not enough free nodes available). Since SuperMUC has a pruned (1:4) network topology, there is a trade-off between performance and more chances to get a job allocated.

LRZ recommends to use an island count which is slightly higher (+1 or +2) than the minimum requirement to give the LoadLeveler more flexibility for finding free nodes. Example: if you want to run on 2000 nodes, you must use at least 4 islands. In this case specify

```
#@ island_count = 4,5
```

instead of just requiring just 4 islands.

For jobs with less than 512 you can also use island_count. But always specify a minimum island_count of 1 to avoid a fragmentation of the machine by small jobs.

#@ node_topology = island

This keyword is needed for SuperMUC, but is automatically set by the submit filter.

#@ node = <min,max>

The scheduler attempts to get max nodes to run the job step, but will start the job step on min nodes if necessary.

#@ node = <number>

The scheduler will find number nodes on which to run the job step.

#@ tasks_per_node = <number>

Used in conjunction with #@ node, each node is assigned number tasks. tasks_per_node must be less or equal 40.

`#@ total_tasks = <number>`

Rather than specifying the number of tasks to start on each node, the total number of tasks in the job step across all nodes can be specified.

`#@ blocking = <number|unlimited>`

Tasks are allocated in groups (blocks) of the specified number (blocking factor). A node could run more than one block of tasks. The assignment function will assign one block at a time to the machine which is next in the order of priority until all of the tasks have been assigned. If the total number of tasks are not evenly divisible by the blocking factor, the remainder of tasks are allocated to a single node. The blocking keyword must be specified with the `total_tasks` keyword. Specifying `#@ blocking = unlimited` is also known as packing, where the fewest number of nodes possible are selected.

`#@ task_geometry`

Allows you to specify which tasks run together on the same machines, although you cannot specify which machines.

Example:

```
#@ task_geometry = {(5,2) (1,3) (4,6,0)}
```

The 7 tasks are grouped to run on 3 nodes, where tasks 5 and 2 run on one of the nodes, tasks 1 and 3 on another, and tasks 0, 4 and 6 on the third.

Valid combinations of keywords for parallel jobs:

Keyword	**Valid combinations**				
`total_tasks`	X	X			
`tasks_per_node`			X	X	
`node = <min, max>`			X		
`node = <number>`	X			X	
`task_geometry`					X
`blocking`		X			

see also: Task Assignment Considerations [http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=/com.ibm.cluster.loadl41j.admin.doc/am2ug_tskasgn.html]

4.12.5.2. Network protocol

`#@ network.protocol = type[, usage[,mode[,comm_level]]]`

`protocol`: specifies the communication protocols that are used with an adapter,

- `MPI` Specifies the message passing interface (MPI).
- `LAPI` Specifies the low-level application programming interface.
- `MPI_LAPI` Specifies both MPI and LAPI.

`usage`: Specifies whether the adapter can be shared with tasks of other job steps. Possible values are

- `shared`, which is the default,
- `not_shared`.

`type`: This field is required and specifies one of the following:

- `sn_all`: Specifies that striped communication should be used over all available switch networks.

`mode`: Specifies the communication subsystem mode used by the communication protocol that you specify

- `US` (User Space), always use US!
- `IP`, do not use IP, but it is the default!

4.12.6. Restart

```
#@ restart = yes | no
```

Specifies whether LoadLeveler considers a job to be restartable.

If `restart=yes` (default), and the job is vacated (e.g. in case of system errors) from its executing machine before completing, the central manager requeues the job. It can start running again when a machine on which it can run becomes available.

If `restart=no`, a vacated job is canceled rather than requeued.

4.12.7. Files and directories

```
#@ input = filename
#@ output = filename
#@ error =filename
#@ initialdir = pathname
```

`initialdir` specifies the path name of the directory to use as the initial working directory during execution of the job step.

If no filename is specified, LoadLeveler uses `/dev/null`. `filename` can be a absolute filename or it can be relative to the current working directory.

4.12.8. Executable and arguments

The Job Manager copies the executable from the submitting node to the spool directory

```
#@ executable = program_name
```

- The name of the executable that will be started by LoadLeveler.
- If blank, the job command file itself will be used as the executable.
- A different executable can be specified for each step in a job.

```
#@ arguments = arg1 arg2 ...
```

- Specifies the list of arguments passed to the program when the job step runs.
- Different arguments can be specified for each step, even if all steps are using the job command file as the executable.

```
#@ environment = env1; env2; ...
```

Specifies the environment variables that will be set for the job step by LoadLeveler when the job step starts.

- `COPY_ALL`: Copies all environment variables from your shell.
- `$var`: Copies `var` into the job step's environment.

- `!var`: Omits `var` from the job step's environment.
- `var=value`: Specifies that the environment variable `var` be set to the value "`value`" and copied to the job step's environment when the job step is started.

LoadLeveler sets the environment before the login shell is executed, so anything set by the shell (such as settings in .profile) will override the variables set by LoadLeveler.

```
#@ env_copy = all | master
```

Specifies whether environment variables are copied to all nodes of the job step or only to the master node. By default, the environment is copied to all nodes.

```
#@ shell = name
```

When LoadLeveler starts the job, the specified shell will be used instead of the default shell for the user from `/etc/passwd`.

4.12.9. Job steps

A single job can contain more than one step. Every job contains, by default, at least one step. A step can have dependencies on the exit codes of other steps in the same job. Each "`#@ queue`" statement marks the end of one step and the beginning of the next. Most keyword values are inherited from the previous step. A group of steps within a job can be co-scheduled: they are treated as one entity that will all be started at the same time.

4.12.9.1. Dependent steps

```
#@ dependency = step_name operator value
```

`value` is usually a number that specifies the job return code to which the `step_name` is set.

It can also be one of the following LoadLeveler defined job step return codes:

- `CC_NOTRUN`: The return code set by LoadLeveler for a job step which is not run because the dependency is not met. The value of `CC_NOTRUN` is 1002.
- `CC_REMOVED`: The return code set by LoadLeveler for a job step which is removed from the system (because, for example, llcancel was issued against he job step). The value of `CC_REMOVED` is 1001.

Operators include `==, !=, <=, >=, <, >, &&, ||`

A step can have dependencies on more than one step, like

```
#@ dependency = (step1 == 0) && (step2 >= 0)
```

Example job with dependencies:

```
#@ job_name = sim_workflow
#@ step_name = setup
#@ job_type = serial
#@ executable = ~/$(job_name)/setup
#@ queue
#@ step_name = wf_costep1
#@ dependency = (setup == 0)
#@ coschedule = yes
#@ job_type = parallel
#@ executable = /bin/poe
#@ arguments = ~/$(job_name)/runsim
#@ node = 16
```

```
#@ total_tasks = 128
#@ queue
#@ step_name = wf_costep2
#@ dependency = (setup == 0)
#@ coschedule = yes
#@ job_type = parallel
#@ executable = /bin/poe
#@ arguments = ~/$(job_name)/runsim
#@ node = 16
#@ total_tasks = 128
#@ queue
```

4.12.10. Variables

4.12.10.1. Variables only accessible for LoadLeveler command file (at submit time)

Several variables are available for use in job command files.

- `$(domain)`: The domain of the host from which the job was submitted.
- `$(home)`: The home directory for the user on the cluster selected to run the job.
- `$(user)`: The user name that will be used to run the job. This might be a different user name.
- `$(host)`: The hostname of the machine from which the job was submitted.
- `$(jobid)`: The sequential number assigned to this job by the schedd daemon.
- `$(stepid)`: The sequential number assigned to this job step when multiple queue statements are used with the job command file.

Some variables are set from other keywords defined in the job command file:

- `$(executable)`
- `$(class)`
- `$(comment)`
- `$(job_name)`
- `$(step_name)`
- `$(base_executable)`: Automatically set from the executable keyword; consists of the executable file name without the directory component (basename).

Example: `#@ output = $(home)/$(job_name)/$(step_name).$(schedd_host).$(jobid).$(stepid).out`

4.12.10.2. Environment variables (accessible with job)

LoadLeveler sets several environment variables in the application's environment. A complete list is available in Using and Administering [http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.loadl.v5r1.load100.doc/c2367920_xtoc.html], but here are a few examples and explanations:

`LOADLBATCH=yes`	Set when it is a batchjob.
`LOADL_HOSTFILE=filename`	Contains the list of hosts where the job is run.
`LOADL_JOB_NAME=i01adm01.sm.lrz.de.514`	The three part job identifier.
`LOADL_STEP_ID=i01adm01.sm.lrz.de.514.0`	The process ID of the starter process.

LOADL_STEP_COMMAND=/home/prxxxx/luyyyy/JOB	The name of the executable (or the name of the job command file if the job command file is the executable).
LOADL_STEP_CLASS=general	The job class for serial jobs.
LOADL_STEP_ARGS=input1	Any arguments passed by the job step.
LOADL_STEP_ERR=err.51458	The file used for standard error messages (stderr).
LOADL_STEP_OUT=out.51458	The file used for standard output (stdout).
LOADL_STEP_IN=/dev/null	The file used for standard input (stdin).
LOADL_STEP_INITDIR=/home/prxxxx/luyyyy/	The initial working directory.
LOADL_STEP_NAME=0	The name of the job step.
LOADL_TOTAL_TASKS=800	Specifies the total number of tasks of the MPICH job step. This variable is available only when the job_type is set to MPICH.

4.12.11. Notification

`#@ notification = always|error|start|never|complete`

Specifies when mail is sent to the adress in the `notify_user` keywordl:

- `always` : Notify the user when the job begins, ends, or if it incurs error conditions.
- `error`: Notify the user only if the job fails.
- `start`: Notify the user only when the job begins.
- `never`: Never notify the user.
- `complete`: Notify the user only when the job ends (default).

`#@ notify_user = email_address`.

Specifies the address to which mail is sent based on the `notification` keyword.

4.13. Working with energy aware jobs on SuperMUC

Using the energy function, a job can run with a lower CPU frequency to save energy. You can set an acceptable performance degradation (`max_perf_decrease_allowed`) or required energy saving (`energy_saving_req`) for the job in the job command file. LoadLeveler will choose a suitable CPU frequency for the job or reject its submission based on the specified value.

The `energy_policy_tag` helps LoadLeveler identify the energy data associated with a job. With the energy data, LoadLeveler can decide which frequency should be used to run the job with minimal performance degradation.

The energy policy tag identifies the energy associated with a job. The energy data includes:

- Power consumption and the elapsed time when run in the nominal frequency.
- The estimated power consumption.
- The elapsed time in other frequencies.
- The percentage of performance degradation.

Setting the energy policy tag in the job command file, the energy data will be generated and stored in the database when running the job for the first time. If the job is submitted again with the same energy policy tag, the same

policy will be used. Submitting jobs using a new energy function for the first time, take care of keeping the tag name unique among the tags you have previously generated.

Not specifying the `energy_policy_tag` and using in subsequent jobs the `max_perf_decrease_allowed` tag will run the job with default, not maximum clock speed. In that case defaults `forenergy_policy_tag` and `max_perf_decrease_allowed` are used and the job will run at 2.3 GHz.

To set the energy keywords in the job command file to use the energy function, follow these steps:

- Provide an unique identifier for the `energy_policy_tag` when submitting a job for the *first* time. For example:

  ```
  #@ energy_policy_tag = my_energy_tag
  ```

  The identifier may contain all alpha numeric charaters, as well as, - and _ , but no @ and . character. The period may only be used to attach usernames (see below).

- LoadLeveler generates the energy data associated with this energy tag for the job when the job runs. After the job ended, the energy data using the energy tag can be queried the following way:

  ```
  llrqetag -e my_long_running_job [-u user] [-j jobid]
  ```

- You can set an acceptable level of performance degradation (`#@ max_perf_decrease_allowed`) or the wished level of energy saving (`#@ energy_saving_req`) in the job command file and resubmit the job to run at a lower energy level.

  You cannot submit a job with an existing energy tag without specifying `max_perf_decrease_allowed`! For the moment, `max_perf_decrease_allowed` accepts only values > 0 and `energy_saving_req` is temporarily not working!

  For example add the following to your job command file and submit it again:

  ```
  #@ energy_policy_tag = my_energy_tag#@
  ```

  ```
  max_perf_decrease_allowed = 20
  ```

- If you want to use the energy data generated by another user, you can access them by using the asociated energy tag together with the other users username as prefix separated by a period, e.g.:

  ```
  #@ energy_policy_tag = USERNAME.his_energy_tag#@
  ```

  ```
  max_perf_decrease_allowed = 20
  ```

- You can remove energy tags by:

  ```
  llrrmetag -e energy_tag [-u user] [-j job] [-t MM/DD/[YY]YY]tag]
  ```

  Here the `-t MM/DD/[YY]YY` option means: The energy tag will be removed if it has not been used since the date specified.

- At the moment there are some problem when using the energy function and libraries for performance measurements at the same time. In those case it is necessary to switch the the energy function off. You can do that via

  ```
  #@ energy_policy_tag = NONE
  ```

  or by setting the environment `$LL_BYPASS_ETAG` variable in the submitting shell.

4.14. Querying the status of a job

The `llq` command lists all job steps in the queue, one job step per line.

`llq -u userlist` filters out only those job steps belonging to the specified users.

`llq -j joblist` will display only the specified jobs.

The format of a job ID is host.jobid.The format of a step ID is host.jobid.stepid.

4.14.1. Fields in llq's listing

Class: Job class.

Id: The format of a full LoadLeveler step identifier is host.jobid.stepid.

Owner: User ID that the job will be run under.

PRI: User priority of the job step.

Running On: If running, the name of the machine the job step is running on. This is blank when the job is not running. For a parallel job step, only the first machine is shown.

ST: Current state of the job step.

Idle (I): The job step is waiting to be scheduled.

NotQueued (NQ): The job step is not being considered for scheduling, but it has been submitted to LoadLeveler and the Job Manager and Scheduler do know about it, e.g:

Job steps whose dependencies cannot yet be determined.

Job steps that have requested to run in a non-existent reservation.

Job steps submitted above installation-defined limits on queued or idle jobs.

No user intervention can move the job step to Idle state.

User Hold (H): The job step is not being considered for scheduling. It can be released from hold using the llhold -r command by the user who submitted the job.

System Hold (S): The job step is not being considered for scheduling. It can be released only by a LoadLeveler administrator.

User & System Hold (HS): It must be released from hold by a LoadLeveler administrator *and* by the user before it can be scheduled.

Deferred (D): The job step was submitted with a startdate, and that date and time have not yet arrived.

Running (R): The job step is currently running.

Pending (P): The scheduler has assigned resources to the job step and is in the process of sending the start request to the resource manager.

Starting (ST): The resource manager has received the start request from the scheduler and is in the process of dispatching the job step to the nodes where it will run. The next state will normally be Running.

Completed (C): The job step has completed.

Canceled (CA): The job step was canceled by a user or an administrator.

Preempted (E): The job step has been preempted by the suspend method, either by the scheduler or by an administrator using the llpreempt command.

Preempt Pending (EP): LoadLeveler is in the process of preempting the job step.

Resume Pending (MP): The job step is being resumed from preemption. The next state will normally be Running.

Submitted: Date and time of job submission.

4.15. Using llrun to start interactive or batch jobs

A convenient way to start both interactive or batch jobs is the LRZ command "`llrun`". Load the module `lrz-tools` and use `llrun`:

```
  module load lrztools
  llrun -N <nodes> -n <mpi_processes> -t <threads> executable
```

```
Usage : llrun [<options>] <exe> [<user_or_poe_args>]
  <options>:
-N: number of nodes (Default: 1)
-p: total number of processes, same as -n (Default: 1)
-n: total number of processes, same as -p (Default: 1)
-P: Task per node
-t: number of threads per process (Default: 1)
-s: use SMT/Hyperthreading (oversubscribe nodes)
-A: No automatic adjustment of number of nodes (Default: automatic adj.)
-f: Pin processes and Threads to Fixed physical cores (IBM MPI only)
-b: submit batch job
-c: submit to class (default: test)
-m: email
-w: submit batch job with wallclock limit (Default: 00:15:00)
-i: include content of file before parallel execution
-I: include content of file after  parallel execution
-o: do not run/submit job but save to file
-h: help (this message)
-v: verbose (Default)
-V: NO verbose
```

Examples:

```
 llrun -b -n 320 ./myexe
```

(runs 320 tasks on 8 Nodes because of automatic adjustment)

```
 llrun -b -N 10 -n 320 ./myexe
```

(runs 320 tasks on 8 Nodes because of automatic adjustment)

```
 llrun -b -N 10 -P 32 -A ./myexe
```

(runs 320 tasks on 10 Nodes, no automatic adjustment)

```
 llrun -b -t 2 -n 320 ./myexe
```

(runs 320 tasks each which 2 threads on 16 Nodes)

```
llrun -b -n 320 -M intel ./myexe.intel
```

(runs 320 tasks with an Intel-MPI executable)

```
llrun -b -n 40 -c test ./myexe
```

(runs 40 tasks in the test queue)

```
llrun -b -n 40 -c general ./myexe
```

(runs 40 tasks in the general production queue)

```
llrun -n 40 ./myexe
```

(runs directly 40 tasks using terminal for stdout and sterr)

4.16. Why isn't my job running?

First, run `llq -s job_ids`

- it will -provide information on why a selected list of jobs remain in the Hold, NotQueued, Idle or Deferred state. Is the job step's class available?

Are machines configured to run your job class available?

- Run `llclass` to see if the class is defined and has available initiators.
- `llstatus -l` will show Configured Classes and Available Classes.
- `llstatus` will show if machines are Idle or Busy.

Does the job step have requirements which cannot be met by any available machines?

Are higher priority job steps getting scheduled ahead of your job step or are they reserving resources that your job step cannot backfill?

- `llq -l` will show `q_sysprio` which is what is used to order the job steps in the queue.
- The `llprio` command can be used to adjust a job step's priority relative to that user's other submitted job steps.

Is the job step bound to a reservation that has not yet become active?

If the job status immediately goes to "Hold" check that the files for "outpout" and "error" really can be written (diirectory exists and has write permissions etc.).

4.17. Further information

- IBM Manuals on LoadLeveler
- Job command file syntax [http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.loadl.v5r1.load100.doc/am2ug_jobsyn.htm]
- Job command file keyword descriptions [http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic/com.ibm.cluster.loadl.v5r1.load100.doc/am2ug_jobkey.htm]

5. Programming environment

We recommend the use of IBM MPI in combination with the Intel compiler suite. The modules `mpi.ibm` *and* `fortran/intel` *and* `ccomp/intel` *are loaded per default.*

5.1. Development tools and libraries

5.1.1. Overview

Activity	**Tools**	**Linux versions**
Source code development	Editors	vi, vim, emacs, etc.
Executable creation	Compilers	icc, icpc, ifort, gcc, gfortran, g95, pgif90, pgicc
Parallel executable creation	Compilers	mpif90, mpicc, mpiCC: provided by module for the IBM-MPI and Intel-MPI environment
Archiving	Library Archiver	ar
Object and executable file inspection	Object tools	objdump, ldd
Debugging	Debuggers	gdb, idb, totalview, ddd, DTT, Threading Tools
Performance analysis	Profilers	VTune Amplifier XE (aka VTune), Intel Tracing Tools,
Automation	Make	make, gmake
Environment configuration	modules, embedded Tcl	module

For details see the SuperMUC Software [http://www.lrz.de/services/compute/supermuc/software/] page for a listing of available software packages.

5.1.2. Eclipse (integrated development environment)

Eclipse was designed as an integrated development environments (IDE). The Eclipse CDT [http://www.eclipse.org/cdt/] installed at LRZ provides an IDE for C/C++ development, as well as keeping the (generic) support for developing Java applications and Eclipse plug ins, since Eclipse itself is programmed in Java.

Main features of the Eclipse CDT include:

- C/C++ Editor (basic functionality, syntax highlighting, code completion etc.)
- C/C++ Debugger (using GDB)
- C/C++ Launcher (APIs & Default implementation, launches and external application)
- Search Engine
- Content Assist Provider
- Makefile generator
- Graphical CVS management

Photran [http://www.eclipse.org/photran/], which provides an IDE for Fortran90, 95, possibly 95+, has been integrated with CDT, and VTune Amplifier XE [http://www.lrz.de/services/software/programmierung/intel_amplifier/] (formerly known as VTune) with full Eclipse support. Hopefully, Threading Tools will be included in future, as well. Furthermore, there are plans at LRZ to install and support additional Eclipse toolkits as these become available and/or stable enough to be recommended for general use.

- Details on Eclipse at LRZ [http://www.lrz.de/services/software/programmierung/eclipse/]

5.2. Compilers and parallel programming

A complete list of all compilers and parallel programming libraries available on SuperMUC can be obtained using the following module command

```
module avail -c compilers
module avail -c parallel
```

which list all packages installed in LRZ's module classes compilers and parallel.

Important: Using compilers within batch jobs is explicitly disfavored and no compiler licences are provided on compute nodes. Moreover, massive parallel batch jobs are able to execute numerously compiler licences requests which will eventually bring down the licence server. Therfore, such jobs must be avoided.

5.2.1. Intel compilers and performance libraries

Since SuperMUC is based on Intel's SandyBridge/Westmere technology, LRZ recommends the usage of the Intel Compilers [http://www.lrz.de/services/software/programmierung/intel_compilers/] and Performance Libraries [http://www.lrz.de/services/software/programmierung/intel_libs/] as a first choice. Licensing and support agreements with Intel ensure that bug-fixes should be available within reasonably short order.

5.2.2. GNU compilers

We recommend to use the Intel Compilers on SuperMUC. Only if strict compatibility to gcc/gfortran is necessary use these compilers.

5.2.3. PGI compilers

Some commercial packages still require availability of the Portland Group [http://www.pgroup.com/] compiler suite. High Performance Fortran is also supported by the PGI Fortran compiler, as well. Hence, this package is available on SuperMUC. In order to use the PGI compiler on the login-nodes of SuperMUC, please load the compiler modules *ccomp/pgi* and *fortran/pgi*. Anyway, support for the PGI compilers is limited: the LRZ HPC team will report bugs to PGI, but this is kept at low priority.

Documentation for the PGI compilers is available from the PGI web site [http://www.pgroup.com/resources/docs.htm].

5.2.4. MPI

Programs parallelized with MPI can of course run on SuperMUC. Please refer to the Section "Parallelization models supported on SuperMUC" for general information about MPI on SuperMUC, and to the Sections IBM MPI and POE, and IntelMPI, for specific informations about the MPI standard flavors on SuperMUC. In the MPI introductory document [http://www.lrz.de/services/software/parallel/mpi/] additional details on MPI and further available MPI flavours and their handling can be found.

5.2.5. OpenMP

The Multi-Threading capabilities of Intel SMPs can be used via the OpenMP implementations of the Intel and PGI Compilers. Examples for OpenMP programming in Fortran as well as general information about OpenMP can be found in the Sections Parallelization models supported on SuperMUC and the OpenMP introduction.

5.3. An overview of compiler functionality

The following sections discuss the most commonly used compiler switches and extensions implemented by the Intel, PGI, and g95 Compilers. We give an overview of the available optimization switches. If you experience any difficulties, you might have to progressively switch off some of them again.

Please also consult the Performance analysis section for further details and tools for optimization.

5.3.1. Optimization options for x86_64 processors

(see Compiler Documentations [http://www.lrz.de/services/compute/linux-cluster/doc/index.html#compilers] for further details.)

Option *Intel*	**Option *PGI***	**Option gcc, gfortran**	**Meaning**	**Comments**
`-O[0-3]`	t.b.d.	t.b.d.	Specifies the code optimization level for applications.	Here `-O0` specifies no optimization, whereas `-O3` specifies the highest optimization level).
`-fast`	t.b.d.	t.b.d.	Maximizes speed across the entire program.	Sets the following options `-ipo, -O3, -no-prec-div, -static,` and `-xHost` .
`-xHost`	t.b.d.	t.b.d.	Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor.	Host maybe replaced by `AVX1/2, SSE 4.1 / 2, SSE 2/3, SSSE3`.
Intel compiler version 10 and higher have `-opt-streaming-stores [always|never|auto]`, for other versions please use the source code directive `!DEC$ VECTOR NONTEMPORAL` instead.	`-Mnon temporal`		Some programs may slow down with `-fastsse` due to prefetches used. Adding `-Mnontemporal` offers a different data movement scheme which may improve performance.	Worth a try during code tuning. May especially be useful for memory-bound code, since this supports *cache bypass* for streaming writes.

5.3.2. Options for code transformations, aliasing and interprocedural optimization

Option *Intel*	**Option *PGI***	**Option gcc, gfortran**	**Meaning**	**Comments**
`-fno-alias (-fno-fnalias)`	n/a	n/a	Assume no aliasing (within functions).	This may give a considerable performance increase. *Beware:* Check your code yourself for pointer aliasing!
`-unroll [<number>]`	`-Munroll [=n:< number>]`	`-funroll-loops,`	Unroll loops.	`<number>` (optional) gives the maximum number of times for unrolling.

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
		`-fun-roll-all-loops`		0 disables unrolling, omitting it enables compiler heuristics for unrolling. Note that for the Intel compiler you can instead use a source code directive `!DEC$ UNROLL( <number>)` in your code, which might be more useful.
`-ip`	`-Minline [=option [, option, ... ]]`	`-fin-line-func-tions`	Enables interprocedural optimizations for single file compilation.	Performs inline function expansion for calls to functions defined within the current source file. For Intel compilers, you can disable full/partial inlining enabled by this option by also specifying `-ip_no_inlining/-ip_no_pinlining`. For the PGI compiler, please check out man page and user's guide for more information on inlining.
`-ipo`	`-Minline` and `-Mextract` with suboptions	n/a	Enables multifile interprocedural (IP) optimizations (between files).	Performs inline function expansion for calls to functions defined in separate files. For the Intel compiler, a set of source files must be specified as an argument. For the PGI compiler, an inline library must be explicitly created.

5.3.3. Linkage options

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
`-static`	`-Bstatic`	`-Wl,-Bstatic -nonshared`	force static linkage	Recommended if binary is to be run on a machine where the compiler is not installed. Considerably increases executable size!
`-[no-]heap-arrays`				Allocate automatic arrays on heap (Fortran; default is to allocate on stack, which may lead to trouble for low stack limits).
`-auto`				Direct all local variables to be automatic (Fortran).
`-c`			compile only, do not link	This follows conventional usage.

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
n/a	`-g77libs`	n/a	add GNU Fortran libraries	Needed if g77-built objects are to be linked correctly. The Intel Compiler does *not* support this.
`-Ldir`			look for libraries in `dir` as well	This follows conventional usage.
`-lmylib`			link with library `libmylib.{a|so}`	This follows conventional usage.

5.3.4. Source format and preprocessing

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
`-FI` or `-fixed [-72|-80|-132]`	`-Mfixed`		fixed format source code [with possibly extended width]	Source file extension `.f` (Intel: also `.ftn .for`) automatically assumes fixed form.
`-FR` or `-free`	`-Mfree`		free format source code	Source file extension `.f90` automatically assumes free form.
`-fpp [-P]`	`-F`		Invoke preprocessor (C-style includes)	Intel Compiler: optional `-P` switch puts preprocessing results in `output_file` instead of compiling it. Open64 Compiler: `-o` switch required for preprocessing to `output_file`. PGI Compiler: source file must have extension `.F`, output is put into matching file with extension `.f`.
`-Dname [=value]`	define preprocessor macro	this follows conventional usage.		
`-Idir`	look for include files in `dir` as well.	This follows conventional usage.		

5.3.5. Options for data and I/O

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
`-i{2|4|8}`			`INTEGER` and `LOGICAL` types of unspecified `KIND` use the indicated amount of bytes	Default value is 4; `-i2` *not* available for Open64.
`-r{4|8|16}`	`-Mr8`	`-r{4|8}`	`REAL` types of unspecified `KIND` use the indicated amount of bytes	Default value is 4. A value of 8 would change all `REAL` variables to `DOUBLE PRECISION`. For the PGI Compilers only promotion

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
				from 4 to 8 byte REAL is available.
Controlled via environment run time option.	`-Mbyteswapio -byteswapio`	(probably not available)	Do unformatted I/O in big endian instead of little endian	PGI Compiler: should enable you to read and write data compatible to Sun and SGI platforms.

5.3.6. Diagnostics, runtime checking and debugging

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
`-g`			Include symbols for debugging	Use idb or Totalview to debug, or pgdbg for PGI-compiled binaries.
`-check all` This option applies to Fortran Compilers only. The argument "`all`" switches on all available checks. It can be replaced by: • `arg_temp_created:` check for copy-in/copy-out for procedure arguments. • `bounds:` performs run-time checks on array subscripts and substring references • `format, output_conversion:` performs run-time checks on formatted I/O • `pointers:` performs run-time checks on pointers and allocatables • `uninit:` run-time checks	`-C`	(g77 had `-ffortran-bounds-check`)	run time checking	Full checking may incur a large performance penalty.

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
on uninitialized variables (except module globals)				

`-opt-report`
 	
`-opt-report-level[min|max]`
		n/a	generate optimization report	The Intel compiler writes the report to `stderr`.

`-list`	`-Mlist`	n/a	provide source listing	The Intel compiler writes the source listing to `STDOUT`, while the PGI compiler produces a file `myprog.lst` from `myprog.f`.

5.3.7. Parallelization and vectorization options

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
`-openmp`	`-mp`		generate multithreaded code from OpenMP directives in the source code	If used, this option must also be specified for linkage.
`-openmp-stubs`	n/a		Compile OpenMP programs for serial mode; directives are ignored and a stub library for the function calls is linked.	If used, this option must also be specified for linkage.
`-openmp-report[0|1|2]`	n/a		Diagnostic level for OpenMP parallelization	
`-parallel`	`-Mconcur [=option [, option]]`		perform (shared-memory) auto-parallelization	If used, this option must also be specified for linkage. Please refer to the PGI User's Guide, Section 3.1.2 for information on the `-Mconcur` suboptions.
`-par-report[0|1|2]`	n/a		Diagnostic level for automatic parallelization	
`-par-threshold{n}`	n/a		set threshold for autoparallelization of loops	`-par_threshold0` *always* parallelize `-par_threshold25`: parallelize if chance of perf. increase is 25%. `-par_threshold75`: parallelize if chance of perf. increase is 75% (*default*).

Option *Intel*	Option *PGI*	Option gcc, gfortran	Meaning	Comments
				`-par_threshold100`: *only* parallelize if absolutely sure. For the PGI compiler, the `-Mconcur` suboptions allow for a finer control of autoparallelization.
`-vec`	t.b.d.		Enables or disables vectorization.	
`-simd`	t.b.d.		Enables or disables the SIMD vectorization feature of the compiler.	
`-vec-report[0-5]`	t.b.d.		Controls the diagnostic information reported by the vectorizer.	Here 0 specifies to report no diagnostic information, for the other levels please consult the Compiler Documentations.
`-vec-threshold[n]`	t.b.d.		Sets a threshold for the vectorization of loops.	`-vec_threshold0`: *always* vectorize. `-vec_threshold75`: vectorize if chance of perf. increase is 50%. `-vec_threshold100`: *only* vectorize if absolutely sure (default).

5.3.8. Compiler directives for the Intel compiler

The following table shows the *source code directives* as supported by the Intel Fortran compiler to help with tuning or debugging applications. Note that for fixed source format the "`!`" comment symbol in the first column needs to be replaced with a "`c`" comment symbol.

Directive	Meaning
`!DEC$ ivdep`	Ignore vector dependencies.
`!DEC$ loop count N`	Software pipelining hint.
`!DEC$ distribute point`	Split large loop.
`!DEC$ unroll N`	Unroll inner loop `N` times. Compiler heuristics used if `N` omitted.
`!DEC$ nounroll`	Do not unroll loop.
`!DEC$ prefetch A`	Prefetch Array `A`
`!DEC$ noprefetch A`	Do not prefetch array `A`.
`!DEC$ vector [CLAUSE]`	Vectorize loop, `CLAUSE= { ALWAYS [ASSERT]| ALIGNED|UNALIGNED|TEMPORAL| NONTEMPORAL [(var1 [, var2]...)] }`

Directive	Meaning
	For further details please see Compiler Documentations [http://www.lrz.de/services/compute/linux-cluster/doc/index.html#compilers].
`!DEC$ novector`	Do not vectorize loop.

5.3.9. Threading Tools

The Threading Tools [http://www.lrz.de/services/software/parallel/threading_tools/] allow you to perform correctness and performance checking on multi-threaded applications (running in shared memory). The parallelization method may be based on POSIX or Linux Threads, or on OpenMP. For OpenMP applications it is necessary to use the Intel compilers in combination with suitably chosen compiler switches to perform the analysis of applications.

- *Thread Checker* is the tool which identifies and locates threading issues. Very often, concurrency problems (race conditions) are overlooked by the user during the parallelization process. This tool reliably identifies all problems of this kind; under the right conditions it is also possible to specify the exact location in the source code where things go wrong.
- *Thread Profiler* is the tool which provides performance analysis for threaded applications. For each parallel region in the code, scalability extrapolations can be performed, provided a sufficient number of program runs with varying number of threads are performed.

5.4. Parallelization models supported on SuperMUC

SuperMUC supports a variety of parallel programming models. This document provides a short overview of these models, and points at more detailed documentation on usage and LRZ-specific provisions.

5.4.1. MPI

This is the library-based approach to distributed memory parallelism in most common use. The fully supported MPI environments are described in more detail in section "Supported MPI implementations" . There is also a general MPI [http://www.lrz.de/services/software/parallel/mpi/] overview page which has a complete list of MPI environments and includes general usage instructions.

5.4.1.1. Fully supported MPI environments

Hardware Interface	supported Compiler	MPI flavour	Environment Module Name	Compiler Wrappers	Command for Starting Executable
Infiniband and shared memory	Intel compilers (others are possible)	IBM MPI	`mpi.ibm`	`mpif90, mpicc, mpiCC`	`poe, mpiexec`
Infiniband and shared memory	Intel compilers (others are possible)	Intel MPI	`mpi.intel`	`mpif90, mpicc, mpiCC`	`poe, mpiexec`

5.4.1.2. Experimental MPI environments

Hardware Interface	supported Compiler	MPI flavour	Environment Module Name	Compiler Wrappers	Command for Starting Executable
Any, but may only partially work	Intel Fortran, C, C++	Open MPI [http://www.lrz.de/	`mpi.ompi`	`mpif90, mpicc, mpiCC`	`mpiexec`

Hardware Interface	supported Compiler	MPI flavour	Environment Module Name	Compiler Wrappers	Command for Starting Executable
or have reduced performance		services/software/parallel/mpi/openmpi/]			
Any, but may have reduced performance for distributed systems	Intel Compilers, GCC (Others are possible)	MPICH2	`mpi.mpich2`	`mpif90, mpicc, mpiCC`	`mpiexec`

5.4.2. OpenMP

This directive-based approach to shared-memory parallelism is discussed in more detail in the OpenMP section.

supporting compiler	compiler invocation	OpenMP compiler switch
Intel	`ifort, icc, icpc`	`-openmp`
GNU, GCC	`gfortran, gcc, g++`	`-fopenmp`

5.5. Supported MPI implementations

5.5.1. IBM MPI

The MPI implementation by IBM is the default MPI environment in use on the Petaflop-class Supercomputer SuperMUC. This section gives implementation-specific hints on the usage of this MPI variant.

5.5.1.1. Overview

IBM's Parallel Environment is a development and execution environment for parallel applications (distributed-memory, message-passing applications running across multiple nodes). It is designed to support development, testing, debuggging, tuning and running high-performance parallel applications written in C, C++ and Fortran on highly scalable SMP clusters.

The Parallel Environment includes the following components:

- The Parallel Operating Environment (POE) for submitting and managing jobs.
- IBM's MPI and LAPI/PAMI libraries for communication between parallel tasks.
- A parallel debugger (pdb) for debugging parallel programs.
- IBM High Performance Computing Toolkit for analyzing performance of parallel and serial applications.

This section explains the basic usage of POE and IBM MPI; the Parallel Environment Tools are explained elsewhere.

5.5.1.2. Starting up MPI programs using mpiexec/POE

5.5.1.2.1. SPMD

```
export MP_NODES=4
export MP_PROCS=160
```

```
poe ./myexe
#or
mpiexec ./myexec
```

5.5.1.2.2. MPMD

```
cat >cmdfile <<EOD
./master
./slave1
./slave1
./slave1
./slave2
./slave2
EOD
export MP_NODES=1
export MP_PROCS=6
export MP_CMDFILE=cmdfile
poe ./myexe
#or
mpiexec ./myexec
```

5.5.1.3. Further information

- IBM Manuals on POE and MPI
- `man poe`

5.5.2. Intel MPI

Intel's MPI implementation allows to build an MPI application once and run it on various interconnects. Good performance can be achieved also over proprietary interconnects if the vendor provides a DAPL implementation which Intel MPI can make use of.

5.5.2.1. Setting up for the use of Intel MPI

Since Intel MPI is not the default MPI flavour, you may need to unload the default MPI environment before loading the Intel MPI module:

```
module unload mpi.altix mpi.mpt mpi.ibm mpi.parastation mpi.mpich2

module load mpi.intel
```

This environment module makes available all tools needed to compile and execute MPI programs. Since Intel MPI is (probably) not binary compatible to any other MPI flavour, you should completely re-compile and re-link your application under the new environment.

5.5.2.2. Compiling and linking

The following table lists a number of options which can be used with the compiler wrappers in addition to the usual switches for optimization etc.

Option	***Meaning***	***Remarks***
`-mt-mpi`	link against thread-safe MPI	thread-safeness up to `MPI_THREAD_MULTIPLE` is provided

Option	*Meaning*	*Remarks*
`-static-mpi`	use static instead of dynamic MPI libraries	default is dynamic
`-t[=log]`	compile with MPI tracing.	The module `mpi_tracing` must be loaded after the `mpi.intel` module
`-ilp64`	link against MPI interface with 8 byte integers	you may need to also specify `-i8` for compiling your code.

5.5.2.3. Executing Intel MPI programs

5.5.2.3.1. Execution on SuperMUC

Before executing the binary, the `mpi.intel` module must again be loaded in the executing shell; this will, where necessary, also set up the so-called MPD ring by automatically running the `mpdboot` command. For both interactive and batch execution, the `mpiexec` command should be used to start up the MPI program:

```
mpiexec -n 12 ./myprog.exe
```

5.5.2.3.2. Hybrid program execution

Newer versions of Intel MPI can use process pinning to achieve good performance for mixed MPI and OpenMP programs: For example, the command sequence

```
export OMP_NUM_THREADS=4

export I_MPI_PIN_DOMAIN=omp:compact

mpiexec -n 12 ./myprog.exe
```

will start 12 MPI tasks with 4 threads each, keeping threads as near to their master tasks by spreading out the MPI tasks. This is probably the most efficient way to proceed in the majority of cases.

5.5.2.3.3. Handling environment variables

The mpiexec command takes a number of options to control how environment variables are transmitted to the started MPI tasks. A typical command line might look like

```
mpiexec -genv MY_VAR_1 value1 -genv MY_VAR_2 value2 -n 12 ./myprog.exe
```

Please consult the documentation linked below for further details and options.

5.5.2.4. Documentation

5.5.2.4.1. General information on MPI

Please refer to the MPI page at LRZ [http://www.lrz.de/services/software/parallel/mpi/] for the API documentation and information about the different flavors of MPI available.

5.5.2.4.2. Intel MPI documentation

After the *mpi.intel* module is loaded, the $MPI_DOC environment variable points at a directory containing PDF format reference manuals and other documents.

For the most up-to-date release, the documentation can also be found on Intel's web site [http://software.intel.com/en-us/articles/intel-mpi-library-documentation/].

5.6. OpenMP - Parallel programming on shared memory systems

5.6.1. An abstract description of OpenMP

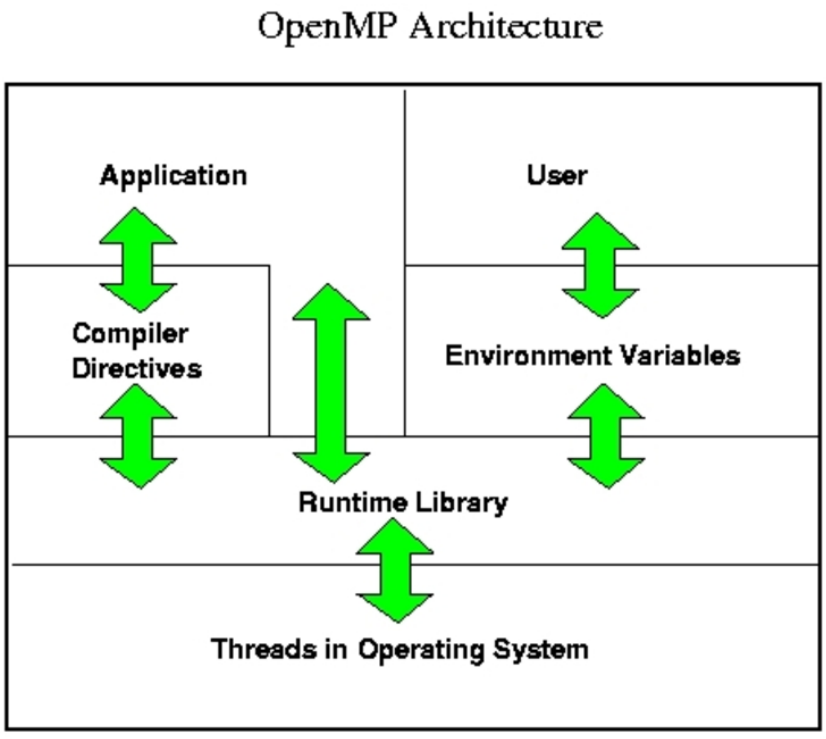
OpenMP is a parallelization method available for the programming languages Fortran, C and C++, which is targeted toward use on shared memory systems. Since the OpenMP standard was developed with support from many vendors, programs parallelized with OpenMP should be widely portable.

The most current unified standard 3.1 [http://www.lrz-muenchen.de/services/software/parallel/openmp/spec31.pdf] for the Fortran, C and C++ base languages was released in July, 2011. This is supported by most compilers.

5.6.1.1. The OpenMP parallelization model

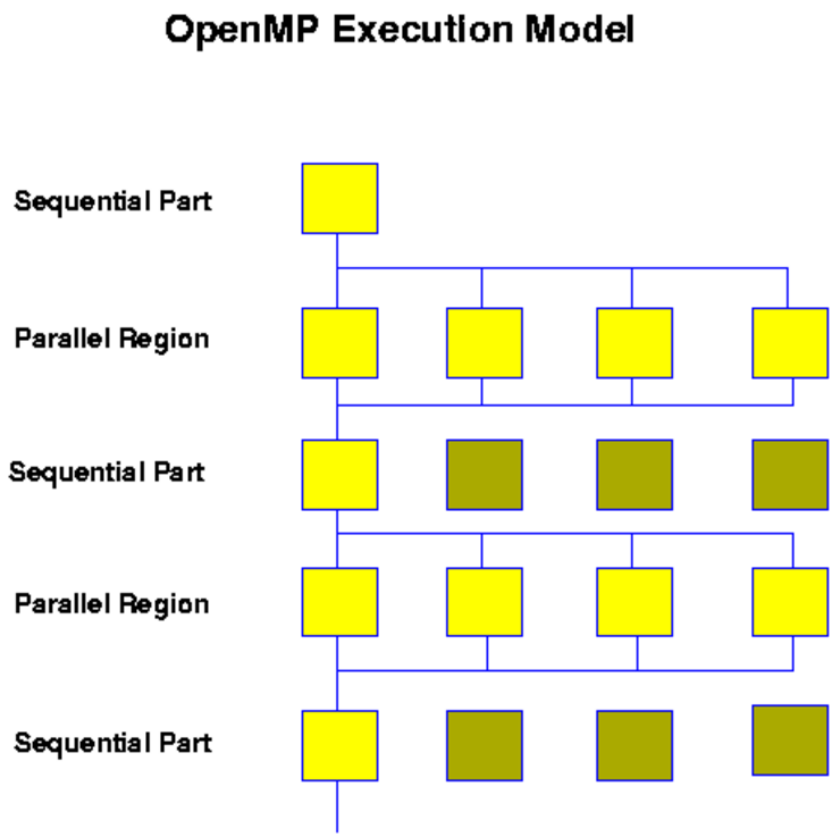
From the *operating system* point of view, OpenMP functionality is based on the use of threads, while the *user's* job simply consists in inserting suitable parallelization directives into her/his code. These directives should not influence the sequential functionality of the code; this is supported through their taking the form of Fortran comments and C/C++ preprocessor pragmas, respectively. However, an OpenMP aware compiler will be capable of transforming the code-blocks marked by OpenMP directives into threaded code; at run time the user can then decide (by setting suitable environment variables) what resources should be made available to the parallel parts of his executable, and how they are organized or scheduled. The following image illustrates this.

Figure 10. OpenMP architecture



However hardware and operation mode of the computing system put limits to the application of OpenMP parallel programs: Usually, it will not be sensible to share processors with other applications because scalability of the codes will be negatively impacted due to load imbalance and/or memory contention. For much the same reasons it is in many cases not useful to generate more threads than CPUs are available. Correspondingly, you need to be aware of your computing centers' policies regarding the usage of multiprocessing resources.

At run time the following situation presents itself: Certain regions of the application can be executed in parallel, the rest - which should be as small as possible - will be executed serially (i.e., by one CPU with one thread).



Program execution always starts in serial mode; as soon as the first parallel region is reached, a team of threads is formed ("forked") based on the user's requirements (4 threads in the image above), and each thread executes code enclosed in the parallel region. Of course it is necessary to impose a suitable division of the work to be done among the threads ("work sharing"). At the end of the parallel region all threads are synchronized ("join"), and the following serial code is only worked on by the master thread, while the slave threads wait (shaded yellow squares) until a new parallel region begins. This alteration between serial and parallel regions can be repeated arbitrarily often; the threads are only terminated when the application finishes.

The OpenMP standard also allows nesting of parallelism: A thread in a team of threads may generate a sub-team; some OpenMP implementations however do not allow the use of more than one thread below the top nesting level.

A priori the number of threads used does not need to conform with the number of CPUs available for a job. However for achieving good performance it is necessary to determine and possibly enforce an optimal assignment of CPUs to threads. This may involve additional functionality in the operating system, and is *not* covered by the OpenMP standard.

5.6.1.2. Comparison with other parallelization methods

In contrast to using MPI (which usually requires a lot of work), one can very quickly obtain a functioning parallel program with OpenMP in many cases. However, in order to achieve good scalability and high performance it will be necessary to use suitable tools to perform further optimization. Even then, scalability of the resulting code will not always be on par with the corresponding code parallelized with MPI.

	MPI (shared and distributed memory systems)	**OpenMP (shared memory systems)**	**proprietary directives**	**PGAS (coarrays, UPC)**
portable	yes	yes	no	yes

	MPI (shared and distributed memory systems)	**OpenMP (shared memory systems)**	**proprietary directives**	**PGAS (coarrays, UPC)**
scalable	yes	partially	partially	yes
supports data parallelism	no	yes	yes	partially
supports incremental parallelization	no	yes	yes	partially
serial functionality intact?	no	yes	yes	yes
correctness verifiable?	no	yes	?	in principle yes

On high performance computing systems with a combined shared and distributed memory architecture using MPI and OpenMP in a complementary manner is one possible strategy for parallelization ("hybrid parallel programs"). Schematically one obtains the following hierarchy of parallelism:

- The job to be done is subdivided into large chunks, which are distributed to (fat) compute nodes using MPI.
- Each chunk of work is then further subdivided by suitable OpenMP directives. Hence each compute node generates a team of threads, each thread working on part of a chunk.
- On the lowest level, e.g. the loops within part of a chunk, the well-known optimization methods (either by compiler or by manual optimization) should be used to obtain good single CPU or single thread performance. The method used will depend on the hardware (e.g., RISC-like vs. vector-like).

Note that the hardware architecture may also have an influence on the OpenMP parallelization method itself. Furthermore, unlimited intermixing of MPI and OpenMP requires a thread-safe implementation of MPI; the level of thread-safeness can be obtained by calling the *MPI_Init_thread* subroutine with suitable arguments; depending on the result, appropriate care may be required to follow the limitations inherent in the various threading support levels defined by the MPI standard.

5.6.2. Overview of OpenMP functionality

A partial description of OpenMP functionality is provided by the LRZ OpenMP presentation [http://www.lrz.de/services/software/parallel/openmp/Kurs/OpenMP.pdf] (900 kByte PDF), with a separate presentation [http://www.lrz.de/services/software/parallel/openmp/Kurs/OpenMP_optex.pdf] (500 kByte PDF) on optimization and performance issues.

5.6.3. Remarks on the usage of OpenMP

5.6.3.1. OpenMP compilers at LRZ

The following OpenMP enabled compilers are available:

- The Intel Compiler suite presently supports OpenMP 3.1 (12.1 and higher releases).
- The PGI Compiler suite supports OpenMP on x86_64 based systems.
- The GCC supports OpenMP provided at least version 4.2 is used. OpenMP 3.1 support is avaiable in the 4.7 release.

5.6.3.2. Compiler switches

For activation of OpenMP directives at least one additional compiler switch is required.

Vendor	**Compiler calls**	**OpenMP option**
Intel	`ifort / icc / icpc`	`-openmp`

Vendor	Compiler calls	OpenMP option
GCC	`gfortran / gcc / g++`	`-fopenmp`
PGI	`pgf90 / pgcc / pgCC`	`-mp`

5.6.3.3. Controlling the run time environment

5.6.3.4. Stub library, module and include file for Fortran

In order to keep code compilable for the serial functionality, any OpenMP function calls or declarations should also be decorated with an active comment:

```
      implicit none
      ...
!$    integer OMP_GET_THREAD_NUM
!$    external OMP_GET_THREAD_NUM
      ...
      mythread = 0
!$    mythread = OMP_GET_THREAD_NUM()
```

Note that without the `IMPLICIT NONE` statement and missing declaration `OMP_GET_THREAD_NUM` has the wrong type!

If you do not wish to do this, it is also possible to use

- either an include file `omp_lib.h`
- or a Fortran 90 module `omp_lib.f90`

for compilation of the serial code. For linkage one also needs a stub library. All this is provided in the Intel compilers via the option `-openmp-stubs`, which will otherwise produce purely serial code. The above code can then be written as follows:

Fortran 77 style	Fortran 90 style

`implicit none`
`...`
`include 'omp_lib.h'`
`...`
`mythread = OMP_GET_THREAD_NUM()`
	`use omp_lib`
	`implicit none`
	`...`
	`...`
	`mythread = OMP_GET_THREAD_NUM()`

5.6.3.5. OpenMP extensions supported by the Intel compilers

The Intel Fortran and C/C++ compilers provide some additional functionality described in the following.

5.6.3.5.1. Run time control: environment variables

Name	Explanation	Default value
`KMP_AFFINITY`	See the Tuning section.	Schedule threads to cores or threads (logical CPUs) in a user controlled manner.
`KMP_ALL_THREADS`	Maximum number of threads available to a parallel region.	`max(32, 4*OMP_NUM_THREADS, 4*(No. of processors)`
`KMP_BLOCKTIME`	Interval after which inactive thread is put to sleep, in milliseconds. Should be short in throughput mode, can	200 milliseconds

Name	Explanation	Default value
	be longer in turnaround mode; see KMP_LIBRARY below.	
KMP_LIBRARY	Select execution mode for OpenMP runtime library. Possible values are: • throughput: optimized for sharing resources with other programs. • turnaround: suited to dedicated use of resources, as in HPC. • serial: enforce serial execution	throughput
KMP_MONITOR_STACKSIZE	Set stacksize in bytes for monitor thread.	max(32768, system minimum thread stack size)
KMP_STACKSIZE	stack size (in bytes) usable for each thread. *Change if your application segfaults for no apparent reason.* You may also need to increase your shell stack limit appropriately. With OpenMP 3.0 and higher, the standardized variable OMP_STACKSIZE should be used.	2 MByte

Notes:

- Setting suitable postfixes where appropriate allows you to specify units. I.e., KMP_STACKSIZE=6m sets a value of 6 MByte.
- There are also some extension routine calls, i.e. kmp_set_stacksize_s(...) with an implementation dependent integer kind as argument, which can be used instead of the environment variables described above. However this will usually not be portable and usage is hence discouraged unless for specific needs.

5.6.3.5.2. NUMA-related directives

For Fortran, Intel has implemented an additional *proprietary* directive which supports correctly distributed memory initialization and NUMA pre-fetching. This directive is of the form

```
!DIR$ MEMORYTOUCH (array-name[, schedule-type [(chunk-size)]]
                                    [ , init-type])
```

where the parameter names have the following meaning:

- array-name is an array of type INTEGER(4), INTEGER(8), REAL(4) or REAL(8).
- schedule-type is one of STATIC, GUIDED, RUNTIME or DYNAMIC, and should be consistent with the OpenMP conforming processing of the subsequent parallel loops.
- chunk-size is an integer expression.
- init-type is one of LOAD or STORE.

If init-type is LOAD, the compiler generates an OpenMP loop which fetches elements of array-name into a temporary variable. If init-type is STORE, the compiler generates an OpenMP loop which sets elements of array-name to zero. Examples:

```
!DIR$ memorytouch (A)
!DIR$ memorytouch (A , LOAD)
!DIR$ memorytouch (A , STATIC (load+jf(3)) )
!DIR$ memorytouch (A , GUIDED (20), STORE)
```

While the `MEMORYTOUCH` directive is accepted on all platforms, at present it is meaningful only on certain Itanium-based systems with NUMA designs and when OpenMP is enabled.

5.6.4. References, examples and documentation

- Intel Compiler documentation [http://www.lrz.de/services/compute/linux-cluster/doc/] on the LRZ web site.
- OpenMP home page: [http://www.openmp.org/] The central source of information about OpenMP.
- OpenMP specifications [http://openmp.org/wp/openmp-specifications/] for Fortran, C, and C++.
- Compilers and tools: [http://openmp.org/wp/openmp-compilers/] various vendor's implementations and add-ons.

Acknowledgments go to Isabel Loebich and Michael Resch, Höchstleistungsrechenzentrum Stuttgart, for a very stimulating OpenMP workshop and the permission to reuse material from this workshop in this document.

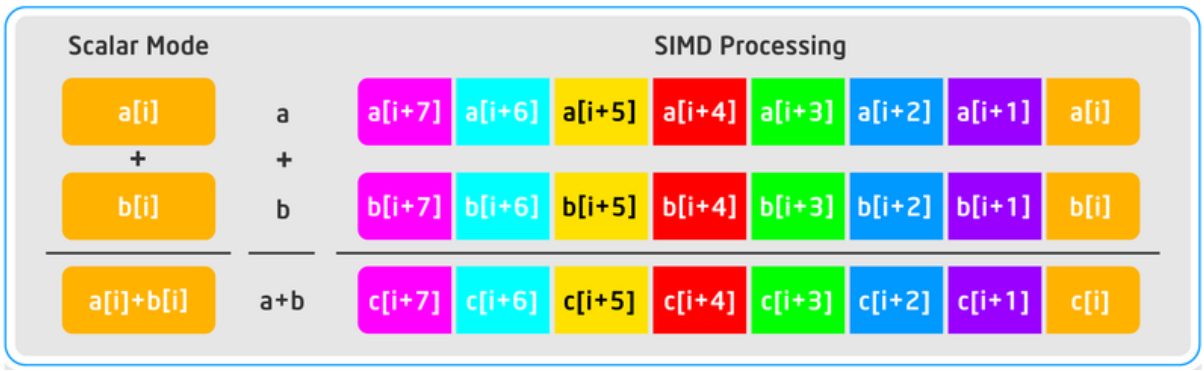
5.7. Vectorization

In the preceding sections, we have discussed about parallelism aimed at exploiting multiple CPU cores through threading and message passing. This section takes a look at exploiting an equally important form of parallelism known as vector parallelism. Vectorization is parallelism within a single CPU core and it is a key form of hardware support for data parallelism. With vectorization, certain operations can be performed on multiple pieces of data at once. It is accomplished by using special instructions called Single Instruction Multiple Data (SIMD) operations. SIMD instructions, and the hardware that supports them, has been present for over a decade. There are several SIMD instruction sets that can be used on SuperMUC. Intel Sandy Bridge processors on the thin node islands support Streaming SIMD Extensions (SSE) up to version 4.2 and Advanced Vector Extensions (AVX). Intel Westmere processors on the fat node islands and on the migration system support SSE4.2 and below, but not AVX.

5.7.1. Overview

In the typical scalar (non-vectorized) case, each variable you use will each be stored in its own CPU register (Figure 11). If you perform an operation on two variables, such as addition, the two register quantities are added and the result is stored back into a register. The vectorized version of this example would first fill a register with multiple variables to be added, which is called "packing" the register. For example, on processors supporting Intel AVX, up to eight single precision floating-point data elements can be packed into one register. Then, using one SIMD instruction, these data elements can be combined with another packed register full of elements, generating multiple results at once. Being able to do these operations in parallel rather than separately can result in significant performance gains for suitable code.

Figure 11. A SIMD addition applied to array elements



Vectorization is typically applied to array or vector data elements that are processed in loops. Besides addition, there are SIMD instructions for many mathematical functions, logical operations, and even string operations.

Developers can access the SIMD instructions in their applications in a variety of ways. Traditionally, vectorization has been accomplished by manipulating SIMD instructions and registers directly, using assembly code or intrinsics provided by the compiler. This method required developers to become experts in SIMD architecture and to hand-tune the code for various CPUs. In addition to requiring significant effort to develop and maintain the code, vectorizing using assembly or intrinsics also has the disadvantage that code is not portable across compilers. Because code must be written for a targeted set of SIMD hardware, the vectorization achieved with this method would not scale forward, meaning it must be re-implemented for new CPUs. When you vectorize your code with one of the methods below, it will be vectorizable without changes for future architectures and CPUs as well.

- Using the compiler auto-vectorizer: When enabled, the compiler auto-vectorizer will look for opportunities to vectorize loops with no changes required to your source code. This method may be all that is needed for vectorization-friendly code. However, the compiler will not vectorize loops if it can't prove that it will be safe to perform the operations in parallel. For this reason, you may see even more vectorization by following up this method with one of the techniques below.
- Using a high-level construct provided by Intel Cilk Plus: Cilk Plus is a set of language extensions for C and C ++ (and in one case, Fortran) that support parallelism and vectorization. Currently Cilk Plus is fully supported by the Intel Compiler, and partially implemented in GCC. It is an open standard. Cilk Plus provides a variety of constructs that can be applied to your code to give the compiler information that it needs to vectorize. Many of these constructs are simple to add to your code — involving only a change in notation or the addition of a pragma.
- Using a high-level Fortran construct: Fortran includes several vectorization-friendly features, such as the ability to operate on arrays as a whole, and the `FORALL` and the `DO CONCURRENT` statements. The Cilk Plus mandatory vectorization directive (`!DIR$ SIMD`) is also available for Fortran.

5.7.2. Automatic vectorization by compiler

5.7.2.1. Enabling vectorization

The compiler will look for vectorization opportunities when higher optimization levels are used. To allow comparisons of vectorized with unvectorized code, vectorization may be disabled.

Intel	PGI	GCC	Effect on vectorization
`-O2 or higher`	`-fast`	`-O3`	Enable (with other performance optimizations)
n/a	`-Mvect=simd[:128|256]`	`-O2 -ftree-vectorize`	Enable (specifically)
`-no-vec`	`-Mnovect`	`-fno-tree-vectorize`	Disable

Intel Sandy Bridge processors on the thin node islands support SSE up to version 4.2 and AVX. Intel Westmere processors on the fat node islands and on the migration system support SSE4.2 and below, but not AVX. To generate code that supports all the available instruction sets, use the following options.

Intel	PGI	GCC	Effect on vectorization
`-xHost`	`-fast`	`-march=native`	Support SSE4.2 and AVX (AVX support is only available in the GCC version 4.6 or later)

5.7.2.2. Vectorization reports

The vectorization report provides you with important information. First, the vectorization report will inform you which loops in your code are being vecrorized. Second, and at least as important, it gives information about why the compiler did *not* vectorize a loop. Vectorization reporting mechanism is not enabled by default.

Intel	PGI	GCC	Effect on vectorization
`-vec-report[0-5]`	`-Minfo=vect`	`-ftree-vectorizer-verbose=[0-7]`	Enable reporting

Below is a typical vectorization report.

```
.\main.c(30): warning : LOOP WAS VECTORIZED.
.\scalar_dep.c(80): warning: LOOP WAS VECTORIZED.
.\main.c(47): warning: loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: nonstandard loop is not a
                                       vectorization candidate
.): warning : loop was not vectorized: nonstandard loop is not a
                                       vectorization candidate
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
.): warning : loop was not vectorized: existence of vector dependence.
.): warning : loop was not vectorized: not inner loop.
```

5.7.2.3. Guided auto-parallelization

The Intel compiler has a very convenient tool that helps you to help the compiler to vectorize: Guided Auto-Parallelization (GAP). GAP causes the compiler to generate diagnostics suggesting ways to improve auto-vectorization, auto-parallelization, and data transformation.

Intel	PGI	GCC	Effect on vectorization
`-guide`	n/a	n/a	Generate GAP report

Note 1: The compiler does not produce any objects or executables when the GAP reporting is enabled.

Note 2: Intel and GCC compilers are compatible. You can run the Intel compiler only on just parts of your application if needed.

As an example, consider a nonvectorizing loop and GAP advice for it.

```
for (i=0; i<n; i++) {

  if (A[i] > 0) { b = A[i]; A[i] = 1 / A[i]; }
  if (A[i] > 1) { A[i] += b; }
}
```

```
scalar_dep.c(80): warning #30515: (VECT)
Assign a value to the variable(s) "b" at
the beginning of the body of the loop in
line 80. This will allow the loop to be
vectorized. [VERIFY] Make sure that, in the
original program, the variable(s) "b"
read in any iteration of the loop has been
defined earlier in the same iteration.
```

The report may suggest making a code change. Make sure the change would be safe to do. Otherwise the compiler may generate incorrect code causing the application to execute incorrectly.

Returning to the example, we make changes suggested in the GAP report. The modified loop is shown below.

```
for (i=0; i<n; i++) {
  b = A[i];
  if (A[i] > 0) { A[i] = 1 / A[i]; }
  if (A[i] > 1) { A[i] += b; }
}
```

5.7.3. Cilk Plus language extensions for C and C++

User-mandated vectorization (`#pragma simd`) is a feature that allows the user to instruct the compiler to enforce vectorization of loops. It is designed to minimise the amount of source code changes needed in order to obtain vectorized code. Pragma simd supplements automatic vectorization just like OpenMP parallelization supplements automatic parallelization. Similar to the way that OpenMP can be used to parallelize loops that the compiler does not normally auto-parallelize, pragma simd can be used to vectorize loops that the compiler does not normally auto-vectorize, even with the use of vectorization hints such as `#pragma vector always` or `#pragma ivdep`.

Below is a loop vectorized with `#pragma simd`. Using it to force vectorization of loops that are not vectorizable due to data dependencies or other reasons may cause the compiler to generate incorrect code. Therefore you must be very careful when applying user-mandated vectorization.

```
#pragma simd
  for (i=x+1; i<n; i++)
    a[i] += a[i-x];
```

Cilk Plus is an open standard, but thus far it is only supported by the Intel compiler.

5.7.4. Vectorization-friendly features of Fortran

Fortran includes several vectorization-friendly features, such as the ability to operate on arrays as a whole, and the `FORALL` and the `DO CONCURRENT` statements. Like Cilk Plus, these constructs are used to give information to the compiler so that it knows when it can vectorize. The Cilk Plus mandatory vectorization directive (`!DIR$ SIMD`) is also available, but thus far only the Intel compiler supports it.

As an example of the SIMD directive, consider a subroutine where the compiler does not automatically vectorize the loop due to a vector dependence. To resolve the problem, you can use the data dependence assertion via the auto-vectorization hint, `!DIR$ IVDEP`, to let the compiler decide to vectorize the loop or not, or you can enforce vectorization of the loop using `!DIR$ SIMD`. The difference between these directives is that with the SIMD directive, the compiler generates a warning when it is unable to vectorize the loop. With auto-vectorization hints, actual vectorization is still under the discretion of the compiler, even when you use the `!DIR$ VECTOR ALWAYS` hint.

Here is an example loop vectorized with the `!DIR$ SIMD` directive.

```
!DIR$ SIMD
DO I=X+1, N
  A(I) = A(I) + A(I-X)
ENDDO
```

When vectorization of a loop is forced with the SIMD directive, the compiler may generate incorrect code. Therefore you must be very careful when adding this instruction.

5.7.5. References

- "Help Future-Proof Performance of Your Application with Vectorization in Six Steps", Issue 10 of the Intel Parallel Universe Magazine [http://software.intel.com/en-us/intel-parallel-universe-magazine/], by Shannon Cepeda and Wendy Doerner
- A Guide to Vectorization with Intel C++ Compilers [http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/]
- Intel Cilk Plus website [http://cilkplus.org/]
- Compiler manpages

6. Performance analysis

Tuning is the last and a fundamental process in software development. However, improving the performance and efficiency of parallel and serial programs is not an easy task. Specific and powerful tools are available to help you to analyze and to deal with performance bottle necks. The user is encorauge to read about, to select and to apply the proper tools and mechanisms that are best suited for own needs.

6.1. Available performance analysis tools

6.1.1. Loading the tools

- Information about avail tools and version: `module avail`
- Loading the appropriate tools: `module load tool or tool/version`

6.1.2. Information

6.1.2.1. HWLoc

The hardware locality toolset provides command line tools as well as a programming interface for identifying and controlling resources and resource mappings for threaded execution.

Further information [http://www.open-mpi.org/projects/hwloc/]

6.1.2.2. likwid-topology

Modern computers get more and more complicated. They consist of multiple cores and each core can support multiple hardware threads. Because cores share caches and main memory access it is important to pin threads to dedicated cores. To decide this it is important to know a machines topology. `likwid-topology` extracts this information from the cpuid instruction.

Further information [http://code.google.com/p/likwid/wiki/LikwidTopology]

6.1.3. Timing & profiling

6.1.3.1. Timing commands and timing functions

Timers can be used to measure the total run time of an application. Different implementations are available on the UNIX and Linux systems. Some subroutines are also available to be called within your code to measure specific sections.

Further information [http://www.lrz.de/services/compute/supermuc/tuning/timers/]

6.1.3.2. gprof

gprof calculates the amount of time spent in each routine. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file which is produced by compiling/linking the executable with `-pg`.

Further information [http://www.lrz.de/services/compute/supermuc/tuning/timers/]

6.1.3.3. Profile guided optimization

Main purpose of profile guided optimization is to re-order instructions in an optimal way. The instrumented executable is run one or more times with different typical data sets. The dynamic profiling information is merged, and the combined information is used to generate a profile-optimized excecutable.

Further information [http://www.lrz.de/services/compute/supermuc/tuning/pgo/]

6.1.4. Using hardware perfomance counters

6.1.4.1. Intel Amplifier XE

The Intel Amplifier XE (formerly VTune) analyzer collects, analyzes, and displays hardware performance data from the system-wide view down to a specific function, module, or instruction.

Further information [http://www.lrz.de/services/software/programmierung/intel_amplifier/]

6.1.4.2. LIKWID

Likwid (Like I knew what I am doing) provides easy to use command line tools for Linux to support programmers in developing high performance multi threaded programs.

Further information [http://www.lrz.de/services/software/programmierung/likwid/]

6.1.4.3. IBM High Performance Computing Toolkit (hpccount)

Report summary hardware performance counters and resource usage statistcs.

Further information [http://www.lrz.de/services/compute/supermuc/tuning/hpccount/]

6.1.4.4. PAPI

PAPI (Performance Application Programming Interface) aims to provide the tool designer and application engineer with a consistent interface and methodology for use of the *performance counter hardware* found in most major microprocessors. PAPI enables software engineers to see, in near real time, the relation between software performance and processor events.

Further information [http://www.lrz.de/services/software/programmierung/papi/]

6.1.4.5. PerSystreport

Perfomance properties are collected by the PerSyst Monitoring system at LRZ. Using HTML format, and also with the capability of generating a detailed report, it presents a user friendly interface.

Further information [http://www.lrz.de/services/compute/supermuc/tuning/persystreport/]

6.1.5. MPI and OpenMP profiling and tracing

6.1.5.1. Vampir NG

Vampir is the *State-Of-The-Art Tool* for tracing parallel programs based on MPI, OpenMP or CUDA, and serial programs. It is designed to provide accurate trace information of MPI and user function calls. The user interface and parallel processing of tracing data makes Vampir NG the most powerful tool for tracing. It includes the capability for performance-counter analysis based on PAPI [http://www.lrz.de/services/software/programmierung/papi/].

Further information [http://www.lrz.de/services/software/parallel/vampir_ng/]

6.1.5.2. Scalasca

Scalasca is an open-source project developed by the Jülich Supercomputing Centre which focuses on analyzing OpenMP, MPI and hybrid OpenMP/MPI parallel applications. Scalasca can be used to help identify bottlenecks by providing a number of important features: profiling and tracing of highly parallel programs; automated trace analysis that localizes and quantifies communication and synchronization inefficiencies; flexibility and integration with PAPI [http://www.lrz.de/services/software/programmierung/papi/] hardware counters for performance analysis.

Further information [http://www.lrz.de/services/software/parallel/scalasca/]

6.1.5.3. Intel Tracing Tools

The Intel Tracing Tools support the development and tuning of programs parallelized using MPI. By using these tools you are able to investigate the communication structure of your parallel program, and hence to isolate incorrect and/or inefficient MPI programming. The *Trace Collector* is a set of MPI tracing libraries, and the *Trace Analyzer* provides a GUI for analysis of the tracing data.

Further information [http://www.lrz.de/services/software/parallel/vampir/]

6.1.5.4. IPM

IPM is a portable profiling infrastructure for parallel C and Fortran programs. It provides a low-overhead profile of the performance aspects and resource utilization. Communication, computation, and IO are its primary focus. Hardware counter are based on PAPI [http://www.lrz.de/services/software/programmierung/papi/].

Further information [http://www.lrz.de/services/software/parallel/ipm/]

6.1.5.5. mpiP

It is a lightweight and scalable MPI profiling library exclusively for MPI applications. It collects statistical information with minimal overhead. The trace data is small in ASCII and human-readable format.

Further information [http://www.lrz.de/services/software/parallel/mpip/]

6.1.5.6. Threading Tools

The Threading Tools allow you to perform correctness and performance checking on multi-threaded applications (POSIX/Linux Threads or OpenMP). *Thread Checker* identifies and locates threading issues like concurrency problems (race conditions). *Thread Profiler* provides performance analysis for threaded applications.

Further information [http://www.lrz.de/services/software/parallel/threading_tools/]

6.1.5.7. Marmot

Marmot is a MPI correctness checker. It automatically checks the correct usage of MPI functions and their arguments. It can identify deadlocks, wrong ordering of messages, wrong MPI types, etc.

Further information [http://www.lrz.de/services/software/parallel/marmot/]

6.1.5.8. Guideview

GuideView is a tool that displays the performance details of an OpenMP program's parallel execution.

Further information [http://www.lrz.de/services/compute/supermuc/tuning/guideview/]

6.1.6. Memory Leaks

6.1.6.1. MemoryScape

This tool provides a subset of Totalview functionality to detect memory leaks.

Further information [http://www.lrz.de/services/software/programmierung/totalview/]

6.1.6.2. Valgrind

For finding memory leaks, measuring memory consumption as well as identifying performance bottlenecks.

Further information [http://www.valgrind.org/]

6.2. Westmere-EX/Sandy-Bridge-EP specific performance counters

6.2.1. Performance Monitoring

Performance issues in parallel applications often arise as a result of poor utilization of platform resources, inefficient parallelization, or even inherent limitations of both the machine to deliver higher performance, and the application to deliver more parallelism. Typical reasons for poor scaling are load imbalance, large serial portions, synchronization overhead, memory bandwidth saturation, excessive read-write sharing, poor scheduling decisions, etc.. The performance monitoring units (PMUs) found in almost all modern processors can help the user understand the way applications interact with the underlying architecture and spot bottlenecks that hinder applications from scaling.

PMUs are used to count key micro-architectural events in the processor, such as instructions, cache misses and branch mispredictions. Currently, Intel processors support two basic modes of counting, *Event Based Counting (EBC)* and *Event Based Sampling (EBS)*. EBC provides the total number of events for a specific interval (e.g. during the total execution time of an application). EBS profiles the application with respect to a specific event. It performs statistical sampling to compute the distribution of event occurrences across the program source code. In this way the user can quickly find "hot-spots" in the program, i.e. statements that are responsible for the majority of occurrences.

Each event is usually defined together with one or more *unit masks*, which can be used to further qualify the event according to certain aspects of its occurrence. For example, an event in Sandy-Bridge that counts the number of blocks evicted out of the L2 cache can be further qualified according to whether the evicted blocks where clean or dirty, whether the evictions were triggered by the application itself or the hardware prefethcer, and combinations of those cases.

Westmere and Sandy-Bridge architectures support three kinds of performance counters:

- *Fixed performance counters*: non-programmable counters that measure a limited set of predefined events.
- *Programmable performance counters*: counters that measure user-specified events in the processor core.
- *Uncore performance counters*: counters that measure events in the "uncore" part of the processor, i.e. the part that interfaces the main execution core with the memory subsystem (e.g. L3 cache, memory controller, etc.).

The three events measured by fixed performance counters in Westmere-EX and Sandy-Bridge-EP processors are summarized in the following table:

Event name	Event mask	Description
INST_RETIRED	ANY	Number of instructions retired.
EVENT_CPU_CLK_UNHALTED	THREAD	Number of real cycles the core/hyper-thread is not halted.
CPU_CLK_UNHALTED	REF	Number of cycles using the processor reference frequency the core/hyper-thread is not halted.

6.2.2. Using performance counters to detect performance problems

The following sections present some of the most common problems that are responsible for weak scalabity, and provide guidelines on how to use performance counters to detect them.

6.2.2.1. Load imbalance and parallelization overhead

The total number of instructions retired by a thread is a good measure of the amount of work it performed. The retired instructions are those that were successfuly executed by the processor until completion (i.e., excluding any uncompleted instructions that were speculatively executed and discarded due to branch misprediction). Therefore, an uneven distribution of threads' instruction counts clearly indicates load imbalance in the parallel application. On the contrary, an even distribution shows that the total computations were successfully assigned equally to all threads.

Another issue that the instructions count may highlight is the parallelization overhead. Specifically, it might be possible that the parallel implementation yields balanced instruction counts among threads, but their accumulation is increased with respect to the serial version. This implies overhead in the parallel implementation, that may be due to very fine-grained work partitioning and distribution, excessive thread synchronization or other costly operations performed under the hood by the parallel runtime system.

The following table presents the event details for retired instructions, both for the Westmere-EX and Sandy-Bridge-EP processors.

Event name	Event mask	Description	Counter kind
INST_RETIRED	ANY	Number of instructions ‘retired’ (changing program state).	Fixed
INST_RETIRED	ANY	Number of instructions ‘retired’ (changing program state).	Programmable

6.2.2.2. Synchronization overhead

The basic indication of excessive synchronization is when threads spend a large amount of time either within critical sections, or in lock methods waiting for lock acquisition. These cases can be detected by sampling CPU cycles events (e.g. EVENT_CPU_CLK_UNHALTED) and then checking whether the aforementioned cases concentrate a large percentage of samples around them.

Sometimes, a side-effect of excessive synchronization are the large waiting periods of threads trying to enter a critical section. The problem becomes more intense as the critical sections increase in size and/or the competition among threads rises. If the synchronization primitives are implemented in terms of OS-based mutexes, the long waiting periods should appear as increased time spent in kernel code. If the synchronization primitives are implemented as user-level spin locks, the waiting overhead should translate to excessive read-write sharing, which is discussed in a following section.

Another possible source of synchronization overhead are atomic instructions. Both Westmere-EX and Sandy-Bridge-EP support atomic execution mode for some instructions (e.g. integer addition). The following table provides events corresponding to the number of cycles that a specific part of memory hierarchy (e.g. caches) was locked as a result of atomic instructions. If the ratio of locked cycles to the total cycles is rather increased, then this would probably indicate high contention while executing atomic operations.

Westmere-EX:

Event name	Event mask	Description
CACHE_LOCK_CYCLES	L1D	Cycle count during which the L1D is locked.
CACHE_LOCK_CYCLES	L1D_L2	Cycle count during which the L1D and L2 are locked.

Sandy-Bridge-EP:

Event name	Event mask	Description
LOCK_CYCLES	CACHE_LOCK_DURATION	Cycles in which the L1D is locked.

Event name	Event mask	Description
LOCK_CYCLES	SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked.

6.2.2.3. Memory bandwidth saturation

Memory-intensive applications can often saturate the memory bus at low thread counts and cannot benefit from using additional cores. This is true not only for conventional SMP architectures, but also for NUMA platforms, such as the SuperMUC's fat and thin nodes, that incorporate multiple *memory nodes* (i.e., multiple sockets with their local memory controller and bus). Even in the latter case, memory-intensive applications might not scale any further if more cores from the same memory node are utilized, because the corresponding memory bus has been saturated and cannot deliver data at higher rates.

To detect bus saturation on a certain memory node the user needs to measure the application bandwidth usage on that node and compare it against the maximum it can sustain. If they are nearly equal, then this indicates saturation. To measure the maximum sustainable bandwidth the user can use a benchmark designed for that purpose, such as the STREAM benchmark from University of Virginia (http://www.cs.virginia.edu/stream/).

Memory bandwidth is the number of bytes transferred per second. To find the bandwidth consumed on a certain memory node, the user needs to divide the total bytes transferred by all threads executing on that node by the total execution time. The events that need to be counted and the bandwith estimation formulas for each processor are given below.

Westmere-EX:

Event name	Event mask	Description
UNC_QMC_NORMAL_READS	ANY	Number of Quickpath Memory Controller medium and low priority read requests (by any memory channel).
UNC_QMC_WRITES	FULL.ANY	Number of full cache line writes to DRAM (by any memory channel).

*Memory bandwidth (GB/sec) = 64 * (UNC_QMC_NORMAL_READS.ANY + UNC_QMC_WRITES.FULL.ANY) * 10^{-9} / (execution time in seconds)*

Sandy-Bridge-EP:

Event name	Event mask	Description
UNC_IMPH_CBO_TRK_REQUEST	ALL	Number of core-outgoing entries (accounts for coherent and incoherent traffic).

*Memory bandwidth (GB/sec) = 64 * UNC_IMPH_CBO_TRK_REQUEST.ALL * 10^{-9} / (execution time in seconds)*

We note also that the setup for bandwidth measurement is pre-configured in the Intel Amplifier XE analyzer.

6.2.2.4. True and false read-write sharing

Excessive read-write or write-write sharing between two or more processors on a centralized data structure can introduce large amount of traffic on the bus. Such scenarios typically cause "ping-pong" transfers of the cache-line containing the data structure between the sharers' caches, due to multiple invalidations. In turn, this introduces large performance penalties, not only because of the actual data transfers but also due to the coherence protocol overhead.

This sharing can sometimes be intentional (*true sharing*), e.g. when the cache line contains a common synchronization or reduction variable, but in other cases it might be unintentional (*false sharing*), because logically unre-

lated data items happened to reside on the same cache line. PMUs provide events that can help the programmer spot locations in the code exhibiting intense read-write sharing. However, there is no means to automatically distinguish false from true sharing. Instead, the programmer should inspect the code sections triggering the related events, and then deduce whether data are being read-write shared on purpose or accidentally.

Westmere-EX:

The events in the following table count the number of memory load instructions that hit modified data in a sibling core and a remote socket, respectively. If there is high occurence of these events (e.g., a measurable percentage of total instructions), then there is intense true or false sharing, and the programmer should inspect the code around the triggering program counter to spot the real cause.

Event name	Event mask	Description
MEM_UNCORE_RETIRED	OHTER_CORE_L2_HIT	Load instructions retired that HIT modified data in sibling core.
MEM_UNCORE_RETIRED	REMOTE_HITM	Load instructions retired that HIT modified data in remote socket.

Sandy-Bridge-EP:

The events for true or false sharing detection are pre-configured in the Intel Amplifier XE analyzer, under the "Access contention" option.

6.2.2.5. Misaligned memory references

The x86 architecture allows the execution of memory references at addresses that are not evenly divisible by the word size or multiples of it. These are known as misaligned accesses. Although most compilers typically align variables at a word-size granularity (or multiples of it), it might be possible for an uncareful user-directed allocation to lead to misaligned addresses. The danger here is that the misaligned variables might span adjacent cache lines, both of which should be transfered each time the variable is referenced. This introduces excessive overhead when these variables are read-write or write-write shared among two or more processors.

Most x86-based architectures provide events to count and detect misaligned memory references. The user should inspect the source code lines around which these events are gathered, in order to apply the appropriate techniques to eliminate misalignment (e.g. padding).

Westmere-EX:

Event name	Event mask	Description
MISALIGN_MEMORY	STORE	All stores referenced with misaligned address.

Sandy-Bridge-EP:

Event name	Event mask	Description
MISALIGN_MEM_REF	LOADS	Speculative cache-line split load uops dispatched to L1D.
MISALIGN_MEM_REF	STORES	Speculative cache-line split Store-address uops dispatched to L1D.

6.2.2.6. Other useful events

In the following paragraphs, a list of events is provided that are typically related to single-thread performance.

- *Branch misprediction*

Branch misprediction can sometimes be an important performance issue, since the penalty to recover from a misprediction is usually many cycles. The ratio of mispredicted to total executed branches should be typically in the range of 2-5%.

Westmere-EX:

Event name	Event mask	Description
BR_INST_EXEC	ANY	Counts all near executed branches (not necessarily retired).
BR_MISP_EXEC	ANY	Counts the number of mispredicted near branch instructions that were executed, but not necessarily retired.

Sandy-Bridge-EP:

Event name	Event mask	Description
BR_INST_EXEC	ALL_BRANCHES	Counts all near executed branches (not necessarily retired).
BR_MISP_EXEC	ALL_BRANCHES	Counts the number of mispredicted near branch instructions that were executed, but not necessarily retired.

- *Long-latency arithmetic operations*

  The divider unit potentially can be a significant bottleneck, since divisions are typically long-latency operations and their execution cannot be pipelined. To cope with such problems, some compilers offer the opportunity to replace a division operation with a multiplication by the reciprocal.

  Westmere-EX:

Event name	Event mask	Description
ARITH	CYCLES_DIV_BUSY	Counts the number of cycles the divider is busy executing divide or square root operations.

  Sandy-Bridge-EP:

Event name	Event mask	Description
ARITH	FPU_DIV_ACTIVE	Counts the number of cycles that the divider is active.

- *Long-latency memory accesses*

  Misses in the Last Level Cache (LLC) have always been considered an important performance metric in traditional single-bus-based arhictectures, since they implied going out to main memory. However, this is not always the case in architectures featuring multiple memory nodes, such as SuperMUC's thin or fat nodes, because the missing data can be in the cache hierarchy of a second socket. Therefore, a hit does not always translate to higher performance since it can be more expensive than a miss, e.g. when data is in modified state in the L1/L2 cache of a sibling core. Or, an L3 modified hit in another socket L3 can be more expensive than accessing the local DRAM.

  A more safe approach to detect critical memory accesses is to qualify each memory reference according to its latency. Both Westmere-EX and Sandy-Bridge-EP processors offer events that count the number of memory references (either misses or hits) that took more than a user-specified threshold of cycles to complete.

Westmere-EX:

Event name	Event mask	Description
MEM_INST_RETIRED	LATENCY_ABOVE_THRES-HOLD	Counts all memory accesses taking more tha X cycles (X can be 4,8,16,32,64,...,32768, and is specified using the MSR_PEBS_LD_LAT_THRES-HOLD MSR).

Sandy-Bridge-EP:

Event name	Event mask	Description
MEM_TRANS_RETIRED	LOAD_LATENCY	Counts load accesses tak-ing more tha X cycles (X can be 4,8,16,32,64,...,32768, and is specified using the MSR_PEBS_LD_LAT_THRES-HOLD MSR).

6.2.3. Complete event list

The user is referred to the following manual for a complete listing of events for Westmere-EX and Sandy-Bridge-EP processors: http://download.intel.com/design/processor/manuals/253669.pdf

7. Tuning

7.1. Westmere/Sandy-Bridge specific single-core optimization

7.1.1. Processor-specific optimization

Both Intel (`icc`) and GNU (`gcc`) C compilers support special flags that enable code generation for specific machine types. These flags are the following:

- `-march=corei7` : produces code exclusively for Westmere processor (MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1 and SSE4.2 instruction sets are supported.)
- `-march=corei7-avx` : produces code exclusively for Sandy-Bridge processor (MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AES and PCLMUL instruction sets are supported.)

7.1.2. Vectorization

The use of the Intel SSE instruction set, which is an extension to the x86 architecture, is called vectorization. SSE instructions are SIMD (Single-Instruction Multiple-Data) instructions, meaning that they operate on multiple data elements in one instruction and make use of the vector registers (XMM registers). Westmere processor features SSE4.2 instruction set (and all previous SSE versions), with 16 128-bit XMM registers. Sandy-Bridge supports the new AVX instruction set (and all previous SSE versions), with 16 256-bit XMM registers. The potential benefit of this extension is up to 2x peak FLOPs along with better power efficiency.

Most compilers support *auto-vectorization*, which is the automatic identification of suitable loops and generation of vectorized code, by combining loop unrolling and SIMD instructions generation. Below, a list of guidelines is given that help the compiler to automatically generate vectorized code. For a more complete discussion on auto-vectorization, the user is referred to the following document: http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers

- Prefer countable single entry and single exit “for” loops. Avoid complex loop termination conditions – the loop lower bound and upper bound must be invariant within the loop. It’s OK for it to be a function of the outer loop indices, for the innermost loop in a nest of loops.
- Write straight line code (avoid branches such as switch, goto or return statements, most function calls, or “if” constructs that can’t be treated as masked assignments).
- Avoid dependencies between loop iterations, or at least, avoid read-after-write dependencies.
- Prefer array notation to the use of pointers. C programs in particular impose very few restrictions on the use of pointers; aliased pointers may lead to unexpected dependencies. Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
- Use the loop index directly in array subscripts where possible, instead of incrementing a separate counter for use as an array address.
- Use efficient memory accesses:
  - Favor inner loops with unit stride.
  - Minimize indirect addressing.
  - Align data to 16 byte boundaries (for Intel SSE instructions).
  - Align data to 32 byte boundaries (for Intel AVX instructions).
- Within a loop nest, the innermost loop is the vectorizable one. The only exception is if an original outer loop is transformed into an inner loop as a result of some other prior optimization phase, such as unrolling, loop collapsing or interchange.
- No function calls should be performed within the loop body. The two major exceptions are for intrinsic math functions (e.g. ceil, floor, acos, sqrt, etc.) and for functions that may be inlined.

7.1.2.1. icc compiler support for vectorization

Icc will look for vectorization opportunities whenever the user compiles at default optimization (`-O2`) or higher. Below are some commonly flags related to vectorization in icc. They can be used for performance or diagnostics reasons.

Flag	**Description**
`-vec-report0`	Disable vectorization diagnostics (default).
`-vec-report1`	Report successfully vectorized code.
`-vec-report2`	Also report which loops were not vectorized.
`-vec-report3`	Also report all prohibiting data dependencies.
`-no-vec`	Disable auto-vectorization regardless of any flags or pragmas in the code.
`-vec-threshold0`	Try to auto-vectorize loops even if the compiler doesn’t think it improve performance; use with caution because it will vectorize a lot of loops that are unsuitable, and slow them down.
`-guide`	Enable guided automatic parallelization and vectorization; compiler gives advices on how to get loops to vectorize or parallelize; only available in 12.0 or later compiler.
`-unroll=0`	Disable loop unrolling.
`-restrict`	Enable use of restrict keyword that conveys non-aliasing properties to the compiler.

Also, the following pragmas can be inserted before the targeted loops in user code to help the compiler auto-vectorize them:

Pragma	Description
`#pragma ivdep`	ignore potential (unproven) data dependences.
`#pragma vector always`	Override efficiency heuristics.
`#pragma vector nontemporal`	Hint to use streaming stores.
`#pragma vector [un]aligned`	Assert [un]aligned property.
`#pragma novector`	Disable vectorization for following loop.
`#pragma distribute point`	Hint to split loop at this point.
`#pragma loop count (<int>)`	Hint for likely trip count.
`#pragma simd`	Enforces vectorization; Only available in 12.0 or later compiler.
`restrict`	Keyword to assert exclusive access through pointer; requires command line option "`-restrict`".
`__attribute__(align(<int>,<int>))`	Requires memory alignment.
`__assume_aligned(<var>,<int>)`	Assert alignment property.

As soon as the desired vectorization flags have been selected, the user should use one of the following flags in order to select the appropriate instruction set extension (e.g. SSE4.2, AVX, etc.).

Flag	Description
`-x<extension>`	The compiler will try to make use of all instruction set extensions up to and including "extension". The option targets Intel processors only. Possible values of "extension", from newer to older, are: AVX, SSE4.2, SSE4.1, SSSE3, SSE3, SSE2. Each instruction set extension includes all the others released before. SSE4.2 is the latest extension supported by Westmere, and AVX the one supported by Sandy-Bridge. Furthermore, with this compiler flag a processor-check is added to the program binary, meaning that the application will not start in case the specified extension is not supported.
-m<extension>	This flag is similar to the above, with the exception that it does not perform Intel processor check and Intel-specific optimizations. The application is optimized for and will run on both Intel and non-Intel processors. The missing processor check can cause application to fail if the extension is not available.
`-ax<extension>`	This option generates dual-code paths, a "generic" and an "optimized" one. The optimized path is that specified by "extension", while the "generic" defaults to SSE2.
`-xHost`	The compiler checks the host processor and makes use of the "latest" instruction set extension available. This can be helpful when we want to avoid builds being executed on multiple, unknown platforms.

7.1.2.2. gcc compiler support for vectorization

The compiler will look by default for vectorization opportunities whenever the user compiles at `-O3` optimization level or higher. Alternatively, the user can use the -free-vectorize flag to enable vectorization regardless of the optimization level. Other related compiler flags are listed in the following table:

Flag	Description
`-ftree-vectorize`	Perform loop vectorization on trees. This flag is enabled by default at '`-O3`'.
`-fdump-tree-vect`	Dump each function after applying vectorization of loops. The file name is made by appending '`.vect`' to the source file name.
`-ftree-vectorizer-verbose=n`	This option controls the amount of debugging output the vectorizer prints. This information is written to standard error, unless '`-fdump-tree-all`' or '`-fdump-tree-vect`' is specified, in which case it is output to the usual dump listing file, '`.vect`'. For n=0 no diagnostic information is reported. If n=1 the vectorizer reports each loop that got vectorized, and the total number of loops that got vectorized. If n=2 the vectorizer also reports non-vectorized loops that passed the first analysis phase. Higher verbosity levels mean either more information dumped for each reported loop, or same amount of information reported for more loops: if n=3, vectorizer cost model information is reported. If n=4, alignment related information is added to the reports. If n=5, data-references related information (e.g. memory dependences, memory access-patterns) is added to the reports. If n=6, the vectorizer reports also nonvectorized inner-most loops that did not pass the first analysis phase (i.e., may not be countable, or may have complicated control-flow). If n=7, the vectorizer reports also non-vectorized nested loops. If n=8, SLP related information is added to the reports. For n=9, all the information the vectorizer generates during its analysis and transformation is reported. This is the same verbosity level that '-fdump-tree-vect-details' uses.
`-floop-flatten`	Removes the loop nesting structure: transforms the loop nest into a single loop. This transformation can be useful as an enablement transform for vectorization and parallelization. This feature is experimental.
`-ftree-vect-loop-version`	Perform loop versioning when doing loop vectorization on trees. When a loop appears to be vectorizable except that data alignment or data dependence cannot be determined at compile time, then vectorized and non-vectorized versions of the loop are generated along with run-time checks for alignment or dependence to control which version is executed. This option is enabled by default except at level '`-Os`' where it is disabled.
`-fvect-cost-model`	Enable cost model for vectorization.

As soon as the desired vectorization flags have been selected, the user should use one of the following flags in order to select the appropriate instruction set extension (e.g. SSE4.2, AVX, etc.).

Flag	Description
`-msse4.2`	Enables the use of instructions in the SSE4.2 extended instruction set. This option should be used for Westmere processor.
`-mavx`	Enables the use of instructions in the AVX extended instruction set. This option should be used for SandyBridge processor.

7.1.2.3. Double-port optimization (Sandy-Bridge)

While prior Intel micro-architectures have one load memory port, Sandy-Bridge micro-architecture features two load ports. Thus, two load operations can be performed every cycle, doubling the load throughput of the code. This improves code that reads a lot of data and does not need to write out results to memory very often. To exploit this bandwidth, the data has to stay in the L1 data cache or it should be accessed sequentially, enabling the hardware prefetchers to bring the data to the L1 data cache in time.

The user can take advantage of double load ports in Sandy-Bridge and restructure his code for higher efficiency. In the following example, all the elements of an array are added and accumulated in a single variable. The result of each iteration is used as input to the next iteration, therefore there is a single load operation performed in every iteration. This limits the throughput to one load and ADD operation per cycle.

```
int buff[BUFF_SIZE];
int sum=0;

for (i=0; i<BUFF_SIZE; i++) {
    sum += buff[i];
}
```

The restructured code can take advantage if the additional port by having a second variable to accumulate the array values. Now, two load and two ADD operations can be executed every cycle.

```
int buff[BUFF_SIZE];
int sum, sum1;
sum=sum1=0;

for (i=0; i<BUFF_SIZE; i+=2) {
    sum += buff[i];
    sum1 += buff[i+1];
}
sum += sum1;
```

7.2. Advanced OpenMP tuning

7.2.1. Iteration scheduling

OpenMP offers a set of different scheduling policies that control how iterations of a parallel loop are being scheduled to threads. This enables applications to choose a policy that best matches their needs. These policies can be specified by the programmer either with the schedule clause in the parallel loop constructs, or via the `OMP_SCHEDULE` environment variable.

The `schedule` clause has the following syntax: `schedule (type,[chunk_size])`.`type` can be one of the following:

- `static`: The iteration space is divided into chunks of size `chunk_size`, which are assigned cyclically to threads. This policy is called static because the assignment is decided at compile time. Because of this, there is no runtime overhead associated with work distribution. If `chunk_size` is not specified, the whole iteration space is evenly divided among threads.

- `dynamic`: This policy assigns dynamically chunks of contiguous iterations to threads. Whenever a threads "consumes" its chunk, it requests from the runtime system and gets assigned the next available, until no chunks remain to be distributed. In this way threads are kept always busy, system resources are utilized efficiently and the load imbalance is reduced. The chunk size is again specified with the `chunk_size` parameter. Small chunk sizes achieve better load balancing but entail larger overhead, and vice-versa. The user should experiment with different sizes to find the one that makes the best compromise between low overhead and good scalability.
- `guided`: This policy assigns initially large iteration chunks to threads, reducing them exponentially with each successive assignment. The assignment is dynamic. Due to the "geometry" of chunks allocation, guided scheduling has usually less overhead than dynamic. The minimum size of chunk size beyond of which no further reduction is allowed can be optionally specified through `chunk_size`.
- `auto`: The decision regarding scheduling is left to the compiler and/or the runtime system.
- `runtime`: The schedule type and chunk size is determined at runtime, through the `OMP_SCHEDULE` environment variable.

In Intel and PGI compilers, if no scheduling policy is specified the default behavior is to apply static scheduling with no chunk size specified. In gcc compiler, the default is dynamic scheduling with a chunk size of 1.

7.2.2. Thread affinity

Thread affinity is enforced in OpenMP using a number of generic and implementation specific environment variables.

7.2.2.1. Standard environment variables

`OMP_PROC_BIND bind` : specifies whether threads may be moved between processors. If set to true, OpenMP threads should not be moved. If set to false, the operating system may move them.

7.2.2.2. Intel environment variables

`KMP_AFFINITY` : enables the runtime system to bind OpenMP threads to CPUs. It uses the following generic syntax: `KMP_AFFINITY = [<modifier>,...] <type> [,<permute>] [,<offset>]`

The following table describes the arguments along with their most important possible values:

Argument	Value	Description
`modifier`	`granularity = <specifier>`	Granularity describes the lowest level of processor topology that OpenMP threads are allowed to float (i.e. execute unbounded) within a topology map. Practically, it refers to machines with Hyper-Threading technology enabled. When `specifier` takes the value `core` (default), then OpenMP threads that are bound to a specific core may float between the different hardware contexts (Hyper-Threads) of the core. When `specifier` is initialized to `fine` or `thread`, each OpenMP thread will be bound to a specific hardware context of the core.
	`respect`	It is the default value for `modifier`. It respects the original affinity mask of the thread that initialized the OpenMP runtime library.
	`norespect`	It does not respect original affinity mask of the initial thread. It binds OpenMP threads to all operating system processors.
	`proclist={<proc-list>}`	The user explicitly specifies OpenMP threads assignment by using a list of processor id's (as seen by the OS). It has effect when `explicit` type is specified.

Argument	Value	Description
		OS processors specified in this list are then assigned to OpenMP threads, in order of OpenMP Global Thread IDs. If more OpenMP threads are created than there are elements in the list, the assignment occurs modulo the size of the list. Examples: • `proclist=3,0-2` , and the application creates 6 threads: OpenMP thread 0 will run on processor with id 3, thread 1 on 0, thread 2 on 1, thread 3 on 2, thread 4 on 3 and thread 5 on 0. • `proclist=[3,0,{1,2},{1,2}]` , and the application creates 4 threads: OpenMP thread 0 will run on processor 3, thread 1 on 0, and threads 2 and 3 will be both allowed to float between processors 1 and 2.
	`verbose`	Prints messages concerning the supported affinity. The messages include information about the number of packages, number of cores in each package, number of thread contexts for each core, and OpenMP thread bindings to physical thread contexts.
`type`	`compact`	Assigns consecutive OpenMP threads to processors that are as close as possible to each other, in terms of processor topology. In a multi-socket, multi-core machine, it will first assign threads to all cores of a socket before proceeding to the next socket available.
	`scatter`	Distributes the threads as evenly as possible across the system, in a "breadth-first" fashion. It is the opposite of compact. In a multi-socket, multi-core machine, at first it will distribute threads on core #0 of all sockets, then on core #1, and so on.
	`explicit`	Assigns OpenMP threads to a list of processor ID's that have been explicitly specified by using the `proclist=` modifier.
	`none`	Does not bind OpenMP to particular processors.
	`disabled`	Completely disables thread affinity. Any other affinity mechanism (e.g. affinity system calls) will have no effect.
`permute`	`<positive integer value>`	Controls which levels are most significant when sorting the machine topology map. (see User Guide for a more detailed description).
`offset`	`<positive integer value>`	Indicates the starting position for thread assignment.

For an extensive description of the `KMP_AFFINITY` arguments, along with a variety of examples, the user is referred to the Intel C++ compiler Reference Guide [http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/index.htm].

7.2.2.3. Gcc environment variables

- `GOMP_CPU_AFFINITY` : enforces OpenMP thread-to-CPU mappings, in a way similar to `KMP_AFFINITY`. The variable should contain a space-separated or comma-separated list of CPUs. This list may contain different kinds of entries:

- single CPU numbers in any order,
- a range of CPUs (M-N),
- a range with some stride (M-N:S).

OS processors specified in this list are assigned to OpenMP threads, in order of OpenMP Global Thread IDs.

Examples:

- `GOMP_CPU_AFFINITY="031-24-15:2"` will bind the initial thread to CPU 0, the second to CPU 3, the third to CPU 1, the fourth to CPU 2, the fifth to CPU 4, the sixth through tenth to CPUs 6, 8, 10, 12, and 14 respectively and then start assigning back from the beginning of the list.
- `GOMP_CPU_AFFINITY=0` binds all threads to CPU 0.

If this environment variable is omitted, the host system will handle the assignment of threads to CPUs.

Note also that `GOMP_CPU_AFFINITY=<proc_list>` is equivalent to `KMP_AFFINITY= granularity=fine, proclist= [<proc_list>],explicit` .

7.2.3. Synchronization

7.2.3.1. Standard environment variables

`OMP_WAIT_POLICY policy`: defines the behaviour of threads waiting on synchronization events. When `policy` is set to `ACTIVE`, threads consume processor cycles while waiting ("busy-waiting"). When it is set to `PASSIVE`, threads do not consume CPU power but may require more time to resume.

7.2.3.2. Intel environment variables

`KMP_BLOCKTIME`: sets the time (in milliseconds) that a thread should wait, after completing the execution of a parallel region, before sleeping.

7.2.4. General tuning

7.2.4.1. Standard environment variables

`OMP_NUM_THREADS num`: sets the maximum number of threads to use in parallel regions if no other value is specified in the application. As of version 3.1 of OpenMP specification, the environment variable takes a comma-separated number list that specifies the number of threads at each nesting level.

`OMP_DYNAMIC dynamic` : enables or disables the dynamic adjustment of threads the runtime system uses for parallel regions. Valid values for `dynamic` are `true` or `false`. When enabled, the runtime system adjusts the number of threads so that it makes the most efficient use of system resources. This is meaningful when the user has not specified the desired number of threads globally or on a per-region basis.

`OMP_NESTED nested` : enables or disables nested parallelism, i.e. whether OpenMP threads are allowed to create new thread teams. Valid values for `nested` are `true` or `false`.

`OMP_MAX_ACTIVE_LEVELS levels` : sets the maximum number of nested active parallel regions.

`OMP_THREAD_LIMIT limit` : sets the maximum number of threads participating in the OpenMP program. When the number of threads in the program is not specified (e.g. with `OMP_NUM_THREADS` variable), the number of threads used is the minimum between this variable and the total number of CPUs.

`OMP_STACKSIZE size[B | K | M | G]`: sets the default thread stack size in kilobytes, unless the number is suffixed by B, K, M or G (bytes, KB, MB, GB, respectively).

7.2.4.2. Intel environment variables:

- KMP_DYNAMIC_MODE: Selects the method used to determine the number of threads to use for a parallel region when OMP_DYNAMIC=true. Possible values: (asat | load_balance | thread_limit), where
  - asat: estimates number of threads based on parallel start time;
  - load_balance (default): tries to avoid using more threads than available execution units on the machine;
  - thread_limit: tries to avoid using more threads than total execution units on the machine.
- KMP_LIBRARY: Selects the OpenMP run-time library execution mode. The options for the variable value are throughput (default), turnaround, and serial.

7.3. Advanced MPI tuning

7.3.1. IBM MPI

Normally, the poe module provides reasonable settings for the control variables used by POE. The following table gives a full explanation of these control variables; instead of setting an environment variable, an execution flag can be used.

7.3.1.1. Variable set by POE

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default

MP_CHILD	Number which is set for each task ; may be used to perform trivial parallelization (task farming) with serial jobs. If myscript contains:
	
	`./my_serial_prog`
	`<in.$MP.CHILD`
	`>out.$MP_CHILD`
	
	then
	
	`poe myscript -procs 40`
	
	would start 40 times my_serial_prog with different input and output.
		0 - num_tasks	can not be set

7.3.1.2. Partition manager control

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
MP_PROCS -procs	The number of program tasks. Only useful for interactive jobs; otherwise it is set by LoadLeveler.	Any number from 1 to the maximum supported configuration.	1
MP_NODES -nodes	To specify the number of processor nodes on which to run the parallel tasks. Only useful for interactive jobs; otherwise it is set by LoadLeveler. It may be used alone or in conjunction with MP_TASKS_PER_NODE and/or MP_PROCS,	Any number from 1 to the maximum supported configuration.	None

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
MP_TASKS_PER_NODE -tasks_per_node	To specify the number of tasks to be run on each of the physical nodes. Only useful for interactive jobs; otherwise it is set by LoadLeveler. It may be used alone or in conjunction with MP_NODES.	Any number from 1 to the maximum supported configuration.	None
MP_RETRY -retry	The period of time (in seconds) between processor node allocation retries by POE if there are not enough processor nodes immediately available to run a program. This is valid only if you are using LoadLeveler. If the character string wait is specified instead of a number, no retries are attempted by POE, and the job remains enqueued in LoadLeveler until LoadLeveler either schedules the job or cancels it.	An integer greater than or equal to 0, or the case-insensitive value wait.	0 (no retry)
MP_TIMEOUT (no associated command line flag)	The length of time that POE waits before abandoning an attempt to connect to the remote nodes.	Any number greater than 0. If set to 0 or a negative number, the value is ignored.	150 seconds
MP_HOSTFILE -hostfile -hfile	The name of a host list file for node allocation.	Any file specifier or the word NULL.	host.list in the current directory.
MP_PULSE -pulse	The interval (in seconds) at which POE checks the remote nodes to ensure that they are actively communicating with the home node.	An integer greater than or equal to 0.	600
MP_MSG_API -msg_api	To indicate to POE which message passing API is being used by the application code. • MPI: Indicates that the application makes only MPI calls. • LAPI: Indicates that the application makes only LAPI calls. • MPI_LAPI: Indicates that calls to both message passing APIs are used in the application, and the same set of communication resources (windows, IP addresses) is to be shared between them. • MPI,LAPI: Indicates that calls to both message passing APIs are used in the application, with dedicated resources assigned to each of them. • LAPI, MPI: Has a meaning identical to MPI, LAPI.	MPI | LAPI | MPI_LAPI | MPI , LAPI | LAPI , MPI	MPI

7.3.1.3. Job specification

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default

`MP_CMDFILE`
 	
`-cmdfile`
	The name of a POE commands file used to load the nodes of your partition. If set, POE will read the commands file rather than STDIN.	Any file specifier.	None

`MP_NEWJOB`
 	
`-newjob`
	Whether or not the Partition Manager maintains your partition for multiple job steps.	`yes` `no`	`no`

`MP_PGMMODEL`
 	
`-pgmmodel`
	The programming model you are using.	`spmd` | `mpmd`	`spmd`

`MP_RMFILE`
 	
`-rmfile`
	Exploit new or existing LoadLeveler (as Resource Manager) functionality that is not available using POE options. This includes specification of:
	• task geometry
	• blocking factor
	• machine order
	• consumable resources
	• memory requirements
	• disk space requirements
	• machine architecture
	• run jobs from more than 1 pool
	For more information on the LoadLeveler functionality you can exploit, refer to For more information, see LoadLeveler pages. Run parallel jobs without specifying a host file or pool, thereby causing LoadLeveler to select nodes for the parallel job from any in its cluster.
		Any relative or full path name.	None

`MP_SAVE_RMFILE`
 	
`-save_rmfile`
	When using LoadLeveler for node allocation, the name of the output LoadLeveler job command file to be generated by the Partition Manager. The output LoadLeveler job command file will show the LoadLeveler settings that result from the POE environment variables and/or command line options for the current invocation of POE. It may be saved for debugging and/or for further execution (together with `MP_LLFILE` or `-llfile`) without specifing the arguments for poe.	Any relative or full path name.	None

`MP_SAVEHOSTFILE`
 	
`-savehostfile`
	The name of an output host list file to be generated by the Partition Manager.	Any relative or full path name.	None

7.3.1.4. I/O control

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default

`MP_LABELIO`
 	
`-labelio`
	Whether or not output from the parallel tasks is labeled by task id.	`yes | no`	no. (Note: If `MP_STDOUTMODE` is set to combined, `MP_LABELIO` is automatically set to yes.)

`MP_STDINMODE`
 	
`-stdinmode`
	The input mode. This determines how input is managed for the parallel tasks.
	• `all`: All tasks receive the same input data from STDIN.
	• `none`: No tasks receive input data from STDIN; STDIN will be used by the home node only.
	• `a task id`: STDIN is only sent to the task identified.
		`all | none| taskid`	all

`MP_STDOUTMODE`
 	
`-stdoutmode`
	The output mode. This determines how STDOUT is handled by the parallel tasks.
	• `unordered`: All tasks write output data to STDOUT asynchronously.
	• `ordered`: Output data from each parallel task is written to its own buffer. Later, all buffers are flushed, in task order, to STDOUT.
	• `a task id`: Only the task indicated writes output data to STDOUT.
		`unordered | ordered | taskid`	unordered

7.3.1.5. Diagnostic information

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default

`MP_INFOLEVEL`
 	
`-infolevel`
	The level of message reporting.
	• 0: Error
	• 1: Warning and error
	• 2: Informational, warning, and error
	• 3: Informational, warning, and error. Also reports high-level diagnostic messages for use by the IBM Support Center.
	• 4, 5, 6: Informational, warning, and error. Also reports high- and low-level diagnostic messages for use by the IBM Support Center.
		0 - 6	1

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default

`MP_PRINTENV`
 	
`-printenv`
	Whether to produce a report of the current settings of MPI environment variables, across all tasks in a job. If yes is specified, the MPI environment variable information is gathered at initialization time from all tasks, and forwarded to task 0, where the report is prepared. If a script_name is specified, the script is run on each node, and the output script is forwarded to task 0 and included in the report. When a variable's value is the same for all tasks, it is printed only once. If it is different for some tasks, an asterisk (*) appears in the report after the word "Task".
	
	• no: Do not produce a report of MPI environment variable settings.
	
	• yes: Produce a report of MPI environment variable settings.
	
	• script_name: Produce the report (same as yes), then run the script specified here.
		`no | yes | scriptname`	no

`MP_EUIDEVELOP`
 	
`-euidevelop`
	Controls the level of parameter checking during execution.
	
	• yes: Setting this to yes enables some intertask parameter checking which may help uncover certain problems, but slows execution.
	
	• no or nor: Normal mode does only relatively inexpensive, local parameter checking. Setting this variable to min allows PE MPI to bypass parameter checking on all send and receive operations.
	
	• deb (debug) checking is intended for developing applications, and can significantly slow performance.
	
	• min (for minimum mode) feedback.should only be used with well tested applications because a bug in an application running with min will not provide useful error
		`yes | no | nor | deb | min`	no

`MP_STATISTICS`
 	
`-statistics`
	Provides the ability to gather communication statistics for User Space jobs.	`yes | no | print`	no

7.3.1.6. Pinning of tasks and threads

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default

`MP_PE_AFFINITY`
 	
`-pe_affinity`
	When `MP_PE_AFFINITY=yes` then MPI/POE runtime will not pass information to scheduler such as Load Leveler and it will do its own affinity setting using `MP_TASK_AFFINITY`.	`yes | no`	yes

`MP_TASK_AFFINITY`
 	
`-task_affinity`
	Setting this environment variable attaches task of a parallel job to one of the system cpusets.
	
	• `CORE` Default: specifies that each MPI task runs on a single `physical` processor core.
	
	• `CORE:n` Specifies the number of processor cores to which the threads of an MPI task are constrained (one thread per core), typically n is the number of `OMP_NUM_THREADS` (should be used for OpenMP or hybrid jobs).
	
	• `CPU` Specifies that each MPI task runs on a single logical CPU.
	
	• `CPU:n` Specifies the number of of logical CPUs to which the threads of an MPI task are constrained (one thread per cpu), typically n is the number of `OMP_NUM_THREADS`.
	
	• `MCM` Specifies that the tasks are allocated in a round-robin fashion among the sockets of a node, should only be use if not all cores of a node are not used.
	
	• `list-of-numbers` Specifies that the tasks are assigned on a round-robin basis to this set of sockets.
	
	• `-1` Specifies that no affinity request will be made (disables task affinity).
	
	`CPU` or `CORE` will generate a CPU mask with one or two entries. `CPU:n` will generatea CPU which contains all CPU but will set different for each MPI task, e.g. `KMP_AFFINITY= "proclist=[0-5],exclicit"`.
		`=core|cpu`	`core`

7.3.1.7. Tuning

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default

`MP_USE_BULK_XFER`
 	
`-use_bulk_xfer`
	This transparently causes portions of the user's virtual address space to be pinned	`yes | no`	yes

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
	and mapped to a communications adapter. The low level communication protocol will then use Remote Direct Memory Access (RDMA, also known as bulk transfer) to copy (pull) data from the send buffer to the receive buffer as part of the MPI receive.		

`MP_BULK_MIN_MSG_SIZE`
 	
`-bulk_min_msg_size`
	Contiguous messages with data lengths greater than or equal to the value you specify for this environment variable will use the bulk transfer path. Messages with data lengths that are smaller than the value, or are noncontiguous, will use packet mode transfer.Based on MPI benchmark measurements the altogether best bandwidth performance is achieved with a value of `MP_BULK_MIN_MSG_SIZE=512k`. However, the real performance will depend on the actual used message sizes inside an application and will results in higher memory consumption.	The acceptable range is from 4096 to 2147483647 (`INT_MAX`).
		
		nnnnn (byte)
		
		nnnK (kB)
		
		nnM (MB)
		
		nnG (GB)
			64K

`MP_BUFFER_MEM`
 	
`-buffer_mem`
	To control the amount of memory MPI allows for the buffering of early arrival message data. Message data that is sent without knowing if the receive is posted is said to be sent eagerly. If the message data arrives before the receive is posted, this is called an early arrival and must be buffered at the receive side. See Can also be specified in the form `MP_BUFFER_MEM=M1,M2`, where `M1` specifies the amount of preallocated memory. `M2` specifies an upper bound on the amount of early arrival buffer memory.	nnnnn (byte)
		
		nnnK (kB)
		
		nnM (MB)
		
		nnG (GB)
			64 MB (User Space and IP)

`MP_EAGER_LIMIT`
 	
`-eager_limit`
	To change the threshold value for message size, above which rendezvous protocol is used. To ensure that at least 32 messages can be outstanding between any two tasks, `MP_EAGER_LIMIT` will be adjusted based on the number of tasks according to the following table, when the user has specified neither `MP_BUFFER_MEM` nor `MP_EAGER_LIMIT`:
	
	Number of Tasks / MP_EAGER_LIMIT
	
	0001 to 0256 — 32768
	0257 to 0512 — 16384
	0513 to 1024 — 8192
		nnnnn	65536

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
	1025 to 2048 4096 2049 to 4096 2048 4097 to 8192 1024 MPI uses the `MP_BUFFER_MEM` and the `MP_EAGER_LIMIT` values that are selected for a job to determine how many complete point-to-point messages, each with a size that is equal to or less than the eager_limit, can be sent eagerly from every task of the job to a single task, without causing the single target to run out of buffer space. This is done by allocating to each sending task a number of message credits for each target. The sending task will consume one message credit for each eager send to a particular target. It will get that credit back after the message has been matched at that target. The following equation is used to calculate the number of credits to be allocated: `MP_BUFFER_MEM / (MP_PROCS * MAX(MP_EAGER_LIMIT, 64))` MPI uses this equation to ensure that there are at least two credits for each target. If needed, MPI reduces the initially selected value of `MP_EAGER_LIMIT`, or increases the initially selected value of `MP_BUFFER_MEM`, in order to achieve this minimum threshold of two credits for each target.		

`MP_CC_BUF_MEM`
 	
`-cc_buf_mem`
	Specifies the size of the Early Arrival (EA) buffer that isused by the communication subsystem to buffer eager sendmessages, for collective communications operations, that arrivebefore there is a matching receive posted.	nnnnn (byte)
		
		nnnK (kB)
		
		nnM (MB)
		
		nnG (GB)	

`MP_CC_SCRATCH_BUF`
 	
`-cc_scratch_buf`
	Use the fastest collective communication algorithm even if that algorithm requires allocation of more scratch buffer space.	`yes | no`	yes

`MP_SINGLE_ THREAD`
 	
`-single_thread`
	To avoid lock overheads in a program that is known to be single-threaded. *MPI-IO and MPI one-sided communications are unavailable if this variable is set to yes.* Results are undefined if this variable is set to yes with multiple application message passing threads in use. See [IBM Parallel Environment: MPI Programming Guide] for more information.	yes no	yes

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
MP_WAIT_MODE -wait_mode	Set: to specify how a thread or task behaves when it discovers it is blocked, waiting for a message to arrive. MPI_WAIT_MODE set to nopoll may reduce CPU consumption for applications that post a receive call on a separate thread, and that receive call does not expect an immediate message arrival. Also, using MPI_WAIT_MODE set to nopoll may increase delay between message arrival and the blocking call's return. It is recommended that MP_CSS_INTERRUPT be set to yes when the nopoll wait is selected, so that the system wait can be interrupted by the arrival of a message. Otherwise, the nopoll wait is interrupted at the timing interval set by MP_POLLING_INTERVAL.	nopoll | poll | sleep | yield	poll (for User Space and IP)
MP_POLLING_ INTERVAL -polling_interval	To change the polling interval (in microseconds).	An integer between 1 and 2 billion.	400000
MP_CSS_INTERRUPT -css_interrupt	User Space is an unreliable packet transport (packets may be dropped during transport without an error being reported), the message dispatcher manages packet acknowledgment and retransmission with a *sliding window protocol*. This message dispatcher is also run on a hidden thread once every few hundred milliseconds and, if environment variable MP_CSS_INTERRUPT is set, upon notification of packet arrival. To specify whether or not arriving packets generate interrupts. Using this environment variable may provide better performance for certain applications. Setting this variable explicitly will suppress the MPI-directed switching of interrupt mode, leaving the user in control for the rest of the run. For more information, refer to the MPI_FILE_OPEN and MPI_WIN_CREATE subroutines in [IBM Parallel Environment: MPI Subroutine Reference].	yes | no	no

7.3.1.8. Advanced tuning parameters

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
MP_HINTS_FILTERED -hints_filtered	To specify whether or not MPI info objects reject hints (key and value pairs) that	yes | no	yes

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
	are not meaningful to the MPI implementation.		
MP_IONODEFILE -ionodefile	To specify the name of a parallel I/O node file, a text file that lists the nodes that should be handling parallel I/O. Setting this variable enables you to limit the number of nodes that participate in parallel I/O and guarantees that all I/O operations are performed on the same node.	Any relative path name or full path name.	None. All nodes will participate in parallel I/O.
MP_MSG_ ENVELOPE_BUF -msg_envelope_buf	The size of the message envelope buffer (that is, uncompleted send and receive descriptors).	Any positive number. There is no upper limit, but any value less than 1 MB is ignored.	8 MB
MP_RETRANSMIT_ INTERVAL -retransmit_ interval	Control how often the communication subsystem library checks to see if it should retransmit packets that have not been acknowledged. The value nnnn is the number of polling loops between checks.	The acceptable range is from 1000 to INT_MAX.	400000 (User Space)
MP_THREAD_ STACK-SIZE -thread_stacksize	To specify the additional stack size allocated for user subroutines running on an MPI service thread. If you do not allocate enough space, the program may encounter a SIGSEGV exception or more subtle failures.	bytes	0
MP_TIMEOUT (no associated command line flag)	To change the length of time (in seconds) the communication subsystem will wait for a connection to be established during message-passing initialization.	An integer greater than 0.	150
MP_ACK_THRESH -ack_thresh	Allows the user to control the packet acknowledgement threshold. Specify a positive integer.	A positive integer limited to 31.	30
MP_IO_BUFFER_SIZE -io_buffer_size	To specify the default size of the data buffer used by MPI-IO agents. nnnn nnnK (where K=1024 bytes) nnnM (where M=1024*1024 bytes)	bytes	The number of bytes that corresponds to 16 file blocks.
MP_IO_ERRLOG -io_errlog	To specify whether or not to turn on I/O error logging.	yes | no	no
MP_REXMIT_BUF_ SIZE -rexmit_buf_size	The maximum LAPI level message size that will be buffered locally, to more quickly free up the user send buffer. This sets the size of the local buffers that will be allocated to store such messages, and will impact memory usage, while potentially improving performance. The MPI application message size supported is smaller by, at most, 32 bytes.	nnn bytes (where: nnn > 0 bytes)	16352 bytes

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
MP_REXMIT_BUF_CNT -rexmit_buf_cnt	The number of retransmit buffers that will be allocated per task. Each buffer is of size MP_REXMIT_BUF_SIZE * MP_REXMIT_BUF_CNT. This count controls the number of in-flight messages that can be buffered to allow prompt return of application send buffers.	nnn (where: nnn > 0)	128

7.3.1.9. Gprof and corefile handling

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
MP_PROFDIR -profdir	Allows you to specify the directory into which POE stores the gmon.out file for each task. A gmon.out file contains profiling data and is produced by compiling a program with the -pg flag.	Any relative path name or full path name.	profdir.task_id
MP_COREFILE_FORMAT -corefile_format	POE processes that terminate abnormally will can generate standard corefiles. If you prefer, you can instruct POE to write the stack trace information to standard error instead. It is highly recommended to limit the amount of information written for large parallel application by setting it to STDERR.	noset | STDERR	STDERR
MP_COREDIR -coredir	Creates a separate directory for each task's core file.	Any valid directory name, or "none" to bypass creating a new directory.	coredir.taskid

7.3.1.10. Other

Environment Variable / Command Line Flag(s)	Item	Possible Values	Default
MP_FENCE (no associated command line flag)	A fence character string for separating arguments you want parsed by POE from those you do not.	Any string.	None
MP_NOARGLIST (no associated command line flag)	Whether or not POE ignores the argument list. If set to yes, POE will not attempt to remove POE command line flags before passing the argument list to the user's program.	yes | no	no

7.3.2. Intel MPI

The Intel MPI Library provides the mpitune tuning utility to help the user automatically select optimal values for many environment variables that can be used to influence program behavior and performance at run time.

The mpitune utility can be used to create a set of configuration files that contain optimal settings for a particular cluster or application. These configuration files can be reused in the mpirun job launcher using the -tune option. If configuration files from previous mpitune sessions exist, mpitune creates a copy of the existing files before starting execution.

The mpitune utility operates in two modes:

- Cluster-specific, evaluating a given cluster environment using either the Intel MPI Benchmarks or a user-provided benchmarking program to find the most suitable configuration of the Intel MPI Library. This mode is used by default, and should be run once after the Intel MPI Library installation and after every cluster configuration change (processor or memory upgrade, network reconfiguration, etc.).
- Application-specific, evaluating the performance of a given MPI application to find the best configuration for the Intel MPI Library for the particular application. Application tuning is enabled by the `--application` command line option.

Example of application-specific tuning:

1. Collect configuration settings for the given application:

```
$ mpitune --application \";mpiexec -n 32 ./myprog\" --output-file
                                                        ./myprog.conf
```

2. Use the optimal recorded values for the user application:

```
$ mpiexec --tune ./myprog.conf -n 32 ./myprog
```

Based on the default tuning rules, the automated tuning utility evaluates a full set of the library configuration parameters to minimize the application execution time. By default, all generated files will be saved in the current working directory.

Details about the supported environment variables can be found in the Intel MPI Library Reference Manual http://software.intel.com/sites/products/documentation/hpc/mpi/linux/reference_manual.pdf

7.4. Hybrid programming

7.4.1. General

Hybrid MPI-OpenMP programming is a natural approach in large-scale hierarchical machines, where MPI is used for communication between nodes and OpenMP for parallelization within a node. In its most typical form, there are a few MPI processes running in each node, and within each process multiple OpenMP threads operate in parallel on process's shared data. Usually, there are no MPI calls inside parallel regions. An example which corresponds to this scenario is shown below:

```
while ( i <  iters ) {
   MPI_Send(prev_data);
   MPI_Recv(curr_data);
   #pragma omp parallel for
   {
   /* process in parallel curr_data
   */
   }
   ...
}
```

Shifting from a pure-MPI scheme to a hybrid MPI-OpenMP one with fewer MPI processes per node, has a number of advantages:

- the MPI overhead for intra-node communication (essentially, message packing / unpacking) is eliminated; communication between OpenMP threads is performed solely through shared memory,
- the memory footprint is reduced, due to the reduction of message buffers and various bookkeeping structures internal to the MPI library. This allows the application to execute with larger working sets,
- the pressure on the network is mitigated, due to the reduction of communication channels and total number of messages. It is often more beneficial to have fewer but larger messages being exchanged, than the opposite,
- MPI calls are reduced,
- load balancing can be improved when utilizing OpenMP dynamic work-sharing constructs. This would be hard to implement with pure MPI,
- under several cases, cache sharing in multi-core nodes may be exploited by OpenMP threads in a beneficial way. For example, when threads work under common data, they effectively have larger cache space available or opportunities for mutual data prefetching. This would not be possible in pure MPI implementations, where each MPI process works on its private address space.

In the opposite direction, moving from a pure-OpenMP configuration to a hybrid one, has the following basic advantage:

- naturally provides OpenMP threads grouping on a per-MPI process basis, which in most cases translates to NUMA-aware execution. When MPI processes are initially created, they can be easily mapped (by the OS or the user) to different sockets/NUMA nodes. The OpenMP threads that each process will later spawn, can be restricted to run on any core of the process's socket. This clustering of threads in a fashion that resembles the underlying topology, guarantees:
  - uncontended access to main memory from the MPI processes and their threads, since they are mapped to different NUMA nodes,
  - fast communication between OpenMP threads in a team and their "parent" MPI process, since they are all located under the same NUMA node, and possibly share some level of cache hierarchy,
  - minimization of (expensive) inter-socket communication, since OpenMP threads of a specific team are not spread across multiple NUMA nodes.

7.4.2. Optimal MPI processes/OpenMP threads configuration

Determining the optimal configuration and placement of MPI processes and OpenMP threads in the hybrid model is not trivial. In order to achieve best performance, the programmer needs to target at reducing the following overheads when examining a specific processes/threads combination:

- MPI communication overhead,
- synchronization overhead,
- memory consumption,
- load imbalance.

Let C be the total number of cores in a system node. We consider the following alternatives regarding the processes/threads configuration, denoted by P x T, where P is the number of MPI processes and T the number of threads. We also assume that, in all cases, there is no over- or under-subscription of a node, i.e. P x T = C.

Cx1 : This corresponds to a pure MPI implementation. It treats all MPI processes as peers, disregarding the fact that some cores (i.e., those in the same node) can use the shared-memory infrastructure to communicate. This scheme can be easily adopted by an application, without requiring major code refactoring. Its drawback is the MPI

overhead (e.g. message processing) that must be paid for intra-node communication. In practice, the MPI library will optimize this case and force communication to happen via shared memory, in a way that is transparent to the programmer. However, the user lacks the flexibility offered by the shared memory programming model, and the ability to leverage certain key features of OpenMP (e.g. automatic work scheduling and load balancing).

1xC : This corresponds to an (almost) pure OpenMP implementation, where a single MPI process does all the communication and the C OpenMP threads the computations. One drawback of this strategy is that all OpenMP threads need to wait while the MPI process communicates data. Depending on the application, this communication overhead could quickly become the limiting factor of scalability. Furthermore, the total inter-node network bandwidth would remain under-utilized. A possible way to overcome both problems would be to overlap computation with communication, by assigning some OpenMP threads to perform communication in parallel with computation. For many applications, however, this approach is not so straightforward to implement. Another possible problem with this scheme is when two or more threads tend to work on the same data, following a read-write pattern. In this case the shared data must travel between the threads' private caches. If the two threads share e.g. the L2 cache, then the communication will be rather fast. However, if they are bound to different sockets which do not share any level of cache, then the communication will be relatively slow since it has to go off-chip. The problem becomes more intense as the number of sharers, or their distance in terms of processor topology, increases. Under several cases (e.g. with a more judicious placement of threads on cores) the negative effects of read-write sharing can be mitigated.

NxM : This approach lies between the two previous extremes, and represents the most typical case for the hybrid model. There are N MPI processes, each of which is paired with a team of M OpenMP threads. Again, the MPI processes do all the communication and the OpenMP threads the computation, as shown in the code snippet in the previous section. The literature has shown that, in most cases, this configuration best balances the pros and cons of Cx1 and 1xC schemes and yields optimal performance.

There is no rule-of-thumb for choosing the best configuration for N x M scheme (i.e., the best values for N and M). However, especially for NUMA systems, an option that matches well the hierarchical memory subsystem and the hierarchical nature of the hybrid MPI-OpenMP model, is to use as many MPI processes as the number of NUMAdomains (i.e., the NUMA memory node together with its local processor socket), and as many OpenMP threads (per-MPI process) as the number of cores in each domain. This choice promotes data sharing between cores of the same NUMA domain (fast), avoids data exchange between cores on different NUMA domains (slow), and at the same time offers all the advantages of the NxM scheme discussed so far.

For example, in a SuperMUC thin node which features 2 NUMA domains (sockets + memory modules) with 8 cores each, a reasonable configuration would be to use 2 MPI processes with 8 OpenMP threads each. Of course, since different applications have different demands and characteristics (in terms of communication, synchronization, locality, load balancing, etc.), it would be advisable for the user to test additionally "neighboring" configurations (e.g., 4x4, 1x16, etc.) in order to find the one that performs best. In any case, the total number of threads should be equal to the total number of cores in the node, i.e. 16.

7.4.3. Enforcing processor/memory affinity

An arbitrary placement of threads on cores may not take advantage of NUMA capabilities. For example, a pathologic situation is when threads have disjoint working sets but all of them are allocated on a single memory node. This can lead to bus saturation as all threads contend for the same bus. Instead, a much better approach would be to evenly distribute data sets across memory nodes, and then schedule each thread to the core that is local to the memory node containing its data set. In such cases, performance problems can be solved if the programmer takes care of the thread-to-processor mappings using appropriate affinity calls.

In order to map processes and threads on the underlying architecture according to a certain configuration, it is necessary to enforce processor and memory affinity to the application. By processor affinity we mean the set of system processors where a process/thread is allowed to run, and by memory affinity the set of NUMA nodes where it can allocate memory from.

The (hybrid) POE module pins threads to cores per default. For fine-tuning the taskset utility can be used for processor affinity, and the numactl utility for memory (and processor) affinity. Alternatively, implementation-specific OpenMP runtime variables may be used to control processor affinity of threads.

8. Debugging

8.1. Debuggers with graphical interface (GUI)

- DDT [http://www.lrz.de/services/software/programmierung/ddt/]: Distributed Debugging Tool: a commercial product by Allinea Software.
- Totalview [http://www.lrz.de/services/software/programmierung/totalview/]: A commercial product by Etnus.

The GUI driven debuggers offer a graphical user interface; simple debugging sessions can therefore be handled without intensive, prior study of man-pages and manual.

- DDT and Totalview are advanced tools for more complex debugging, especially when it comes to debugging parallel codes (MPI, OpenMP). They allow to inspect data structures in the different threads of a parallel program, set global breakpoints, set breakpoints in individual threads, etc.
- DDT is the preferred debugger at SuperMUC, and the largest number of licences is available.
- Totalview can also be used in CLI mode, whereas *DDT* is a pure GUI tool.

8.2. Table of available debuggers and info

Please note, that the environment variables given in the column *Documentation* (e.g. `$TOTALVIEW_DOC`) refer to environment variables set by the `module` command on the LRZ HPC systems.

Name	**Interface**	**Supported Compilers**	**Programming Model**	**LRZ Module**
DDT	GUI	g77, g95, icc, ifort	serial, parallel (MPI, OpenMP)	`module load ddt`
Totalview	GUI, CLI	g77, icc, ifort	serial, parallel (MPI, OpenMP)	`module load totalview`

Other debuggers like gdb, idb or DDD are available, but they can hardly be used on the compute nodes.

9. Further documentation

9.1. IBM Parallel Operating Environment: POE

- Parallel Environment for AIX and Linux - Operation and Use: PDF [http://publib.boulder.ibm.com/epubs/pdf/c2366673.pdf]
- Parallel Environment for AIX and Linux - Messages: PDF [http://publib.boulder.ibm.com/epubs/pdf/c2366693.pdf]
- Parallel Environment for AIX and Linux - MPI Programming Guide: PDF [http://publib.boulder.ibm.com/epubs/pdf/c2366704.pdf]
- Parallel Environment for AIX and Linux - MPI Subroutine Reference: PDF [http://publib.boulder.ibm.com/epubs/pdf/c2366713.pdf]

9.2. IBM LoadLeveler

- Tivoli Workload Scheduler LoadLeveler - Using and Administering: PDF [http://publib.boulder.ibm.com/epubs/pdf/c2367920.pdf]

- Tivoli Workload Scheduler LoadLeveler - Diagnosis and Messages Guide: PDF [http://publib.boulder.ibm.com/epubs/pdf/c2367930.pdf]
- Tivoli Workload Scheduler LoadLeveler - Command and API Reference: PDF [http://publib.boulder.ibm.com/epubs/pdf/c2367940.pdf]
- Tivoli Workload Scheduler LoadLeveler - Resource Manager: PDF [http://publib.boulder.ibm.com/epubs/pdf/c2367900.pdf]

9.3. IBM Central

- High Performance Computing Central is a joint IBM/Customer accessible and editable forum to provide improved HPC technical communications: HTML [https://www.ibm.com/developerworks/wikis/display/hpccentral/HPC+Central]

9.4. Intel compiler, libraries and tools

- Search for a Manual [http://software.intel.com/sites/products/search]

9.5. Intel optimization and tuning

- Quick-Reference Guide to Optimization with Intel Compilers [http://software.intel.com/sites/products/collateral/hpc/compilers/compiler_qrg12.pdf]
- Compiling for AVX: PDF [http://software.intel.com/file/34217/]
- Guide to Auto-Vectorization [http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/]
- Requirements for Vectorizable Loops [http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/]
- Intel 64 and IA-32 Architectures Optimization Reference Manual: PDF [http://www.intel.com/Assets/PDF/manual/248966.pdf]
- Developing Multithreaded Applications: A Platform Consistent Approach: PDF [http://software.intel.com/file/25018]
- How To Optimize Your Software For The Upcoming Intel Advanced Vector Extensions (Intel AVX)): PDF [http://software.intel.com/file/32266]
- Code Coverage and Test Prioritization Tools: PDF [http://software.intel.com/sites/products/collateral/hpc/compilers/code_coverage_tool.pdf]
- Consistency of Floating-Point Results Using the Intel Compiler: PDF [http://software.intel.com/en-us/articles/consistency-of-floating-point-results-using-the-intel-compiler/]

9.6. Intel 64 and IA-32 architectures software developer's manuals

- Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture: PDF [http://www.intel.com/Assets/PDF/manual/253665.pdf]
- Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A and 2B: Instruction Set Reference, A-Z: PDF [http://www.intel.com/Assets/PDF/manual/325383.pdf]
- Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A and 3B: System Programming Guide, Parts 1 and 2: PDF [http://www.intel.com/Assets/PDF/manual/325384.pdf]