Effective Communication and Computation Overlap with Hybrid MPI/SMPSs

Vladimir Marjanović[1,2], Jesús Labarta[1,2], Eduard Ayguadé[1,2], and Mateo Valero[1,2]

[1]Computer Sciences Department - Barcelona Supercomputing Center (BSC-CNS)
[2] Computer Architecture Department – Technical University of Catalunya (UPC)
{vladimir.marjanovic, jesus.labarta, eduard.ayguade, mateo.valero}@bsc.es

Abstract

Communication overhead is one of the dominant factors affecting 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 development effort and making less readable programs. This paper presents the hybrid use of MPI and SMPSs (SMP superscalar, a task-based shared-memory programming model) that allows the programmer to easily introduce the asynchrony necessary to overlap communication and computation. We demonstrate the hybrid use of MPI/SMPSs with the high-performance LINPACK benchmark (HPL), and compare it to the pure MPI implementation, which uses the look-ahead technique to overlap communication and computation. The hybrid MPI/SMPSs version significantly improves the performance of the pure MPI version, getting close to the asymptotic performance at medium problem sizes and still getting significant benefits at small/large problem sizes.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Classifications – concurrent, distributed, and parallel languages, data-flow languages; D.2.4 [**Computer-Communication Networks**]: Distributed Systems – distributed applications.

General Terms: Algorithms, Performance, Languages.

Keywords: parallel programming model, MPI, hybrid MPI/SMPSs, LINPACK.

1. Introduction and motivation

The Message Passing Interface (MPI) programming model has the widest practical acceptance for programming 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. To improve performance and scalability, programmers have to modify their application in order to: 1) overlap communication and computation and 2) accelerate the critical path in the computation. To achieve 1), programmers have to use the asynchronous (non-blocking) communication calls available in MPI: a communication request can be issued as soon as the data (or container for reception) is ready, then perform a computation not dependent on this data, and finally 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 less critical computations. The use of these techniques results in increased code complexity and in decreased programmer productivity.

The proposal in this paper achieves the potential performance benefits mentioned in the previous paragraph with minimal simple program annotations in pure MPI code. The annotations are from the SMPSs (SMP Superscalar [1]) 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 parallelism in a dataflow way. The proposed hybrid MPI/SMPSs enables the global asynchronous dataflow execution of both communication and computation tasks. Overlapping computation and communication is automatically achieved by the runtime system, which appropriately schedules communication and computation tasks in a dataflow way. In addition, the asynchrony of the model allows fast progress through the critical path of the application task graph. The implementation of the SMPSs runtime tailored to the hybrid approach is described in an extended version of this paper [2].

To demonstrate the benefits of the proposed hybrid MPI/SMPSs approach, both in terms of programming productivity and execution efficiency, we use HPL [3], a portable implementation of the high-performance LINPACK benchmark for distributed-memory computers. Additional results for varying number of processors as well as a study of the tolerance to network bandwidth and OS noise achieved by the proposed approach can also be found in [2].

2. Hybrid MPI/SMPSs programming model

MPI processes contain sequential computation regions between MPI calls; fine-grain shared memory parallelism in these regions can be exploited using for example a hybrid MPI/OpenMP approach. The fork/join execution model with barrier synchronizations in these hybrid approaches preclude the exploitation of parallelism across regions separated by MPI calls.

2.1 SMP Superscalar programming model

The SMPSs programming model [1] extends the standard C/Fortran programming language with a set of pragmas/directives to declare functions that are potential tasks:

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

and the intended use of the arguments of these functions through the following clauses:

PPoPP'10 January 9–14, 2010, Bangalore, India.
ACM 978-1-60558-708-0/10/01.

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

Two additional clauses are used to specify high priority when scheduling the task (`highpriority`) and to map it to certain architectural resources (`target(resource)`). An example for the two main functions in HPL is shown below:

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

Based on the input/output specifications and the actual arguments in each function invocation, the SMPSs 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 allow the exploitation of high degrees of distant parallelism, which is not possible when using barrier synchronization.

2.2 Taskifying MPI calls

The dataflow execution model in SMPSs can be effectively used to exploit the distant parallelism that may exist between tasks in different regions separated by MPI calls. In order to achieve this, MPI calls need to be encapsulated in SMPSs tasks. From the local point of view of a process, tasks sending data to another process should receive the buffer as an input argument. Tasks receiving data from other processes should specify the buffer as an output of the task. An example for *MPI_Recv* is shown below.

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

With this encapsulation, the SMPSs scheduler is able to reorder the execution of communication tasks relative to the computational tasks, just guaranteeing that the dependences are fulfilled. In this way, the programmer is relieved from the responsibility to schedule the communication requests, leading to simpler programs. At the global application level, MPI will impose synchronization between matching communication tasks. The fact that each of these tasks can be reordered with respect to the computation tasks enables the propagation of the asynchronous dataflow execution within each node to the whole MPI program.

Tasks that encapsulate blocking MPI calls have an unpredictable execution time (depending on the MPI synchronization with the matching call in the remote process). This may cause deadlock if we actually devote processors to these tasks and not to advance computational tasks. In order to solve the deadlock problem, we need to ensure that every process can always devote resources to the computational task such that local progress is guaranteed. A second effect of communication tasks is that they do not make an efficient use of processor time, wasting resources while they are blocked. It would be interesting to maximize the amount of actual computation performed while the data transfer activities are overlapped with it.

An implementation of the SMPSs runtime system tailored to the hybrid MPI/SMPSs model is detailed in [2]. It instantiates as many threads as cores in the node to execute computational tasks plus one additional thread to execute those tasks with `comm._thread` target. The implementation minimizes the contention between computation/communication threads while accelerating the execution of the communication threads as this would free local dependences.

3. Performance results

Figure 1 shows the GFlop/s achieved with the pure MPI and hybrid MPI/SMPSs approaches for the HPL benchmark using a two-dimensional data decomposition with different problem sizes. The evaluation is done using 512 processors of a cluster made of IBM JS21 blades (128 nodes, 4 cores each) and Myrinet interconnection network.

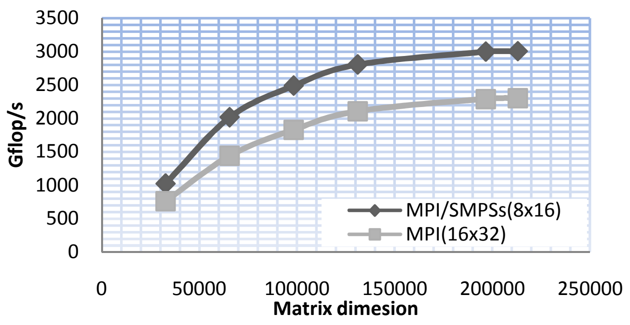

Figure 1.Performance rate of the LINPACK benchmark for two different versions (original HPL with look-ahead one and hybrid MPI/SMPSs).

We differentiate three regions in Figure 1:

- For very small matrices, the computation part of the application is small and there are not many possibilities to overlap communication and computation, which makes the network parameters (bandwidth and latency) the dominant factors. The hybrid MPI/SMPSs gets 5% better performance than the original MPI version.
- By increasing the problem size the hybrid MPI/SMPSs version exhibits its full strength against the pure MPI version with look-ahead. The hybrid version increases performance by 40% when N=131072.
- For the largest problem sizes we tried, the communication overhead is less dominant and as a consequence the gain of the hybrid MPI/SMPSs version goes down to 30% for N=212992.

Acknowledgments

This work is supported by the Spanish Ministry of Science and Innovation (contracts TIN2007-60625 and CSD2007-00050), the European Commission in the context of the HiPEAC Network of Excellence (contract IST-004408) and the HPC-Europa2 research infrastructure (contract 228398), and the MareIncognito project under the BSC-IBM collaboration agreement.

References

[1] J.M. Perez, L. Martinell, R.M. Badia and J. Labarta. A dependency aware task-based programming environment for multi-core architecture. *Proceedings of IEEE Cluster 2008*, September 2008.

[2] V. Marjanovic, J.M. Perez, E. Ayguadé, J. Labarta and M. Valero. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. *UPC-DAC-RR-2009-35 Research Report,* Technical University of Catalunya. May 2009.

[3] J. Dongarra, P. Luszczek and A. Petitet. The LINPACK Benchmark: Past, Present and Future. Concurrency and Computation: *Practice and Experience*. Vol. 15, issue 9, pp. 803–820. 2003.