Dynamical Exascale Entry Platform

Grant Agreement Number: 287530

D5.1
Prototype programming environment in Booster Node

Approved
Prototype programming environment in Booster Node

Project and Deliverable Information Sheet

<table>
<thead>
<tr>
<th>DEEP Project</th>
<th>Project Ref. №: 287530</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Title: Dynamical Exascale Entry Platform</td>
<td></td>
</tr>
<tr>
<td>Project Web Site: <a href="http://www.deep-project.eu">http://www.deep-project.eu</a></td>
<td></td>
</tr>
<tr>
<td>Deliverable ID: D5.1</td>
<td></td>
</tr>
<tr>
<td>Deliverable Nature: Report</td>
<td></td>
</tr>
<tr>
<td>Deliverable Level: PU*</td>
<td></td>
</tr>
<tr>
<td>Contractual Date of Delivery: 30/11/2012</td>
<td></td>
</tr>
<tr>
<td>Actual Date of Delivery: 30/11/2012</td>
<td></td>
</tr>
<tr>
<td>EC Project Officer: Luis Carlos Busquets Pérez</td>
<td></td>
</tr>
</tbody>
</table>

* - The dissemination level are indicated as follows: PU – Public, PP – Restricted to other participants (including the Commission Services), RE – Restricted to a group specified by the consortium (including the Commission Services). CO – Confidential, only for members of the consortium (including the Commission Services).

Document Control Sheet

<table>
<thead>
<tr>
<th>Document</th>
<th>Title: Prototype programming environment in Booster Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID: D5.1</td>
<td>Version: 2.0 Status: Approved</td>
</tr>
<tr>
<td>Available at: <a href="http://www.deep-project.eu">http://www.deep-project.eu</a></td>
<td></td>
</tr>
<tr>
<td>Software Tool: Microsoft Word</td>
<td></td>
</tr>
<tr>
<td>File(s): DEEP_D5.1_Prototype_programming_environment_in_BN_v2.0-ECapproved</td>
<td></td>
</tr>
<tr>
<td>Written by: V. Beltran (BSC), J. Labarta (BSC)</td>
<td></td>
</tr>
<tr>
<td>Contributors: D. Alvarez (JSC), A. Jakobs (JSC)</td>
<td></td>
</tr>
<tr>
<td>Reviewed by: H. Merx (CYI), E. Suarez (JUELICH)</td>
<td></td>
</tr>
<tr>
<td>Approved by: BoP/PMT</td>
<td></td>
</tr>
</tbody>
</table>

DEEP - 287530 2 25.02.2013
## Document Status Sheet

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>16/November/2012</td>
<td>Draft</td>
<td>Reviewed by PMT</td>
</tr>
<tr>
<td>0.2</td>
<td>16/November/2012</td>
<td>Draft</td>
<td>Change the order of sections</td>
</tr>
<tr>
<td>0.23</td>
<td>26/November/2012</td>
<td>Draft</td>
<td>Addressed review comments</td>
</tr>
<tr>
<td>0.3</td>
<td>26/November/2012</td>
<td>Draft</td>
<td>Review libraries part</td>
</tr>
<tr>
<td>1.0</td>
<td>30/November/2012</td>
<td>Final</td>
<td>EC submission</td>
</tr>
<tr>
<td>2.0</td>
<td>25/February/2013</td>
<td>Approved</td>
<td>Approved by EC</td>
</tr>
</tbody>
</table>
Document Keywords

| Keywords: | DEEP, HPC, Exascale, MIC, OmpSs, linear algebra, Global MPI |

Copyright notices

© 2011-2012 DEEP Consortium Partners. All rights reserved. This document is a project document of the DEEP project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the DEEP partners, except as mandated by the European Commission contract 287530 for reviewing and dissemination purposes.

All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.
Table of Contents

Project and Deliverable Information Sheet ................................................................. 2
Document Control Sheet ......................................................................................... 2
Document Status Sheet ......................................................................................... 3
Document Keywords ............................................................................................. 4
Table of Contents ................................................................................................. 5
List of Figures ....................................................................................................... 5
List of Tables ......................................................................................................... 5
Executive Summary .............................................................................................. 6
1 DEEP programming environment ...................................................................... 7
2 Task offload ....................................................................................................... 7
3 Description of OmpSs ...................................................................................... 10
4 OmpSs on the Xeon Phi .................................................................................... 13
   4.1 Performance evaluation ............................................................................ 13
5 Linear Algebra Libraries on Xeon Phi™ ............................................................ 16
   5.1 Performance tests on KNC ...................................................................... 16
References and Applicable Documents ................................................................. 20
List of Acronyms and Abbreviations .................................................................... 21

List of Figures

Figure 1. OmpSs version of Cholesky decomposition .............................................. 11
Figure 2. Mercurium dataflow diagram ................................................................. 12
Figure 3. Task dependency-graph of Cholesky ...................................................... 12
Figure 4: Cholesky scalability on KNF (OmpSs vs MKL) ....................................... 14
Figure 5: Trace file of a Cholesky execution phase using OmpSs ......................... 14
Figure 6. Trace file of Cholesky (OmpSs with nesting) with 32, 48, 64, 80 and 96 hardware threads. 15
Figure 7: Performance of MKL's DGEMM in KNC with matrices sizes of 7808x7808 elements .......................................................... 17
Figure 8: Performance of MKL's DGEMM in KNC with matrices sizes of 15616x15616 elements ... 18
Figure 9: Performance of MKL's DGEMM in KNC with 61 threads .................... 19
Figure 10: Performance of MKL's DGEMM in KNC with 244 threads .................. 19
Figure 11: Performance of MKL's DGEMM in Sandy Bridge with 32 threads. Results with 16 threads are very similar .............................................................. 19

List of Tables

Table 1: Simple offload example ........................................................................ 8
Table 2: Helper function to allocate BNs ............................................................. 8
Table 3: Traditional MPI application .................................................................. 8
Table 4: Individual offload of tasks .................................................................. 9
Table 5: Collective offload of MPI tasks ......................................................... 9
Table 6: Plain MPI code used on the previous example .................................. 10
Executive Summary

The DEEP project is intended to develop a novel supercomputer architecture that radically departs from existing approaches. While traditional supercomputers are homogeneous at the cluster level but can be heterogeneous inside each node by using some kind of accelerator, our approach targets a heterogeneous supercomputer composed of two interconnected clusters of Cluster Nodes (CN) and Booster Nodes (BN), both of them homogeneous. The CNs are traditional compute nodes that can collectively offload computational kernels to the BNs, which form a cluster of accelerators. The objective of this deliverable was to provide a prototype of the programming environment, including a runtime system and linear algebra libraries, which will be used to develop the kernels that will run on the BNs.

The BNs are based on the Intel Knights Corner (KNC) accelerator, also known as Intel Xeon Phi™. Our current KNC samples are composed of 61 cores, each one with four hardware-threads that can access up to 8GB of RAM. The accelerator is based on the standard x86-64 instruction set and provides a familiar software stack (Linux OS, MPI libraries, OpenMP runtime, linear algebra libraries, etc) that eases the development and porting of applications compared to other accelerators like GPUs. However, to make the most of this accelerator the whole software stack must be optimized to run on it, from the operating system and communication libraries to the runtime system and linear algebra libraries.

To obtain good performance on the KNC the direct use of the new vector instruction-set or the use of the highly optimized libraries provided by Intel is mandatory. Moreover, the programming model must expose a large amount of parallelism to allow the runtime systems to efficiently exploit the large number of hardware threads available. Hence, the runtime system has to efficiently manage fine-grained parallelism to make the most of this accelerator.

This document introduces the DEEP programming model, and its components, including performance results of the OmpSs runtime, which is already ported to the Booster Node architecture, i.e. MIC. First of all, the draft of the high level Cluster/Booster offload model, designed for the DEEP architecture as a combination of MPI and OmpSs, is described (a detailed description of the DEEP programming model will be given in deliverable D5.3, due in month 24). The OmpSs[1] programming model and its tool-chain, which is composed of three main components: the Mercurium compiler, the Nanos++ runtime system and the Extrace[3,4] instrumentation library is presented. The potential of the OmpSs programming model to provide an adequate amount of fine-grained parallelism and the performance and scalability of the Nanos++ runtime to efficiently run on the Xeon Phi are evaluated. The performance of hybrid MPI and OmpSs applications is also evaluated. Finally, this Deliverable presents a preliminary performance evaluation of the MKL library, comparing results obtained on the MIC with those measured on a Xeon (Sandy Bridge) multi-core processor.
1 DEEP programming environment

The software developers from WP4, WP5, and WP8 have engaged in fruitful discussions in the last months, to define the details of the middleware that will be installed on the DEEP System. An agreement on the structure of the DEEP programing environment and the way that its two main elements – OmpSs and ParaStationMPI – will interact with each other has been found. ParastationMPI will provide a standard MPI interface which seamlessly works on the Cluster and Booster at the same time. This MPI implementation will include the dynamic spawning of MPI processes, which will be used to offload the highly scalable parts of the applications from the Cluster to the Booster. On top of the ParastationMPI implementation, the OmpSs programming model will provide the developers with pragmas to ease the offloading process, hiding the complexity of the low-level API.

2 Task offload

The offload of high performance kernels written in MPI is a key feature to effectively exploit the DEEP Architecture. Current MPI implementations already support the $\texttt{MPI}\_\texttt{comm}\_\texttt{spawn(...)}$ primitive to dynamically create new MPI processes on arbitrary nodes. This collective operation creates an intra-communicator that connects the set of nodes that call it, with the new set of MPI processes created on the specified remote nodes. It is worth noting that this MPI primitive supports the creation of new MPI processes even on nodes with a different architecture, which makes it especially well-suited for the DEEP Architecture. However, this function can dramatically increase the complexity of a MPI application because the programmer has to coordinate and manage two or more sets of parallel MPI processes, explicitly sending the required data from side to side. This additional complexity makes this technique cumbersome or even unfeasible on large and complex applications such as the ones targeted by DEEP.

To overcome the above-mentioned issues, OmpSs will be extended to support a flexible approach to offload MPI kernels from the Cluster to the Booster. The goal is to ease, as much as possible the mechanism to offload MPI kernels, but leveraging the current MPI infrastructure. To that end, we have designed an offload model that uses #pragmas to mark the MPI functions that must be offloaded. The compiler and the runtime systems will cooperate to transparently manage all the data transfers between the MPI processes on the Cluster and MPI processes on the Booster.

This pragmatic approach will leverage both the unmodified MPI kernels of the application and the optimized communication infrastructure provided by the MPI library. Our runtime will use under the hood the $\texttt{MPI}\_\texttt{comm}\_\texttt{spawn(...)}, \texttt{MPI}\_\texttt{comm}\_\texttt{send()}, \texttt{etc}.$ functions to transfer the required data from the Cluster to the Booster and vice-versa. Our approach will increase the malleability of the applications and have the potential to apply advanced optimizations using a transparent directory/cache implemented on the runtime system.

An example of this technique is shown on Table 1 where a set of MPI processes collectively offload a set of MPI tasks. An MPI task is a function that contains calls to MPI. The mpi_comm is a valid MPI communicator and rank is an integer that represents a valid rank inside the communicator. The MPI-task marked with this pragma will be offloaded to a MPI process that runs on the node identified by mpi_comm:rank.
D5.1 Prototype programming environment in Booster Node

int main(int argc, char **argv){
    MPI_Init ( &argc, &argv );
    int rank, size, remote_rank, res;
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    MPI_Comm_size ( MPI_COMM_WORLD, &size );
    remote_rank = (rank + 1) % size;
    char string[100];
    sprintf(string, “created on node %d!\n”, rank);
    // foo(...) will run on the remote MPI process identified by comm:rank
    #pragma target device(MPI_COMM_WORLD:remote_rank) copy_deps
    #pragma omp task input(string) output(*res)
    foo(string, &res);
    #pragma omp taskwait
    assert(rank == res);
    return 0;
}

void foo(char * input, int *res){
    MPI_Comm_rank ( MPI_COMM_WORLD, res );
    printf(“Executed on node[%d], but %s\n”, res, input);
}

Table 1: Simple offload example

A helper function, specified in Table 2, to allocate BNs will also be provided. This function will use the MPI_comm_spawn(…) to implement this functionality.

// booster_nodes => number of booster nodes
// booster_np => number of MPI processes
int DEEP_Booster_alloc(MPI_Comm cluster_comm, int booster_nodes,
                        int booster_np, MPI_Comm *intercomm);

Table 2: Helper function to allocate BNs

The rest of this section will present a set of examples to illustrate and clarify how the offload model will work. Table 3 presents a traditional MPI application that performs a Jacobi computation.

MPI_Init ( &argc, &argv );
MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
MPI_Comm_size ( MPI_COMM_WORLD, &ntasks);
size_t BS = SIZE/ntasks;
double vec0[BS+2], vec1[BS+2], res;
initialize_1d(BS, vec0, rank);
jacobi_1d(BS, vec0, vec1, res);
printf(“res: %d\n”, res);
MPI_Finalize ( );

Table 3: Traditional MPI application

The usual calls to MPI_Init(…), MPI_Comm_rank(…) and MPI_Comm_size(…) are used to initialize the MPI library and find out the rank and number of MPI processes, respectively. Then each MPI process initializes its data with a call to initialize_1d() function and finally a call to jacobi_1d() performs the actual computation. The code of the MPI jacobi_1d() function is provided on Table 6. The figure associated to Table 3 shows a diagram representing the execution of the example on two CNs.

The next example, presented in Table 4, uses the DEEP_Booster_alloc(…) to allocate one BN for each CN. The MPI communicator used to collectively offload is MPI_COMM_SELF, hence each MPI process will individually allocate one BN. Then the #pragma target device
D5.1 Prototype programming environment in Booster Node

(my_comm:0) specifies that the next task have to be offloaded to the MPI process identified by (my_comm:0). The initialize_1d(...) function is annotated with standard OmpSs pragmas to specify input and output parameters and will be offloaded to the BN. The jacobi_1d(...) task is also annotated, however it will run on the CN, once the results of the initialize_1d(...) task become ready.

The example in Table 5 uses the DEEP_Booster_alloc(...) to allocate two BNs for two CNs. The MPI communicator used to collectively offload is MPI_COMM_WORLD, hence all the MPI processes (two in this example) will collectively allocate two BNs. The initialize_1d(...) task will run on the CN, while each MPI processes on the CN will offload the jacobi_1d(...) task to a BN with the pragma #pragma target device (my_comm:rank). It is worth noting that the jacobi_1d(...) will use the interconnection network of the BNs to perform the computation.

Table 5: Collective offload of MPI tasks

The examples provided are deliberately kept as simple as possible to facilitate the description of the offloading model proposed. However, it is worth noting, that the proposed extensions to OmpSs supports any offloading pattern that can be expressed with the low-level MPI spawning mechanism.
void jacobi_1d(size_t size,  
    double vec0[size+2],  
    double vec1[size+2],  
    double * res) {  
    double *src = vec0, *dst = vec1;  
    double residual;  
    do{  
        update_halo_1d_mpi(size, src);  
        double l_residual = 0.0;  
        stencil_1d(size, src, dst, &l_residual);  
        residual = 0.0;  
        MPI_Allreduce (&l_residual,&residual,  
            1, DOUBLE, SUM, MPI_COMM_WORLD);  
        src = src == vec0 ? vec1 : vec0;  
        dst = dst == vec0 ? vec1 : vec0;  
    } while(fabs(residual) > DELTA);  
    *res = residual;  
    if(src != vec1){  
        memcpy(vec1, vec0, sizeof(vec1));  
    }  
}  

void update_halo_1d_mpi(int N, double vec[N+2]){  
    MPI_Request requests[4] = { 0, 0, 0, 0};  
    MPI_Status status[4];  
    int right = rank == ntasks-1 ? 0 : rank+1;  
    int left = rank == 0 ? ntasks-1 : rank-1;  
    MPI_Isend ( &vec[N],   1, MPI_DOUBLE,  
        right, 0, MPI_COMM_WORLD, &requests[0]);  
    MPI_Isend ( &vec[1],   1, MPI_DOUBLE,  
        left, 0, MPI_COMM_WORLD, &requests[1]);  
    MPI_Irecv ( &vec[0],   1, MPI_DOUBLE,  
        left, 0, MPI_COMM_WORLD, &requests[3]);  
    MPI_Irecv ( &vec[N+1], 1, MPI_DOUBLE,  
        right, 0, MPI_COMM_WORLD, &requests[2]);  
    MPI_Waitall ( 4, requests, status );  
}  

void stencil_1d(int N, double vec[N+2], double dst[N+2], double *res){  
    for (int i=0; i<N; i++){  
        const int idx = i+1;  
        dst[idx] = (vec[idx-1]+vec[idx]+vec[idx+1])/3;  
        double diff = dst[idx] - vec[idx];  
        *res += diff * diff;  
    }  
}  

Table 6: Plain MPI code used on the previous example

3 Description of OmpSs

OmpSs is a data-flow programming model that extends OpenMP with pragma annotations to support tasks. These annotations are interpreted by our Mercurium source-to-source compiler, which emits calls to Nanos++, our runtime system. Nanos++ uses the information provided by user annotations to dynamically build at runtime a task dependency graph, which is used to schedule tasks in a data-flow way.
Figure 1 shows a sequential implementation of a Cholesky decomposition that has been augmented with OmpSs pragmas to specify input, output and inout parameters of functions `spotrf`, `strsm`, `sgemm` and `ssyrk`. The notation is straightforward, we use the `#pragma omp task` before a function declaration to mark it as a task, and the `input`, `output` or `inout([size]var_name)` clauses to specify the size of the input and output variables of each function. This additional information is parsed and analysed by our Mercurium compiler to convert the original source file with annotations to a new source file with the original code transformed to call to our Nanos++ runtime system. For each plain call to the annotated functions on the original file, Mercurium will generate a call to our runtime system to create a new task. The resulting file is finally compiled by a native compiler and linked with Nanos++. Figure 2 shows a diagram that depicts the usual Mercurium source-to-source compilation flow. It is worth noting that the current implementation of Mercurium supports C/C++ as well as Fortran applications.
The resulting binary will sequentially execute the original code, generating a new task each time one of the previous functions were originally called. Nanos++ will dynamically generate the task dependency graph at runtime, so it will not run a task until all of its dependencies are fulfilled. Figure 3 show the task dependency graph of the previous Cholesky example. This information will be used to optimally map tasks that can run on parallel on the available resources. The task graph dependency provides additional information that the runtime system uses to transparently perform some optimizations like data-prefetching, data renaming to avoid false dependencies or lazy write-back[2].
Extrae is a powerful instrumentation library that is seamlessly integrated with Nanos++ and MPI. Extrae generates a trace file with useful information about the application execution, which can be analysed with the Paraver[3] visualization tools to look for bottlenecks or other issues.

4 OmpSs on the Xeon Phi

To run an OmpSs application on the Xeon Phi, also known as KNC, the Nanos++ runtime system and the Extrae instrumentation library have been fully ported to this platform. On the other hand, the Mercurium source-to-source compiler only required some minor modifications to call the native Intel compiler and to link with the appropriate libraries. These three tools combined with the Intel MPI communication library provide in DEEP the programming environment for the BNs.

The initial port of the Nanos++ runtime on the KNF (a pre-release prototype hardware of the Xeon Phi) was not very complicated. Once we managed to compile Nanos++ with the Intel ICC compiler only some missing atomic operations required some additional effort to have the runtime working. As shown in the next subsection, the performance and scalability of Nanos++ is good and only one minor bottleneck has been identified so far. This bottleneck only shows up when the application ends and the runtime sequentially destroys all the threads, but this behaviour will be fixed soon. The thread/core binding policy of Nanos++ has also been modified to nicely interact with the Intel MPI implementation that runs inside the Xeon Phi. This way MPI+OmpSs applications can correctly perform on this platform, avoiding any interference between instances of Nanos++ running on different MPI processes inside the same Xeon Phi.

The port of the Extrae library to the Xeon Phi architecture has required the port of other libraries such as libxml2, which were missing on this platform. The support of MPI+OmpSs applications also required some modifications of the source code of Extrae. Finally, the use of performance counters is currently not supported because the PAPI framework does not support the KNC platform yet.

4.1 Performance evaluation

Many OmpSs applications, such as the QR, QRCA, Cholesky, Hydro (MPI+OmpSs), …, have been ported and tested on the Xeon Phi platform without any change of the source code, showing reasonable performance on most cases. However, the Xeon Phi processor requires large amounts of parallelism to make the most out of all the cores available. A well-known technique to provide this extra level of parallelism is task nesting, which dramatically increases the available number of tasks, but requires efficient management of fine-grained tasks to get the best performance. This technique has been applied to the Cholesky decomposition to study its performance and scalability.

Figure 4, compares the scalability of three different configurations of the Cholesky decomposition. The X axis shows the number of hardware threads used and the Y axis the GFlop/s obtained. The problem size is set to 8192x8192 elements. The first configuration evaluated (dark blue) is the sequential version of Cholesky compiled and linked with the parallel version of the Intel MKL library, the second one (purple) is the OmpSs version of Cholesky linked with the sequential version of the MKL library, finally the last configuration (light blue) is the OmpSs version augmented with task nesting to generate more task level parallelism. As we can see, the OmpSs version with nesting is the best performing up to 64 hardware threads. With more than 64 hardware threads the MKL obtains the best
D5.1 Prototype programming environment in Booster Node

performance. Finally, the OmpSs version without nesting only scales reasonably up to 32 cores, starting at that point the task parallelism available is insufficient to make the most of more hardware threads.

![Figure 4: Cholesky scalability on KNF (OmpSs vs MKL)](image)

With the Extrae instrumentation library and the Paraver visualization tool the different runs have been analysed to better understand what is going on. Figure 5 shows part of the timeline of the execution circled in red on Figure 4. As we can see there are still many opportunities to improve the performance of this algorithm on the KNC, because there are still many black

![Figure 5: Trace file of a Cholesky execution phase using OmpSs](image)

zones (which mean that nothing is being executed on that hardware thread). The yellow lines mean tasks creations and the blue ones task executions. Hence on that trace we can see how thread 1.1.2 (the second one) is generating most of the tasks, which are then executed by the other threads. This same thread, at the end of the trace, executes a task that is on the critical path, so the other threads are blocked until this task is finished and new tasks are generated again.

Figure 6 compares five traces of the OmpSs version with nesting running with 32, 48, 64, 80 and 96 hardware threads respectively (from top to bottom). The timeline corresponds to the
whole execution of the application. We can clearly see that the initial part of the trace scales well with the number of hardware threads. This part corresponds to the top (and wide) part of the task dependency graph depicted on Figure 3. But the execution time of last part of the trace, marked in red, proportionally increases with the number of threads and it is the main cause of the limited scalability once the hardware threads are more than 64.

The OmpSs toolchain is fully functional on the BNs and can already be used to develop OmpSs, or MPI+OmpSs applications. The Extrae instrumentation is already in place to study and optimize applications. Our next step will be to support the KNC performance counter on the Extrae library, so we can further analyse applications and identify any bottlenecks that hinders and optimal utilization of the KNC hardware.

5 Linear Algebra Libraries on Xeon Phi™

Applications in supercomputing rely on libraries for programmability, portability and performance. Such libraries range from I/O, graph partitioning, mathematical functions, and communications libraries among others. One of the most popular libraries for scientific computing is BLAS.

The BLAS libraries are intended for being used by threads/processes individually. The libraries themselves might be called in parallel using threads, or sequentially, but they are not intended for collective computation of different calling threads/processes.

Intel® provides with its SDK for MIC the MKL, a mathematical library highly optimized for their processors. The MKL contains implementations of many other libraries, including BLAS, LAPACK, FFTW, etc.

Traditionally the MKL has been the reference library for Intel’s processors. Its performance is hardly beaten, especially in those functions that are more frequently used. Therefore, in DEEP we will not replicate efforts to compete with a product with a lifespan and presence beyond DEEP. However DEEP is committed to provide the application developers the tools to efficiently use the KNC and later on the DEEP System. Therefore exhaustive tests are taking place to assess the real performance of the MKL for the KNC. The results and analysis of these tests will be provided as input to Intel’s MKL development team to further improve the library. If the performance of a function needed in DEEP does not reach the expected level and Intel does not provide an improved implementation in a reasonable time, the development of our own function will be studied.

The evaluation of MKL for KNC is underway. In the next subsection the reader can find some brief results that have been obtained up to now. The results are compared to the equivalent for a last generation CPU, a Sandy Bridge processor.

5.1 Performance tests on KNC

The benchmarking up to now has been focused on the most common kernel used: DGEMM. The platform used is a workstation equipped with 2 Sandy Bridge processors clocked at 2.6 GHz, a shared L3 cache of 20 MB and 64 GB of memory. The KNC in this workstation contains 61 cores clocked at 1.05 GHz, a private L2 cache of 512 KB per core and 8 GB of memory. The tests analyzed here focused on the evolution of the performance when the number of threads increases for a given matrix size, and oppositely, the evolution of the performance when the size of the matrix increases with a fixed number of threads.

A proper affinity is of a tremendous importance for extracting the best performance. Therefore the 3 affinity policies available for KNC were tested:
D5.1 Prototype programming environment in Booster Node

- **Scatter**: schedules the threads in different cores
- **Compact**: puts them as close as possible, without oversubscribing
- **Balanced** tries to separate the threads as much as possible, but keeping consecutive threads together when the number of threads used is bigger than the number of cores available. MKL for standard Xeon processors, such as Sandy Bridge, does not support the balanced affinity option.

For processors with a high core count and wide vector units, data alignment is also of an extreme importance. For the presented measurements square matrices with a number of elements per side that is multiple of the number of threads were selected. One exception were the tests with 183 threads (61x3). Due to the high number of hardware threads per core supported by the MIC architecture, an increase of 1 by 1 (per core) has been tested. The maximum matrix size was 15616x15616. With 3 matrices of this size the tests are close to the memory limit (5.45 GB for the 3 matrices).

DGEMM accepts transposed and not transposed matrices. To have an optimum memory access pattern the second matrix has been transposed, allowing a better cache reuse. Other setups have been tested but are not included here due to their predictable results, slightly worse than the reported ones.

Figure 7 and Figure 8 show the performance on the KNC for matrices of 7808x7808 and 15616x15616 respectively. Even though they share the same shape, the performance range is different. This shows that the MIC architecture benefits from large data structures where the prefetchers, cache, and vector units can be close to fully used. The setup with 183 threads performs worse than the setup with 122 threads. This is due to the misalignment produced by the fact that 7808 and 15616 are not multiples of 183. In the future, tests with 11712 elements will take place to overcome this issue.

The balanced policy is the one that shows the best performance in all the cases, following the line of scatter up to 61 threads, and of compact, for 183 and 244 threads. The top performance in both figures was achieved with 64 elements per row and thread.

![Figure 7: Performance of MKL's DGEMM in KNC with matrices sizes of 7808x7808 elements](image-url)
Figure 8: Performance of MKL’s DGEMM in KNC with matrices sizes of 15616x15616 elements

Figure 9 and Figure 10 show that the balanced policy is also always the best performer when increasing the matrix size and keeping the number of threads fixed. It is clear that in order to achieve the best performance in KNC coprocessors the programmers have to use more threads than cores. This is often not true for other processors. Figure 11 shows the same graph for Sandy Bridge, where no difference is found between using 16 threads or 32 threads. The reason for this effect in the KNC is its better hardware support for multiple threads per core, and the fact that the core instruction pipelines cannot be filled with less than 2 threads per core.

The peak performance for KNC was calculated using the data: 1.05 GHz, 8 elements per vector, 2 operations per clock due to the FMA, 2 threads per core to fill the pipeline and 61 cores. $(1.05 \times 8 \times 2 \times 61)/2 = 1024.8$ GigaFLOPS. The peak performance for Sandy Bridge was calculated using: 2.6 GHz, 4 elements per vector, 2 operations per clock due to a feature in Sandy Bridge that allows computing 2 separate operations if one is a multiplication and the other one an addition, and 16 cores. $2.6 \times 4 \times 2 \times 16 = 332.8$ GigaFLOPS. It is important to underline here that this is the performance for 2 Sandy Bridge processors.

Comparing the best performance in KNC and in Sandy Bridge it is easy to see that a single KNC can obtain more than twice the performance of 2 Sandy Bridge processors. However, the best results for Sandy Bridge are ~80% of the peak, whereas the best results for KNC are ~65% of the peak. Also, the curve in the KNC graphs takes off at 1952 elements per side, while for Sandy Bridge this happens at 512 elements per side. This is due to the higher speed of individual cores of Sandy Bridge that can exploit better situations where not enough parallelism is present.

The observed results show the good KNC performance, which can easily beat 2 Sandy Bridge processors when enough parallelism exists. However, it was claimed that KNC can reach 1 TeraFLOPS on DGEMM. The next steps will focus on finding under which circumstances this situation is possible, assessing the performance of other functions of the MKL library, and investigating algorithms to achieve better performance on matrices smaller than 30 MB (1952x1952 elements), since a few important applications rely on matrices of this size.
Prototype programming environment in Booster Node

Figure 9: Performance of MKL's DGEMM in KNC with 61 threads

Figure 10: Performance of MKL's DGEMM in KNC with 244 threads

Figure 11: Performance of MKL's DGEMM in Sandy Bridge with 32 threads. Results with 16 threads are very similar.
References and Applicable Documents

[1] The OmpSs programming model. Available at: http://pm.bsc.es/ompss


[4] Tutorials and workshops about BSC Paraver and Extrae tools:
   
   
List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td></td>
</tr>
<tr>
<td>API:</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td></td>
</tr>
<tr>
<td>BN:</td>
<td>Booster Node (functional entity)</td>
</tr>
<tr>
<td>Booster:</td>
<td>Hardware subsystem of DEEP comprising all Booster Nodes and intra-Booster network</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td></td>
</tr>
<tr>
<td>CN:</td>
<td>Cluster Node (functional entity)</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td></td>
</tr>
<tr>
<td>DEEP:</td>
<td>Dynamical Exascale Entry Platform</td>
</tr>
<tr>
<td>DEEP Architecture:</td>
<td>Functional architecture of DEEP (e.g. concept of an integrated Cluster Booster Architecture)</td>
</tr>
<tr>
<td>DEEP Booster:</td>
<td>Booster part of the DEEP System</td>
</tr>
<tr>
<td>DEEP Supercomputer:</td>
<td>A future Exascale supercomputer based on the DEEP Architecture</td>
</tr>
<tr>
<td>DEEP System:</td>
<td>The production machine based on the DEEP architecture developed and installed by the DEEP project</td>
</tr>
<tr>
<td>DGEMM:</td>
<td>Double precision General Matrix Matrix multiplication</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td></td>
</tr>
<tr>
<td>EC:</td>
<td>European Commission</td>
</tr>
<tr>
<td>EU:</td>
<td>European Union</td>
</tr>
<tr>
<td>GFlop/s:</td>
<td>Gigaflop, $10^{12}$ Floating point operations per second</td>
</tr>
<tr>
<td>Exascale:</td>
<td>Computer systems or Applications, which are able to run with a performance between $10^{15}$ and $10^{18}$ floating point operations per second</td>
</tr>
<tr>
<td>EXTOLL:</td>
<td>High speed interconnect technology for cluster computers developed by University of Heidelberg</td>
</tr>
<tr>
<td>Extrae:</td>
<td>BSC’s instrumentation library for OmpSs and MPI applications.</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td></td>
</tr>
<tr>
<td>FLOP:</td>
<td>Floating point Operation</td>
</tr>
<tr>
<td>FMA:</td>
<td>Fused Multiply-Add</td>
</tr>
<tr>
<td><strong>G</strong></td>
<td></td>
</tr>
<tr>
<td>GPU:</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td><strong>H</strong></td>
<td></td>
</tr>
<tr>
<td>HPC:</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td></td>
</tr>
<tr>
<td>Intel:</td>
<td>Intel GmbH Braunschweig, Germany</td>
</tr>
<tr>
<td><strong>K</strong></td>
<td></td>
</tr>
<tr>
<td>KNC:</td>
<td>Knights Corner, Code name of a processor based on the MIC architecture</td>
</tr>
<tr>
<td>KNF:</td>
<td>Knights Ferry, Intel first available processor based on the MIC</td>
</tr>
</tbody>
</table>
Prototype programming environment in Booster Node

**M**
- **Mercurium**: OmpSs’ source-to-source compiler
- **MIC**: Intel Many Integrated Core architecture, also known as Xeon Phi
- **MPI**: Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages
- **MKL**: Intel linear algebra library optimized for Intel architectures

**N**
- **Nanos++**: OmpSs’ execution runtime.

**O**
- **OmpSs**: BSC’s OpenMP SuperScalar programming model
- **OpenMP**: Open Multi-Processing, application programming interface that supports multiplatform shared memory multiprocessing
- **OS**: Operating System

**P**
- **Paraver**: BSC’s visualization tools
- **PM**: Project Manager of the DEEP project
- **PMT**: Project Management Team of the DEEP project
- **PR**: Public Relations
- **Project Coordinator**: Leading scientist coordinating and representing the DEEP project

**T**
- **ToW**: Team of Work Package leaders within the DEEP project