Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach

Vladimir Marjanovic, Josep M. Perez, Eduard Ayguade, Jesus Labarta, Mateo Valero
Barcelona Supercomputing Center
{vladimir.marjanovic, josep.m.perez, eduard.ayguade, jesus.labarta, mateo.valero}@bsc.es

Abstract – *Communication overhead is one of the dominant factors that affect performance in high-performance computing systems. To reduce the negative impact of communication, programmers overlap communication and computation by using asynchronous communication primitives. This increases code complexity, requiring more effort to write parallel code and making less readable code. This paper presents the hybrid use of MPI and SMPSs (SMP superscalar), a task-based shared-memory programming model, enhanced with a restart mechanism allowing the programmer to introduce the asynchronism that is necessary to enable the effective communication/computation overlap in a productive way. We demonstrate the hybrid use of MPI/SMPSs with the high-performance LINPACK benchmark, which uses the look-ahead technique to overlap communication and computation. MPI/SMPSs improves the performance of a pure MPI with look-ahead by 7,6% on a 1024 processors machine. In addition to better performance, hybrid MPI/SMPSs substantially reduces code complexity, it is less sensitive to network bandwidth and operating system noise, and improves the use of main memory.*

1 Introduction

The Message Passing Interface [1] (MPI) programming model has the widest practical acceptance for programming on distributed-memory architectures. In this model, processes with separate address spaces perform computation on their local data and use communication primitives to share data when necessary. Programmers tend to maximize the amount of computation out of the local memory to minimize the impact that remote communication has on the performance of the application. The two basic issues to achieve good performance and scalability are finding the appropriate work granularity for MPI tasks and finding a balanced distribution of the work.

To further improve performance and scalability, programmers have to modify their application in order to: 1) overlap communication and computation [2] and 2) accelerate the execution critical path in the computation [3]. To achieve 1), the programmer needs to use the asynchronous (non-blocking) communication calls available in MPI. The programmer can issue communication requests as soon as the data (or container for reception) is ready, perform another computation not dependent on this data, and then wait for the end of the communication. To achieve 2), the programmer has to restructure the application code to perform critical computation (and communication requests) as soon as possible delaying other not so critical computations. The use of these techniques results in increased code complexity and in reduced programmer productivity. The approach presented in this paper tries to achieve the potential performance benefits mentioned above with minimal simple program annotations in pure MPI code. The annotations are from the SMPSs (SMP Superscalar [4]) programming model, a task-based shared-memory programming model). In SMPSs the programmer annotates functions as

potential tasks and the intended use of its arguments (input, output or inout). The runtime system uses this information to dynamically build the dependence task graph and exploit the parallelism in a dataflow way.

To motivate the paper and to show the benefits of our proposed hybrid MPI/SMPSs approach, both in terms of programming productivity and execution efficiency, we use HPL [5], a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. As a highly optimized program it uses the above mentioned techniques to squeeze the utmost performance out of the parallel architecture. Even if the problem solved is simple (the solution of a system of equations) the HPL source code is relatively large (more than 19000 lines) and a good understanding of the code goes far beyond the conceptual issues of an LU decomposition [7]. As such the HPL is a good representation of MPI scientific and technical applications.

This paper makes the following contributions:

- A restart mechanism for those SMPSs tasks that block on certain events, such as blocking MPI calls. The runtime reschedules restarted tasks in order to allow fair progress of other computational tasks.
- A hybrid MPI/SMPSs approach to which achieves a global asynchronous dataflow execution of both communication and computation tasks. Overlapping computation and communication is automatically achieved by the runtime system by appropriately scheduling communication and computation tasks in a dataflow way. The proposal is demonstrated using the HPL benchmark, showing how better performance can be achieved with a much simpler program structure. In addition, better tolerance to low network bandwidth and better tolerance to external perturbations such as OS noise, are also achieved.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the HPL benchmark and the techniques used to achieve high performance and their impact in the code structure. Section 4 overviews the SMPSs programming model and the necessary changes to effectively support a hybrid MPI/SMPSs approach. Section 5 describes the implementation of HPL using MPI/SMPSs focusing on code readability. Section 6 evaluates parallel execution performance considering aspects such as the impact of the problem size and tolerance to network bandwidth and preemptions. Section 7 concludes the paper and discusses some future work.

2 Related work

Since the emergence of MPI, there has been a lot of work on improving the performance of the MPI library implementation and on reducing or hiding the negative impact of using the MPI communication primitives in parallel applications. The ability to efficiently overlap communication and computation has long been considered as a significant performance benefit for MPI applications, which has been addressed at the library specification level (non-blocking primitives) and its appropriate use in MPI and hybrid MPI/OpenMP programs [8], at the MPI library implementation level (e.g. using multi-threaded model to implement MPI point-to-point operations [9]) or proposing hardware approaches (e.g. enforcing speculative dataflow [10]).

Clusters comprised of a distributed collection of SMP nodes are becoming common for parallel computing. The hybrid use of MPI with shared-memory paradigms, such as OpenMP, has been subject of research and performance evaluation [11][12][13]. The explicit fork/join paradigm in these shared-memory programming models and the restrictive barrier

synchronization precludes more advance or aggressive overlapping of communication and computation (i.e. across iterations of an outer sequential time step loop).

In order to address the programmer productivity wall in distributed memory architectures, some languages that are based on the partitioned global address-space abstraction (PGAS), such as UPC or CAF, rely on the compiler to perform the appropriate optimizations to overlap communication and computation. The use of pure shared-memory approaches to program these architectures, relying on the compiler to translate from OpenMP to MPI [14] or on the use of a distributed-shared memory (DSM) layer also need to worry about this optimizations at the appropriate level (language extensions to express data distributions and communication [15], compiler optimization [16] or runtime library [17][18]).

Recognizing the popularity and influence in the research area of the HPL as a benchmark, a lot of previous research has focused on improving its behavior. For example, using hybrid MPI/OpenMP for SMP clusters [19], using optimized BLAS routines [20], or using an asynchronous MPI programming model [2] to explicitly code the overlap of communication and computation. In order to address the programmer productivity issue, some implementations of the HPL benchmark using PGAS languages have appeared [21][22], focusing on programming productivity and not in achieving big performance improvements.

The hybrid MPI/SMPSs approach presented in this paper exploits the use of asynchronous MPI calls without increasing complexity of code, which leads to better performance. Overlapping computation and communication is automatically done by the runtime system by appropriately schedule communication and computation tasks in a dataflow way.

3 Motivating example: High-Performance LINPACK

The HPL [5] is the most widely used benchmark to measure the floating-point execution rate of a computer and the basis to rank the fastest supercomputers in the TOP500 list [6]. The kernel solves a system of linear equations. This section describes the techniques used in the parallelization of the HPL benchmark and shows their impact in the code structure and readability, as a motivation for the proposed hybrid MPI/SMPSs approach.

We use the HPL benchmark because we consider it a good representative of a significant set of applications, in terms of communication patterns and granularities, and techniques used to achieve good performance. We analyze the HPL considering important tuning parameters that change behavior of the application and affect performance. These parameters change behavior in terms of: global computation and communication ratio, load balance, amount of fine grain (small frequent messages) communications, performance of inner sequential computation, etc. In this paper we focus more on the resulting code structure than on the actual percentage of peak computation achieved (since in many other applications it is not possible to use such highly optimized inner sequential computation cores as the BLAS routines used in the HPL).

3.1 Basic parallelization

The HPL benchmark implements a LU decomposition with partial pivoting. The elements of the coefficient matrix are double-precision floats initialized with a random distribution. The matrix to be factored has N x N elements and it is decomposed into blocks of size NB x NB, that are distributed onto a grid of P x Q processes. Given the triangular nature of the algorithm and in order to achieve load balance, the blocks are distributed among processes in a cyclic way, as shown in Figure 1.a. In a typical P by Q partition, every process will have a set of blocks

corresponding to different columns and rows regularly spaced over the original matrix. These blocks are stored contiguously in a local matrix which can then be operated on with standard BLAS routines. Of course, highly optimized versions are used in order to achieve a high percentage of processor peak performance.

A step of the main loop of the overall algorithm is composed of the *panel factorization* and the *update of the trailing submatrix*, as shown in Figure 1.b. We will use the term *panel* to refer to the blocks in a column of the matrix and *trailing submatrix* to refer to the blocks on the right of the *panel*. The LU factorization is done by iteratively applying these two steps on the trailing submatrix. The number of iterations inside the main loop is directly related to block dimension NB and matrix dimension N.

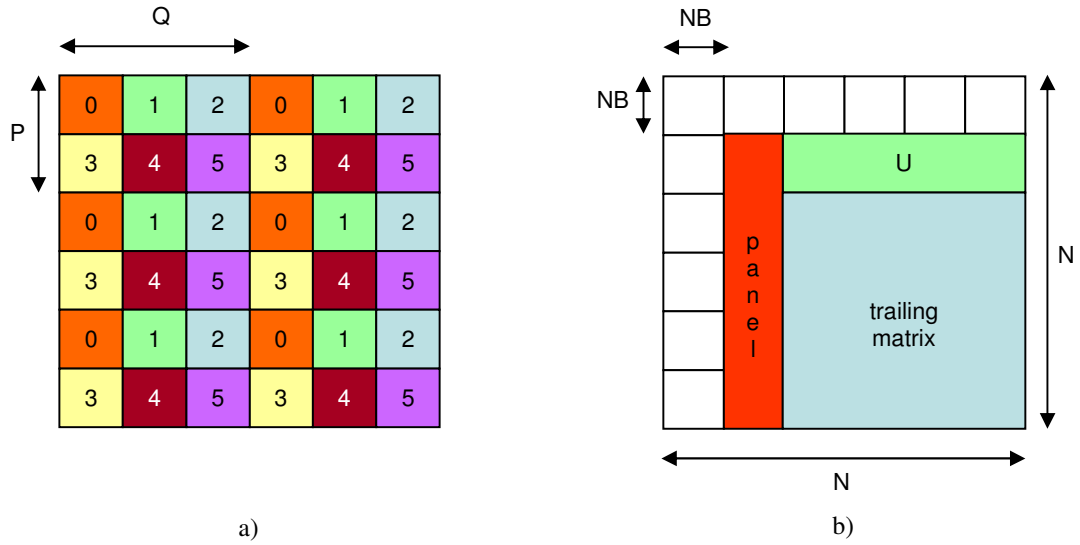


Figure 1. a) P by Q partitioning of the matrix in 6 processes (2x3 decomposition) and b) one step in the LU factorization (panel, U and trailing matrix)

When the computation of the panel factorization is finished, the panel needs to be broadcasted to the other processes along the Q dimension so that they can perform the update of the trailing submatrix. This broadcast can be implemented using the `MPI_Bcast` call if the machine provides an efficient implementation of this primitive (as for instance in Blue Gene [23]). Alternatively, several methods are provided in the HPL distribution to perform the broadcast by circulating the data in one or several rings of point-to-point communications. The pseudo-code for a simplified version of the main loop in the HPL is shown in Figure 2.a.

3.2 Look-ahead

Look-ahead technique restructures the code in order to accelerate the execution of critical path in the computation and to overlap communication and computation. The panel factorization process lies in the critical path of the application. When the panel in iteration j has been factored by processes in column q=j%Q and broadcasted, the globally next urgent job to perform is the factorization and communication of the panel in iteration j+1 by processes in column (q+1)%Q. The HPL code includes a look-ahead option that performs this optimization. As soon as a column of processes q receives a panel factored by its previous column, they update, factor and send the next panel before updating the rest of panels also owned by this column of processes. In this

way, the transmission of the data can be advanced and the global critical path is accelerated. Introducing this optimization requires significant changes in the source code, not only in the main iterative loop, but also in the different routines called inside this loop. In addition to that, the programmer has to explicitly allocate several panels and to specify the part of the code that is executed while the panel has still not arrived. Probing to retransmit messages is also added in every function which increases internal code complexity. Pseudo-code just showing the changes required in the main loop for a version with look-ahead degree of 1 is shown in Figure 2.b. Higher degrees of look-ahead requires further modifications in the code and data structures.

3.3 Partitioning

The rationale for a two-dimensional data distribution originates from the actual amount of data to be transmitted at every step and the potential concurrency of such transmissions. A value of P larger than 1 implies that different blocks of the panel can be sent concurrently as each of the P processes has one part of the panel. However, this also introduces additional communication in the factorization step. These communications are of much finer grain than those in the panel broadcast phase. The value of P introduces a clear trade-off between communication and synchronization overhead in this phase and the parallelism to execute this phase which lies in the critical path. The case of P=1 is a special situation: it avoids all communications in the factorization phase as well as the need to broadcast the *U submatrix* (see Figure 1.b) in the update phase, but has to pay for a long sequential time of the factorization phase and long communication chain for the panel broadcast.

```
#define NPANELS N/NB*Q)
#define root (j%Q==my_rank)

double A[N/P][NPANELS*NB];
double tmp_panel[N/P][NB];

int k=0;

for( j = 0; j < N; j += nb ){
   if (root){
       factorization (&A[k*NB][k*NB], tmp_panel, k);
       k++;
   }
   broadcast (root, tmp_panel);
   for(i = k; i < NPANELS; i++ )
       update (tmp_panel, &A[k*NB][i*NB],k);
}
```

a)

```
double tmp_panel[2][N/P][NB];
double *p[2];

p[0] = tmp_panel[0][0][0];
p[1] = tmp_panel[1][0][0];
k = 0; j = 0;

if (root){
       factorization(&A[k*NB][k*NB], p[0], k);
       k++;
}
broadcast_start(root, p[0]);
for (j = nb; j < N; j += nb){
   broadcast_wait(p[0]);
   if (root){
         update (p[0], &A[k*NB][k*NB], k);
         factorization (&A[k*NB][k*NB,], &p[1], k);
         k++;
   }
   broadcast_start(root, p[1]);
   for (i = k; i < NPANELS; i++)
          update_and_broadcast_progress (p[0], &A[k*NB][i*NB], k, root, p[1]);
   p[0] = p[1];
}
broadcast_wait(p[0]);
for (i = k; i < NPANELS; i++)
    update (p[0], &A[k*NB][i*NB], k);
```

b)

Figure 2. a) Simplified version of the main loop in HPL and b) version with look-ahead equals to one.

4 Hybrid MPI/SMPSs programming model

In this section we overview the SMPSs programming model and describe the proposal to use it in a hybrid MPI/SMPSs approach. We will emphasize the potential to overlap computation and communications and describe the extensions necessary in SMPSs to support an efficient use of the processors.

4.1 SMPSs overview

The SMP superscalar programming model [4] extends the standard C/Fortran programming language with a set of pragmas/directives to declare functions that are potential tasks and the intended use of the arguments of these functions:

```
#pragma css task [clause-list]
{function-header|function-definition}
```

With the following possible clauses:

- `input(data-reference-list)`
- `output(data-reference-list)`
- `inout(data-reference-list)`
- `highpriority`

The first three clauses are used to indicate argument use and the last one to specify high priority when scheduling the task.

Based on the input/output specifications and the actual arguments in function invocations, the runtime system is able to determine the actual dependences between tasks and schedule their parallel execution so that these dependences are satisfied. The dependences derived at runtime replace the use of barriers in most of the cases, allowing the exploitation of higher degrees of distant parallelism.

In addition to the dependences derived from the argument direction, the SMPSs programming model adds a barrier (to wait for the termination of all tasks generated up to this point) and a data dependent task wait construct:

```
#pragma css barrier          #pragma css wait on (data-reference-list)
```

The SMPSs environment consists of a source-to-source compiler that substitutes the original invocations of the annotated functions with calls to an *add_task* runtime call, specifying the function to be executed and its arguments. The resulting source code is compiled using the platform native compiler and linked to the SMPSs runtime library. The *add_task* runtime call uses the memory address, size and direction of each parameter at each function invocation to build a dependence task graph. A node in the task graph is added to represent the newly created task and it is linked to previous tasks on whose output it depends. Once a task is finished, the runtime updates the task graph, inserting in the ready queue all those tasks that have no pending dependences. Concurrently with this main thread, a set of worker threads, started at initialization time, traverse this list looking for tasks ready for execution. In the case that the main thread encounters a synchronization (barrier, wait on specific data or end of the program) it cooperates with the worker threads to execute pending tasks.

The actual schedule of the tasks is selected by the runtime based on its view of the task graph which may be only a partial graph of the whole application. The *highpriority* clause gives a hint to the runtime system about the “urgency” of scheduling the task. The runtime has two ready list

queues and tasks from the high priority queue are selected before tasks in the low priority queue. This mechanism allows a programmer with global understanding of the critical computations to influence the actual schedule.

In order to reduce dependencies, the SMPSs runtime is capable of renaming the data, leaving only true dependencies. This is the same technique used in superscalar processors and optimizing compilers to remove false dependencies due to the reuse of data storage (e.g. registers). In SMPSs the renaming may apply to whole regions of memory passed as arguments to a task. Such renaming is implemented by the runtime, allocating new data regions and passing the appropriate pointers to the tasks, which themselves do not care about the actual storage positions passed as arguments. The runtime is responsible for properly handling the actual object instance passed to successive tasks. Also if necessary, it copies back the data to its original position. This mechanism has the potential to use otherwise available memory to increase the actual amount of parallelism in the node. An uncontrolled usage of these mechanisms may nevertheless result in swapping and thus extreme performance penalty. A parameter in a configuration file limits the size of memory that can be used for renaming.

4.2 Taskifying MPI calls: a first step towards hybrid MPI/SMPSs

An MPI process usually contains sequential computations between MPI calls. Further fine-grain shared memory parallelism can be exploited, if the architecture supports it, using for instance OpenMP for parallel regions between MPI calls. However OpenMP is based on a fork/join execution model with barrier synchronizations. These barriers inside each iteration preclude the exploitation of parallelism across iterations, a feature that is necessary to exploit the lock-ahead parallelism in HPL and achieve the effective communication/computation overlap. The dataflow synchronization in SMPSs will allow to exploit the distant parallelism across multiple iterations, just based on the availability of data at runtime.

In order to allow a pure dataflow execution model, the first step consists on considering MPI calls as SMPSs tasks that consume data (`MPI_Send`) or produce data (`MPI_Recv`) in the task graph. We can encapsulate these communication requests as SMPSs tasks by specifying their inputs (for sends) and outputs (for receives). By doing so, we may rely on the general SMPSs scheduling mechanism to reorder the execution of such tasks relative to the computational tasks just guaranteeing that the dependences are fulfilled. Assuming a sufficient number of processors for each MPI process this would have the effect of propagating the asynchronous dataflow execution supported by SMPSs within each node to the whole MPI program.

4.3 Handling blocking MPI calls: extending SMPSs with restartable tasks

As opposed to standard computation tasks, communications tasks have an undetermined execution time, depending on when (or whether) the communication partner invokes the matching call. In addition, blocking communication calls could lead to deadlock situations [24] in an architecture where the number of threads per node is limited or in our initial target where this number is just one.

To appropriately handle blocking communication calls, the programmer needs to split a blocking call into a non-blocking call to issue the communication request and a wait call to wait for the data. This separation just moved the deadlock risk mentioned above from the blocking communication call to the wait call. To solve the problem we added a new pragma in the SMPSs programming model:

```
#pragma css restart
```

The effect of this pragma is to abort the execution of the current task and put it again in the ready queue. With this new pragma, the wait can be implemented with 1) a `MPI_Test` to check whether data has already arrived or not; 2) if so, the `MPI_Wait` can be done and data is available for SMPSs task depending on it; 3) if not, the `restart` pragma is executed, aborting the wait task and queuing it again in the ready queue for later consideration. The code fragments in Figure 3 show the code transformation done for a blocking receive call and for a broadcast operation.

```
#pragma css task output(buf, req)
void recv (<type>
     buf[count], MPI_Request *req){
  MPI_Irecv(buf,…,req);
}

#pragma css task input(req)
void wait (MPI_Request *req){
  int go;
  MPI_Test (req, &go, ...);
  if (go==0) #pragma css restart;
  MPI_Wait (req_recv, …);
}

void application_receive(){
   recv ();
   wait ();
}
```

a)

```
#pragma css task input(buf) output(req)
void send (<type> buf[N*nb],
          MPI_Request *req);

#pragma css task input(req)
void wait (MPI_Request *req);

#pragma css task output(buf, req)
void recv (<type> buf[size],
          MPI_Request *req);

void broadcast (int root, <type> buf){
  if (root){
       send (buf, req_send);
       wait (req_send);
  } else {
       recv (buf, req_recv);
       wait (req_recv);
       if (necessary) {
              send (buf, req_forward);
              wait (req_forward);
       }
}
```

b)

Figure 3. Taskifying process for a blocking receive (a) and a broadcast (b) with SMPSs.

This approach requires the explicit separation of blocking MPI calls into the appropriate sequence of their corresponding non-blocking calls. Both tasks are invoked in sequence in the source code although if data take some time to arrive, the scheduler will launch the execution of other computational tasks. With the proposed approach, the programmer does not need to think about the placement of both asynchronous calls, which would force a specific schedule which may or may not be the most appropriate. Notice that the transformation described above could be even hidden inside the implementation of the MPI library or in stubs calling it, making the use of the hybrid MPI/SMPSs even more simpler and productive.

The possibility to abort and resubmit a task has several implications. First, the task should not have any side effect on the state of the program or environment, as the whole task could be repeated a number of times that is outside the control of the programmer. Second, the runtime should not immediately selected the aborted task for execution if there are other tasks in the ready queue, as this may result in the same resource starvation and associated deadlock we tried to avoid. And third, the runtime should give these aborted tasks an opportunity to execute

relatively frequently as this will result in better application responsiveness to incoming messages and may result in faster propagation of data along the critical path.

In our current implementation a task that invokes a restart primitive is inserted back in the ready queue after the first ready task, leaving at least a normal ready task between two restarted tasks in the list. This is done to avoid a potential deadlock in the case of two concurrent wait tasks. If the task that is restarted is marked as *highpriority*, it looses this condition and goes into the low priority list. Because the basic mechanism described above re-injects restarted tasks towards the head or the low priority ready queue, the net effect is that the restarted task still goes before the many possibly ready tasks in the low priority queue.

5 Hybrid MPI/SMPSs LINPACK

In this section, we will describe how the LINPACK code can be restructured to use the proposed hybrid MPI/SMPSs model. First we describe the transformation assuming P=1 (one-dimensional data decomposition) and later comment the differences for a two-dimensional decomposition (P>1).

5.1 One-dimensional decomposition

The structure for the one-dimensional decomposition LINPACK version with SMPSs is sketched in Figure 4. Notice that it is the same code as in Figure 2.a just with the specification of the computation and communication tasks. The computation part of algorithm is composed of the panel factorization and the update of trailing submatrix. The factorization is performed by a single task whose input is the updated panel of a previous iteration and whose output is the factorized panel for the current iteration. The update of the trailing submatrix is partitioned in a set of tasks, each of them taking as input the factored panel (either produced locally or received) and a subset of the local panels to update. The code shown uses the broadcast operation already described in Figure 3.b., in which the original sends and receives are replaced by tasks with the appropriate input and output arguments.

Figure 5 shows a partial task graph generated during the execution of this hybrid version. In the original HPL with no look-ahead one process executes all tasks in one iteration j before proceeding the the execution of the next iteration j+NB, precluding the overlapping of communication and computation. The original HPL with look-ahead tries to follow the critical path executing tasks that are a certain number of iterations in advance (degree of look-ahead). The control flow in the HPL code achieves this execution. The hybrid MPI/SMPSs naturally follows the critical path of the execution by executing the task graph in a dataflow way. So for example, process p in Figure 5 would execute recv(j), send(j), first instance of update(j), fact(j+NB), send(j+NB), … With no look-ahead or dataflow execution, fact(j+NB) would not start until all instances of update(j) were finished, delaying the critical path of the application. This global critical path proceeds along the panel factorization, communication to the next process, update of the first uncompleted panel in this process, factorization of this panel and so on. In order to speedup the computation along this path, the send and receive tasks are labeled as *highpriority*. Notice that the renaming mechanism in SMPSs is dynamically doing the replication of panels that is necessary to execute the tasks in a dataflow way and whose management added part of the complexity to the code in Figure 2.b.

```
#pragma css task input(A, k) output(panel)highpriority
void factorization (double A[N/P][NB], double tmp_panel[N/P][NB], int k);

#pragma css task input(panel, k) inout(A)
void update (double tmp_panel[N/P][NB], double A[N/P][NB],int k);

#define NPANELS N/(NB*Q)
#define mine (j%Q==my_rank)

double A[N/P][NPANELS*NB];
double tmp_panel[N/P][NB];

int k=0;

for (j = 0; j < N; j += nb){
   if (root){
       factorization (&A[k*NB][k*NB], tmp_panel, k);
       k++;
   }
   broadcast(root, tmp_panel);
   for (i = k; i < NPANELS; i++)
       update(tmp_panel, &A[k*NB][i*NB], k);
}
```

Figure 4. One-dimensional decomposition version for HPL using hybrid MPI/SMPSs.

5.2 Two-dimensional decomposition

In order to achieve good load balance and scalability of the algorithm, the HPL distributes data onto two dimensions. As we commented in Section 3.3, this data distribution adds new communications in the algorithm and increases the code complexity. New communication operations appear in the factorization and update phases. In the update phase, what is called pivoting broadcasts the U submatrix across the P processes and pivots local rows.

We explored two possibilities to parallelize with our hybrid approach. The first one consists on taskifying all communication operations in panel factorization and pivoting. However, this represents less than 5% of the execution time of the main loop but accounts for more than 99% of the total number of messages. In addition these messages are very small (eager protocol). As a consequence, the overhead introduced to dynamically create and manage these tasks is too large to compensate any benefit. The second alternative explored is much simpler and consists on defining the pivoting function as a new task, with the appropriate clauses to specify the direction of the arguments. The panel broadcast, the most expensive communication part of algorithm, overlaps computation part as well as in the one-dimensional decomposition and we also keep the code readability as well.

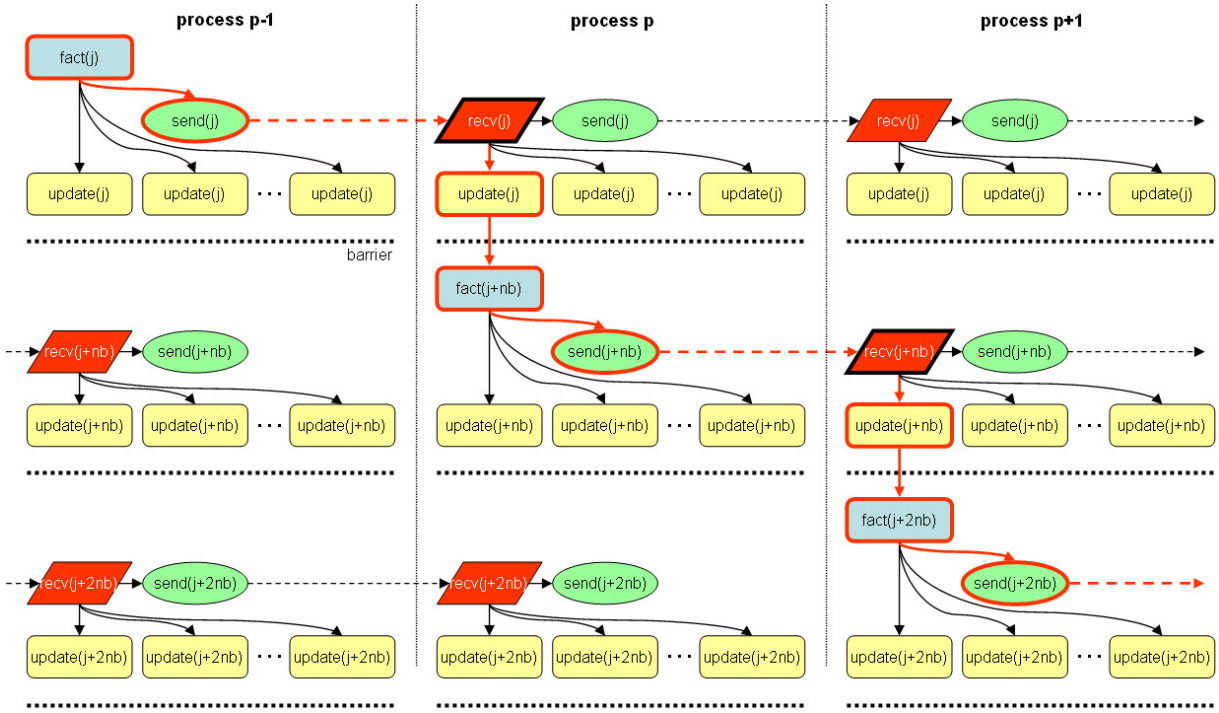


Figure 5. Partial dataflow graph for the execution of HPL: MPI process execution in vertical and iteration j of main loop in horizontal. Nodes correspond to the different tasks: fact (panel factorization), send (panel send), recv (panel receive) and update (panel update). In red the critical path of the partial execution.

6 Performance results

In this section we present results for the experimental evaluation of the proposed hybrid MPI/SMPSs programming model applied to HPL. The evaluation is done using 128, 512 and 1024 processors of a cluster made of IBM JS21 blades and Myrinet interconnection network. First, we analyze the mentioned important tuning parameters from section 3.The one-dimensional decomposition is used for the executions with 128 processors and the two-dimensional decomposition for the executions with 128 (P=8, Q=16) 512 (P=16, Q=32) and 1024 (P=16, Q=64) processors. We compare the raw performance of the original HPL version and our hybrid MPI/SMPSs version, for different problem sizes and number of processors. Second we evaluate other potential gains: 1) tolerance to low network bandwidth; and 2) robustness in the presence of OS noise and preemptions. In order to introduce the perturbations in 1) and 2) we modified the HPL code in such way that modifications do not have influence on the correctness of application results (the modifications only try to simulate the issues without side-effects).

6.1 Performance references

Performance of a system depends on a large variety of factors. Achieving the best performance requires well done analysis of these factors. Linpack offers the list of 31 tuning parameters that defines how the problem is to be solved. Varying these parameters LINPACK stresses some parts of the system more than others and also gives a good representation of some MPI scientific

and technical applications. We did analysis on 128 processors assigning the most important tuning parameters: problem size (N), block size (nb), data decomposition (P and Q), overlapping communication and computation by using look-ahead technique.

The largest problem size (N) that fits in memory gives the best performance of the system. In effect, matrix dimension (N) defines a ration between communication and computation. For small problem size, Linpack is very sensitive to network performance, increasing the problem size communication and computation increases as well, but computation increases much faster and the communication overhead decreases. For very large matrix, the influence of network performance significantly drops. Figure 6 shows performance results for various problem sizes using look-ahead technique and LINPACK version without using look-ahead technique. The LINPACK version with look-ahead turned on decreases the communication overhead by overlapping communication and computations and gives better performance results for small problem sizes, while both versions give almost the same performance for large problem size due to Amdahl's law.

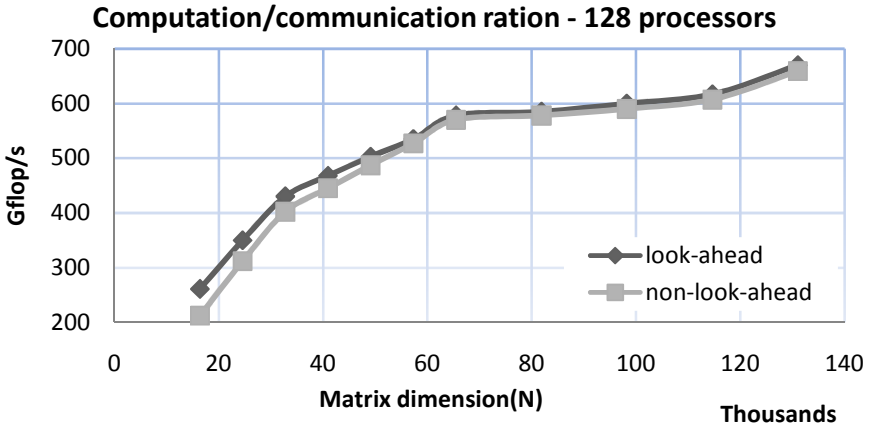


Figure 6. Computation/communication ration. Large problem size reduces the communication overhead by increasing ration between computation and communication((P,Q)=(8,16), nb=128).

Proper block size (nb) responds to data distribution, computation granularity (probing granularity for look-ahead techniques) and performance of BLAS routines. Large block sizes tend to a load imbalance and limits probing for message, while small block sizes increases internal blocking factor of BLAS routines and as such decreases efficiency of matrix multiplication. Figure 7 presents sensitivity to various nb. For this experiment we used N=65536 and P=8 and Q=16, as such nb=128 gives optimal interaction between data distribution and computation granularity.

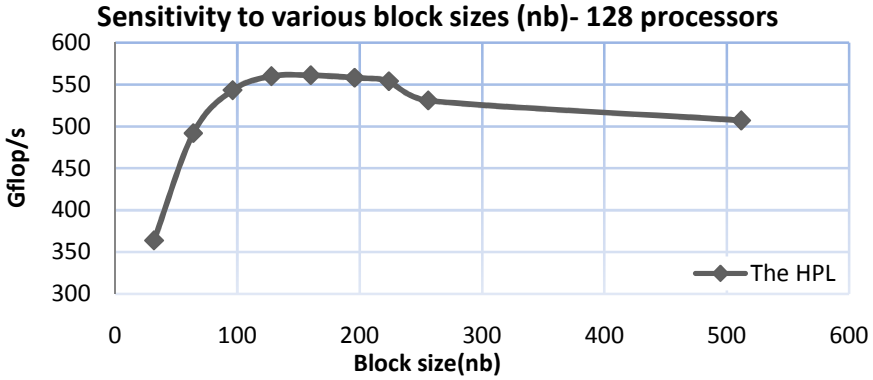


Figure 7. Sensitivity to various block size.

Variables P and Q determine the data distribution. For 128 processors, possible grids are (P,Q)={(1,128),(2,64),(4,32),(8,16),(16,8),(32,4),(128,1)} , these respond for load balance and scalability of the algorithm. In order to analyze load balance, we measure the total execution time for gemm routines, BLAS routine for matrix multiplication in the update phase, as the most expensive computations in the application, Figure 8.a. shows that a good load balance prefers square grids. A factor of load imbalance is ration between the longest and the shortest execution time of computations obtaining from MPI processes.
Processes do the panel broadcast operation over Q-processes, so large value Q may limits scalability of the algorithm. Look-ahead techniques and hybrid MPI./SMPSs attack this issue trying to hide a cost of the broadcast operation, We have already seen how increasing the size of problem reduces communication overhead. Figure 8.b. establishes these statements because hybrid MPI/SMPSs version shows higher performance improvement than pure MPI for small N and P values. Larger P value increases number of fine grain communications and communication latency causes performance degradation. Both versions suffer due to latency impact, especially for small problem sizes.

In order to test our approach we found interesting two cases: first (1,128) decomposition where coarse grain communications do not appear and the communication overhead only comes from the broadcast operation; second (8,16) decomposition that gives the best performance and contains a good ration between large messages for coarse grain communication(broadcast operation) and small messages for fine grain communication (panel factorization). These cases represent behavior of some MPI scientific and technical applications offering challenge to our approach.

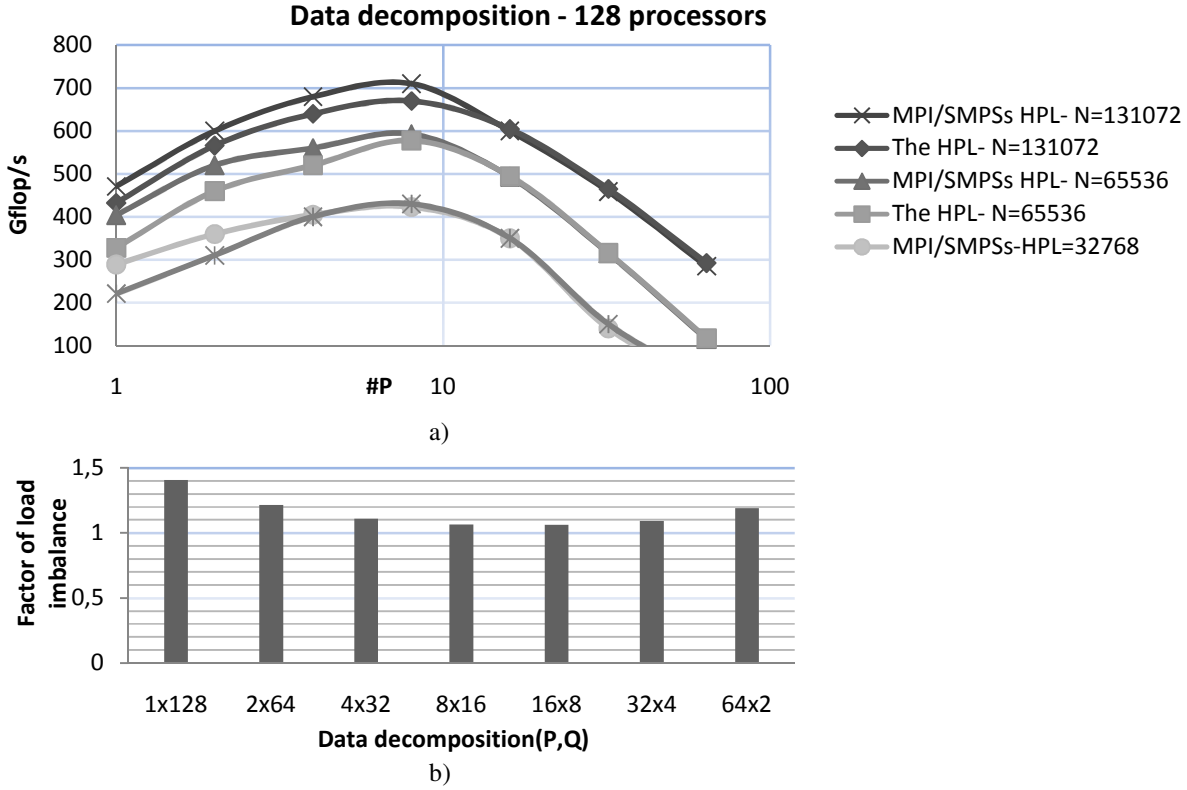


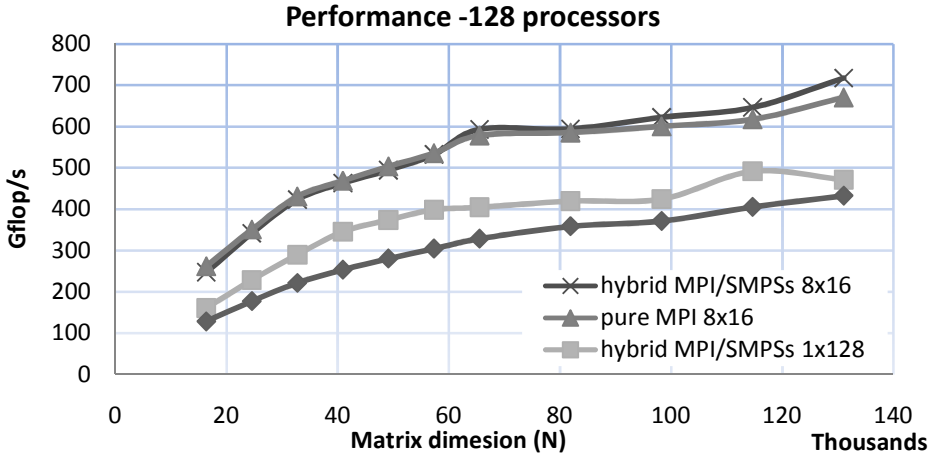
Figure 8. a) Sensitivity to various data decompositions and b) Load imbalance issue due to various data decompositions(N=65536, nb=128).

6.2 Basic comparison

The MPI/SMPSs is low-level programming model but introduces higher level of abstraction than pure MPI, which may cause performance degradation. Figure 9 shows the performance rate (Gflops) of LINPACK for MPI with look-ahead and for hybrid MPI/SMPSs. Notice that in general the hybrid MPI/SMPSs version shows better performance results because of the non-blocking MPI calls that were used and the efficient use of memory to perform the look-aheaded execution technique. In general, increasing the matrix size increases the performance rate because the influence of communication overhead is smaller. We identify three patterns as application case studies for 128 processes:

- For small matrices, the computation part of the application is small and there is not much possibilities to overlap communication and computation, which makes the network parameters (bandwidth and latency) the dominant factors. In this case, the hybrid MPI/SMPSs gets 20% better performance than the original LINPACK version.
- By increasing the problem size the hybrid MPI/SMPSs version exhibits a full strength against the original MPI version with look-ahead. The hybrid version increases performance by 41%.
- For very big problem sizes, the communication overhead is not predominant and as a consequence the hybrid MPI/SMPSs version just improves the performance by 9% for the same input data.

These patterns are not so clearly separated for 2D decomposition with 128, 512 and 1024 processes. In this case, the use of the two-dimensional data decomposition with blocking MPI calls makes the behavior slightly different. For small and large problem sizes the hybrid MPI/SMPSs version shows up to 5% performance improvement, while our programming model demonstrates high potential for medium problem sizes and reaches 15% performance improvement. The hybrid MPI/SMPSs creates the negligible overhead and efficiently uses all asynchronous MPI features.



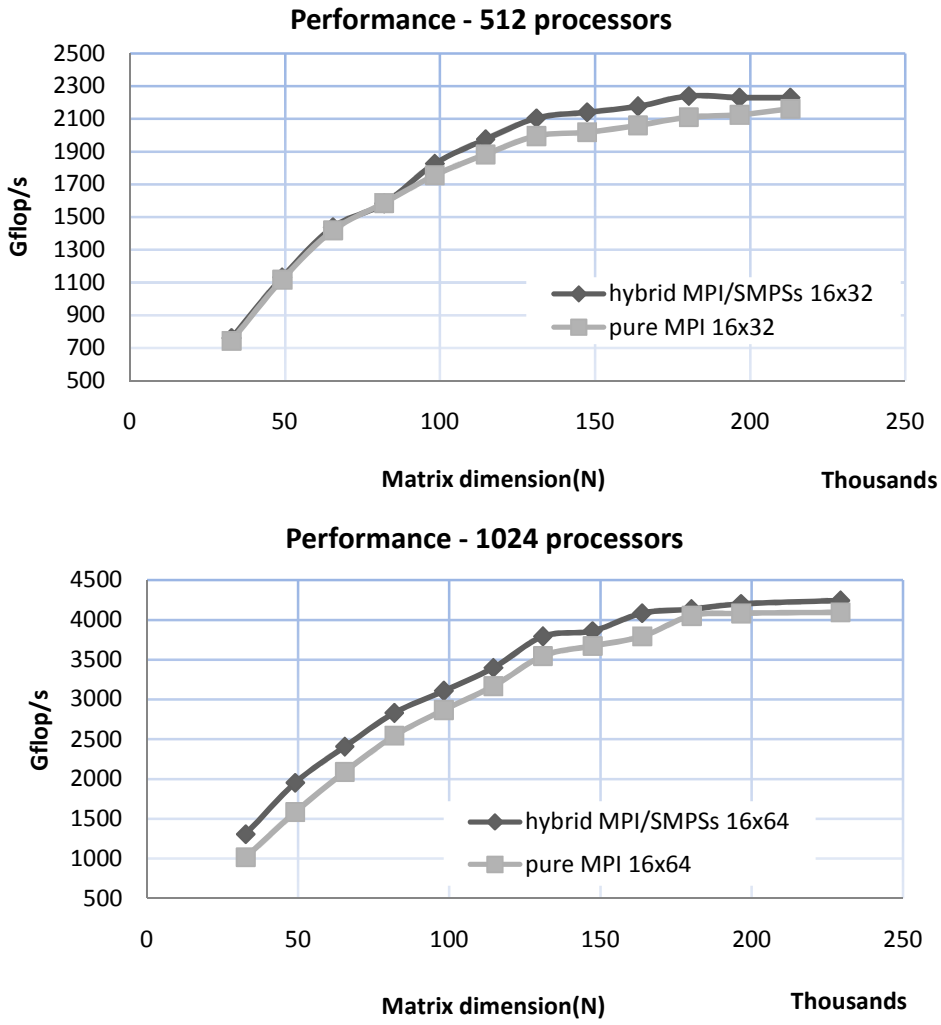


Figure 9. Performance rate of the LINPACK benchmark for two different versions (original HPL with look-ahead one and hybrid MPI/SMPSs). Results are presented for 128, 512 and 1024 processors.

6.3. Tolerance to low bandwidth

Bandwidth is one of the important metrics in the interconnection network technology [26]. In future multi-core systems with a large number of cores per node, the impact of bandwidth will become more important. If computing nodes become much faster relative to the interconnection network, performance will be more sensitive to the low bandwidth. Even in today systems, it is always important to know how sensitive my application is to the network bandwidth, or in other words, how much bandwidth could I save without penalizing much the performance of my application?. The ability of the programming model to overlap communication and computation may change the physical bandwidth requirements of the application.

In order to explore the impact of lower bandwidth we used a dilation technique by modifying the source code such that for each message of size S an additional message of size f*S is

transferred between two dummy buffers at sender and receiver. A value of f=1 would mimic the availability of half the original bandwidth. Figure 10 shows the execution time of a LINPACK run for a problem size that reaches the asymptotic behavior on 128 and 512 processors.

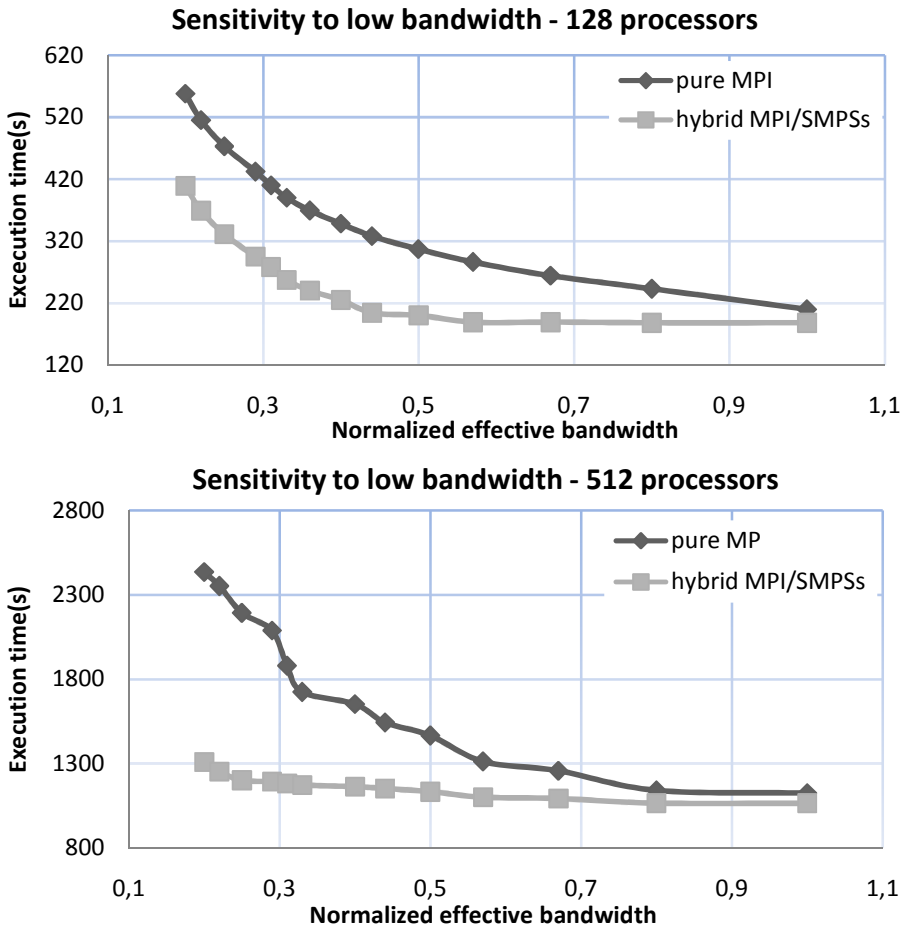


Figure 10. Sensitivity to low network bandwidth. Results are presented for 128 and 512 processors.

The plot shows that even if starting at a significantly smaller execution time, the hybrid MPI/SMPSs version is not affected by a reduction of bandwidth close to 40%. The HPL version is much more sensitive to such reduction, resulting in an increase of the 120% in the execution time. Results for 512 processors show that the hybrid MPI/SMPSs version is almost not affected for five times smaller bandwidth. In the case of HPL, the execution time doubles for the same reduction of bandwidth.

6.4. Tolerance to OS noise

Operating system noise in general and process preemptions in particular have been identified as one of the important potential causes of significant performance degradation. Local perturbations easily propagate and accumulate through the whole program dependence chains and specially at

global synchronization points. The high levels of asynchronism introduced by the hybrid MPI/SMPSs model make the applications more tolerant to such perturbations.

In order to evaluate this effect we have modified the source code of the application by generating an additional thread per process that iterates on a loop that alternates sleeping and computing phases. By controlling the average duration of both phases it is possible to simulate different levels of OS noise. Figure 11 shows the sensitivity of the two versions of the code to process preemptions. Preempting is modeled with periodical computations of 500ms, and its amount can be regulated by changing the period of the sleeping phase. The plot presents the execution time of the application as function of total amount of the noise injected. As can be seen in the figure, the hybrid MPI/SMPSs version tolerates preemption much better. For 128 processors and the period of preemption bursts of 3 seconds, performance of our version does not suffer, while execution time of the HPL is increased for 11%. At very high preemption frequencies, both versions suffer the impact of the perturbation.

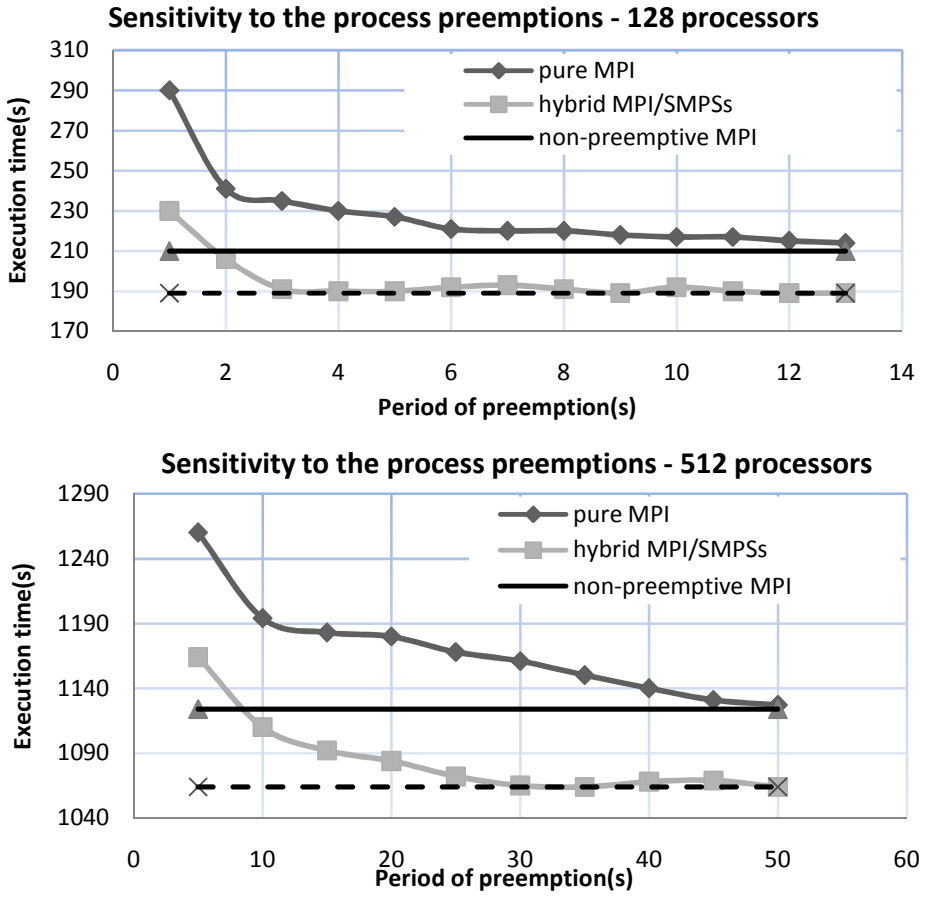


Figure 11. Sensitivity to the process preemptions. Results are presented for 128 and 512 processors.

7. Conclusions and future work

This paper presents the hybrid use of MPI with a task-based shared-memory programming model, SMPSs. The hybrid MPI/SMPSs increases code readability and introduces higher-level

abstraction than pure MPI without performance sacrifice. Simple annotations in the original MPI code allow the programmer to provide hints to the runtime system to achieve a good computation/communication overlap and to fast forward the execution of the critical path of the application. The experimental evaluation on a real supercomputer reveal performance improvements up to 41% when compared to the original version of the LINPACK benchmark for the same input data. . Also, the resulting program is less sensitive to network bandwidth and to operating system noise, such as process preemptions.

In the HPL collective operations, such as the broadcast, are implemented using point-to-point communication calls. This paper demonstrates how the hybrid MPI/SMPSs programming model works well for MPI applications that do not use blocking MPI collective operations. In the future work, we shall explore applications that contain MPI collective operations(`MPI_Alltoall`, `MPI_Scatter`, `MPI_Gather`, etc) by combining the hybrid MPI/SMPSs programming model and non-blocking MPI collective operations library [24].

There are many possibilities for improving the development of the MPI/SMPSs programming model, specially those aspects related with the management of communication tasks and their restartable behavior. The better performance results as well as programmer productivity give a promising future to the proposed programming model. Regarding portability, the same annotations used in SMPSs are also used in CellSs (Cell Superscalar [27]). This means that by just recompiling the hybrid MPI/CellSs could be executed on a cluster based on the Cell B./E. multicore architecture.

References

[1] The Massage Passing Interface standard. Available at: http://www-unix.mcs.anl.gov/mpi/.

[2] Ta Quoc Viet and Tsutomu Yoshinaga. Improving Linpack Performance on SMP Clusters with Asynchronous MPI Programming, IPSJ Digital Courier, Vol. 2, pp.598-606. 2006.

[3] Martin Schulz. Extracting Critical Path Graphs from MPI Applications. Proceedings of the 2005 IEEE International Conference on Cluster Computing September 2005.

[4] Josep M. Perez, Luis Martinell, Rosa M. Badia and Jesus Labarta. A dependency aware task-based programming environment for multi-core architecture. Proceedings of IEEE Cluster 2008, September-October 2008.

[5] Jack Dongarra, Piotr Luszczek and Antoine Petitet. The LINPACK Benchmark: Past, Present and Future. Concurrency and Computation: Practise and Experience. Vol. 15, issue 9, pp. 803–820. 2003.

[6] TOP500 Supercomputer sites. Available at: http://www.top500.org/lists.

[7] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov,Antoine P. Petitet, David W. Walker and R. Clint Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. Technical Report: UT-CS-94-246 , September 1994.

[8] Timothy H. Kaiser, Scott B. Baden. Overlapping communication and computation with OpenMP and MPI. Scientific Programming , Vol. 9, no. 2-3, pp. 73-81, August 2001.

[9] Mao Jiayin, Song Bo, Wu Yongwei, and Yang Guangwen. Overlapping Communication and Computation in MPI by Multithreading. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Application (PDPTA'06), March 2006.

[10] Vladimir Subotic, Jesus Labarta and Mateo Valero. Overlapping MPI Computation and Communication by Enforcing Speculative Dataflow. Proceedings of the 2008 International Conference on High Performance Embedded Architectures & Compilers (HiPEAC-2008), January 2008.

[11] Rolf Rabenseifner, Gerhard Wellein. Comparing of Parallel Programming Models on Clusters of SMP Nodes. Proceedings of Cray User Group Conference, March 2003.

[12] Rolf Rabenseifner. Hybrid Parallel Programming : Performance Problem and Chances. Proceeding of the 45th CUG Conference, May 2003.

[13] Gabriele Jost and Haoqiang Jim. Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster. NAS Technical Report NAS-03-019, November 2003.

[14] Ayon Basumallik and Rudolf Eigenmann, Towards automatic translation of OpenMP to MPI. Proceedings of the 19th Annual International Conference on Supercomputing, June 2005.

[15] Juan Jose Costa, Toni Cortes, Xavier Martorell, Eduard Ayguade and J. Labarta, Running openMP applications efficiently on an everything-shared SDSM. 18th International Parallel and Distributed Processing Symposium (IPDPS-2004). April 2004.
[16] Seung-Jai Min, Ayon Basumallik and Rudolf Eigenmann, Optimizing OpenMP programs on software distributed shared memory systems. International Journal of Parallel Programming, Vol. 31, no. 3, pp. 225-249. June 2003.
[17] Liang Peng, Mingdong Feng and Chung-Kwong Yuen. Evaluation of the performance of multithreaded Cilk runtime system on SMP clusters. Proceedings of 1999 IEEE International Workshop on Cluster Computing, December 1999.
[18] Y. Ojima, M. Sato, H. Harada and Y. Ishikawa. Performance of cluster-enabled OpenMP for the SCASH software distributed shared memory system. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003). May 2003.
[19] Ta Quoc Viet, Tsutomu Yoshinaga and Masahiro Sowa. Optimization for Hybrid MPI-OpenMP Programs with Thread-to-thread Communication. Proceedings of 2004 Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP 2004), 2004.
[20] Intel® Optimized MP LINPACK Benchmark for Clusters 1.0. Intel Corporation 2005.
[21] Berkeley UPC Applications. Poster session at IEEE SuperComputing'05. November 2005.
[22] J.K.Reid, J.M. Rasmussen and P.C. Hansen. The LINPACK Benchmark in Co-Array Fortran. Proceedings of the Sixth European SGI/Cray MPP Workshop, Septemeber 2000.
[23] G. Almasi, C. Archer, J.G. Castanos, J. A. Gunnels, C. C. Erway, P. Heidelberger, X.Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, andB.Toonen. Design and implementation of message-passing services for the Blue Gene/L supercomputer. IBM Journal of Reasearch and Development, vol. 49, no. 2-3 pp. 393-406. March 2005.
[24] Torsten Hoeer1 and Andrew Lumsdaine. Non-Blocking Collective Operations for MPI-3. January 2008.
[25] Jack Dongarra, LINPACK benchmark and some issue. Talk at IEEE/ACM International Conference of High Performance Computing, Networking, Storage and Analysis, (SC08). November 2008
[26] Alex Ramirez, Oriol Prat, Jesus Labarta and Mateo Valero. Performance Impact of the Interconnection Network on MareNostrum Applications. Proceedings of the HiPEAC Workshop on Interconnection Network Architectures, January 2007.
[27] Pieter Bellens, Josep M. Perez, Rosa Badia and Jesus Labarta. CellSs: a Programming Model for the Cell BE Architecture. Proceedings of the ACM/IEEE Supercomputing Conference (SC '06). November 2006.