Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters*

Nikolaos Drosinos and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Zografou Campus, Zografou 15773, Athens, Greece
e-mail: {ndros, nkoziris}@cslab.ece.ntua.gr

Abstract

This paper compares the performance of three programming paradigms for the parallelization of nested loop algorithms onto SMP clusters. More specifically, we propose three alternative models for tiled nested loop algorithms, namely a pure message passing paradigm, as well as two hybrid ones, that implement communication both through message passing and shared memory access. The hybrid models adopt an advanced hyperplane scheduling scheme, that allows both for minimal thread synchronization, as well as for pipelined execution with overlapping of computation and communication phases. We focus on the experimental evaluation of all three models, and test their performance against several iteration spaces and parallelization grains with the aid of a typical micro-kernel benchmark. We conclude that the hybrid models can in some cases be more beneficial compared to the monolithic pure message passing model, as they exploit better the configuration characteristics of an hierarchical parallel platform, such as an SMP cluster.

1 Introduction

Clusters have become the de-facto standard in parallel processing due to their high performance to price ratio. SMP clusters are also gaining on popularity, mainly under the assumption of fast interconnection networks and memory buses. SMP clusters can be thought of as an hierarchical two-level parallel architecture, since they combine features of shared and distributed memory machines. As a consequence, there is an active research interest in hybrid parallel programming models, e.g. models that perform communication both through message passing and memory access. Intuitively, a parallel paradigm that uses memory access for intra-node communication and message passing for inter-node communication seems to exploit better the characteristics of an SMP cluster.

The hybrid model has already been applied to real scientific applications ([7], [3], [11], [6]). Usually, programmers resort to MPI for the message passing communication, while OpenMP is becoming a popular interface for writing multi-threaded applications. Nevertheless, recent scientific work enlightens the complexity of the many aspects, that affect the overall performance of hybrid programs ([2], [10], [13]). The application of hybrid programming paradigms for the parallelization of algorithms and applications is, in many cases, inferior compared to a pure MPI parallelization, as MPI libraries tend to be highly optimized for message passing communication and provide poor support for thread management. The need for an MPI implementation that will efficiently support the hybrid model has been spotted by the research community ([14], [12]).

However, most of the work on the hybrid OpenMP-MPI programming paradigms addresses fine-grain parallelization, e.g. incremental parallelization of computationally intensive code parts through OpenMP work sharing constructs. Applications are submitted to performance profiling, the computationally intensive parts are extracted, and are further parallelized with the aid of OpenMP work sharing constructs. Furthermore, the hybrid parallelization of algorithms that impose data dependencies is avoided, mainly due to the high thread synchronization cost. Usually, only "embarrassingly parallel" algorithms are submitted to incremental OpenMP parallelization, where `DOALL` loops are distributed among OpenMP threads with the aid of work sharing constructs.

Nested loop algorithms represent an important class of computationally intensive scientific applications. Such algorithms usually impose various data dependencies, that

*This work was partially funded by IRAKLITOS - OP "EPEAEK II".

result to the need for frequent data exchange when parallelized. The typical parallelization of these algorithms involves the coarse-grain decomposition of the algorithm's computation into smaller subtasks, that are distributed to processors (tiled code form). Communication is done through message passing, where messages are exchanged per atomic tile computation, thus achieving a reduction in the total number of messages required. However, such pure MPI implementations fail to take into account the hierarchical architecture characteristics of an SMP cluster, as their nodes, despite having shared memory, perform both intra-node and inter-node communication through the message passing API. In some cases, certain MPI implementations allow for the exchange of intra-node messages through shared memory regions, but, either way, all communication is conducted through IPC mechanisms, and therefore incurs an unnecessarily high overhead for the case of MPI processes residing in the same SMP node.

In this paper we propose two hybrid MPI-OpenMP programming paradigms for the efficient parallelization of tiled nested loop algorithms, namely a fine-grain model, as well as a coarse-grain one. We address algorithms with constant flow data dependencies, which often result to relatively high communication needs. We further apply an advanced pipelined hyperplane scheduling, that allows for minimal overall completion time by reducing thread synchronization, while at the same time preserving the data dependencies of the original algorithm. In order to experimentally verify the relative performance of the hybrid models, we also propose an efficient pure MPI parallelization paradigm for our class of target applications, and perform an extensive performance comparison of all three models with the aid of a suitable micro-kernel benchmark. For each iteration space of the algorithm, we propose the parallelization approach that will deliver higher performance by estimating the communication overhead of each programming model.

The rest of the paper is organized as follows: Section 2 briefly presents our algorithmic model and our target architecture. Section 3 refers to our pure MPI parallelization paradigm, while Section 4 describes the two proposed hybrid parallelization paradigms, as well as the adopted pipelined hyperplane scheduling. Section 5 analyzes the experimental results obtained for the ADI micro-kernel benchmark, while Section 6 summarizes our conclusions and proposes future work.

2 Algorithmic model - Target architecture

Our parallelization paradigms can be applied to any n-dimensional nested loop, that has already been transformed using a *tiling* transformation to deliver a coarse-grain parallel equivalent program. Tiling is a popular loop transformation used to achieve coarse-grain parallelism on distributed memory machines and enhance data locality on uniprocessors. The general form of such parallel algorithms follows the scheme below:

```
FORACROSS tile_0 DO
  ...
    FORACROSS tile_{n-2} DO
      FOR tile_{n-1} DO
        Receive(tile);
        Compute(tile);
        Send(tile);
      END FOR
    END FORACROSS
  ...
END FORACROSS
```

The iteration space of the original algorithm has been partitioned into atomic execution units, called *tiles*. In the above code, tiles are identified by an n-dimensional vector $\vec{tile} = (tile_0, \ldots, tile_{n-1})$. FORACROSS implies parallel execution, as opposed to sequential execution (FOR - see also [15]). The parallel algorithm implements computation distribution across the $n-1$ outermost dimensions, and each processor computes a sequence of tiles along the innermost n-th dimension. It can be theoretically proved that, if the innermost dimension is the longest iteration space dimension, the above scheduling will deliver minimal overall execution time according to the UET-UCT model ([9]).

On the other hand, our target architecture concerns SMP clusters of PCs. We adopt a generic approach and assume num_nodes cluster nodes and $num\_threads$ threads of execution per node. Obviously, for a given SMP cluster architecture, one would probably select the number of execution threads to match the number of available CPUs in a node, but nevertheless our approach considers, for the sake of generality, both the number of nodes as well as the number of execution threads per node to be user-defined parameters.

3 Pure MPI parallelization

Pure MPI parallelization of the algorithms described above is based on the tiling transformation. The complete methodology is described more extensively in [5]. It must be noted that, since our prime objective was to experimentally verify the performance benefits of the different parallelization models, for the sake of simplicity we resorted to hand-made parallelization, as opposed to automatic parallelization. Nevertheless, all parallelization models can be automatically generated with minimal compilation time overhead according to the work presented in [4], which reflects the automatic parallelization method for the pure MPI model and can easily be applied in the hybrid model, as well.

Furthermore, the advanced *pipelined* scheduling scheme ([9]) is adopted as follows: In each time step, an MPI process concurrently computes a tile, receives data required for the computation of the next tile and sends data computed at the previous tile. For the true overlapping of computation and communication, as theoretically implied by the above scheme, non-blocking MPI communication primitives are used and DMA support is assumed. Unfortunately, we used the MPICH implementation over FastEthernet (ch_p4 ADI-2 device), which does not support such advanced DMA-driven non-blocking communication, but nevertheless the same limitations hold for our hybrid model and are thus not likely to affect the performance comparison.

Let $\vec{tile} = (tile_0, \ldots, tile_{n-1})$ identify the coordinates of a tile, $\vec{nod} = (nod_0, \ldots, nod_{n-2})$ identify an MPI process in cartesian coordinates and $x = (x_0, \ldots, x_{n-1})$ denote the tile size. The core of the pure MPI code, under the pipelined scheduling scheme ([9]), resembles the following:

```
tile_0 = nod_0;
...
tile_{n-2} = nod_{n-2};
FOR tile_{n-1} = 0 TO ⌊(max_{n-1} − min_{n-1}) / x_{n-1}⌋ DO
   Pack(snd_buf, tile_{n-1} − 1, nod);
   MPI_Isend(snd_buf, dest(nod));
   MPI_Irecv(recv_buf, src(nod));
   Compute(tile);
   MPI_Waitall;
   Unpack(recv_buf, tile_{n-1} + 1, nod);
END FOR
```

Note that each MPI process assumes the execution of tiles successive along the n-th coordinate ($tile_{n-1}$). Equivalently, all tiles with the same $n-1$ outermost coordinates ($tile_0, \ldots, tile_{n-2}$) are mapped to the same MPI process, and thus these $n-1$ coordinates can be used to identify the MPI rank of the owner process, as shown in the above code.

4 Hybrid MPI-OpenMP parallelization

The hybrid MPI-OpenMP programming model intuitively matches the characteristics of a cluster of SMP nodes, since it allows for a two-level communication pattern that distinguishes between intra- and inter-node communication. More specifically, intra-node communication is implemented through common access to each node's shared memory, and appropriate synchronization is used to ensure that data are first calculated and then used, so that the execution order of the initial algorithm is preserved (thread-level synchronization). Inversely, inter-node communication is achieved through message passing between different nodes, that implicitly enforces node-level synchronization to ensure valid data access and execution order.

The different approach of the hybrid models compared to the pure message passing one is schematically depicted in Figure 1, where a three dimensional space is mapped on two dual SMP nodes. Note that in both cases processors assume the execution of tile sequences along dimension Z. Note also that the hybrid model eliminates message passing communication across the XZ surface.

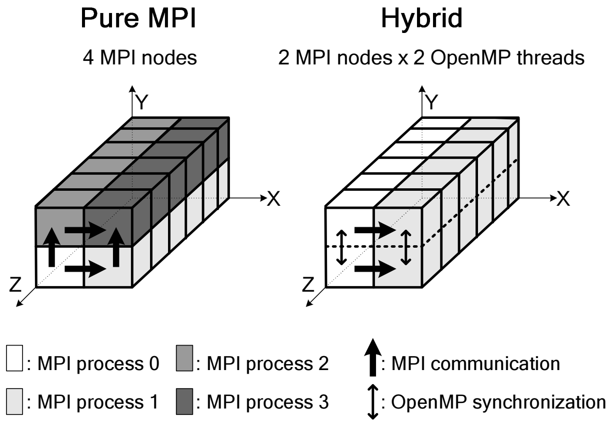

Figure 1. Pure MPI vs Hybrid on two dual SMP nodes

We propose two main variations of the hybrid model, namely a *fine-grain* hybrid model and a *coarse-grain* one. According to the fine-grain model, the computationally intensive parts of the pure MPI code are incrementally parallelized with OpenMP work-sharing directives. According to the coarse-grain model, threads are spawned close to the creation of the MPI processes, and the thread ids are used to enforce an SPMD structure in the hybrid program, similar to the structure of the pure MPI code.

Both hybrid models implement the advanced *hyperplane* scheduling presented in [1], that allows for minimal overall completion time. The hyperplane scheduling, along with the variations of the hybrid model are the subject of the following Subsections.

4.1 Hyperplane scheduling

The proposed hyperplane scheduling distributes the tiles assigned to all threads of a specific process into *groups* that can be concurrently executed. Each group contains all tiles that can be safely executed in parallel by an equal number of threads without violating the data dependencies of the initial algorithm. In a way, each group can be considered as a distinct time step of a process's execution sequence, and determines which threads of that process will be executing a tile at that time step, and which ones will remain idle. This scheduling aims at minimizing the total number of execution steps required for the completion of the hybrid algorithms.

For our hybrid model, each group of tiles will be identified by an n-dimensional vector $\vec{group}$. The first n-1 coordinates of $\vec{group}$ will identify the particular MPI process $\vec{nod}$ this group refers to, while the last coordinate corresponds to the current time step and will implicitly determine whether a given thread $\vec{th} = (th_0, \ldots, th_{n-2})$ of process $\vec{nod}$ will be computing at that time step, and if so which tile $\vec{tile}$. Formally, given a group denoted by the n-dimensional vector $\vec{group} = (group_0, \ldots, group_{n-1})$, the corresponding MPI process $\vec{nod}$ can be determined by the first n-1 coordinates of $\vec{group}$, namely
$nod_i = group_i, 0 \le i \le n-2$
and the tile $\vec{tile}$ to be executed by OpenMP thread $\vec{th}$ of $\vec{nod}$ can be obtained by
$tile_i = group_i \times m_i + th_i, 0 \le i \le n-2$
and
$tile_{n-1} = group_{n-1} - \sum_{i=0}^{n-2}(group_i \times m_i + th_i)$
where m_i the number of threads along the i-th dimension (it holds $0 \le th_i \le m_i - 1, 0 \le i \le n-2$).

The value of $tile_{n-1}$ will establish whether thread $\vec{th}$ will compute during group $\vec{group}$: If the calculated tile is valid, namely if it holds $0 \le tile_{n-1} \le \lfloor \frac{max_{n-1} - min_{n-1}}{t} \rfloor$, then $\vec{th}$ will execute tile $\vec{tile}$ at time step $\vec{group}_{n-1}$. In the opposite case, it will remain idle and wait for the next time step.

The hyperplane scheduling can be implemented in OpenMP according to the following pseudo-code scheme:

```
#pragma omp parallel num_threads(num_threads)
{
    group_0 = nod_0;
    ...
    group_{n-2} = nod_{n-2};
    tile_0 = nod_0 * m_0 + th_0;
    ...
    tile_{n-2} = nod_{n-2} * m_{n-2} + th_{n-2};
    FOR(group_{n-1}){
        tile_{n-1} = group_{n-1} - Σ_{i=0}^{n-2} tile_i;
        if(0 ≤ tile_{n-1} ≤ ⌊(max_{n-1} - min_{n-1})/t⌋)
            Compute(tile);
        #pragma omp barrier
    }
}
```

The hyperplane scheduling is more extensively analyzed in [1].

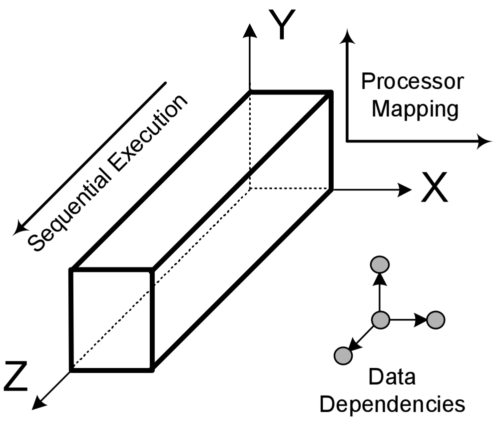

Figure 2. Iteration space of ADI

4.2 Fine-grain hybrid parallelization

The fine-grain hybrid implementation applies an OpenMP `parallel` work-sharing construct to the tile computation of the pure MPI code. According to the hyperplane scheduling described in Subsection 4.1, at each time step corresponding to a group instance, the required threads that are needed for the tile computations are spawned. Inter-node communication occurs outside the OpenMP parallel region, so only a trivial MPI thread support level is required (sometimes referred to as *MASTERONLY*).

Note that the hyperplane scheduling ensures that all computations, concurrently executed by different threads, do not violate the execution order of the original algorithm. The required barrier for the thread synchronization is implicitly enforced by exiting the OpenMP `parallel` construct. Note also that, under the fine-grain approach, there is an overhead of the threads data structures re-initialization for each time step of the pipelined schedule.

The code of the hybrid fine-grain model resembles the following:

```
group_0 = nod_0;
...
group_{n-2} = nod_{n-2};
/*for all time steps in current node*/
FOR(group_{n-1}){
  /*pack previously computed data*/
  Pack(snd_buf, tile_{n-1} - 1, nod);
  /*send communication data*/
  MPI_Isend(snd_buf, dest(nod));
  /*receive data for next tile*/
  MPI_Irecv(recv_buf, src(nod));
  #pragma omp parallel
  {
    tile_0 = nod_0 * m_0 + th_0;
```

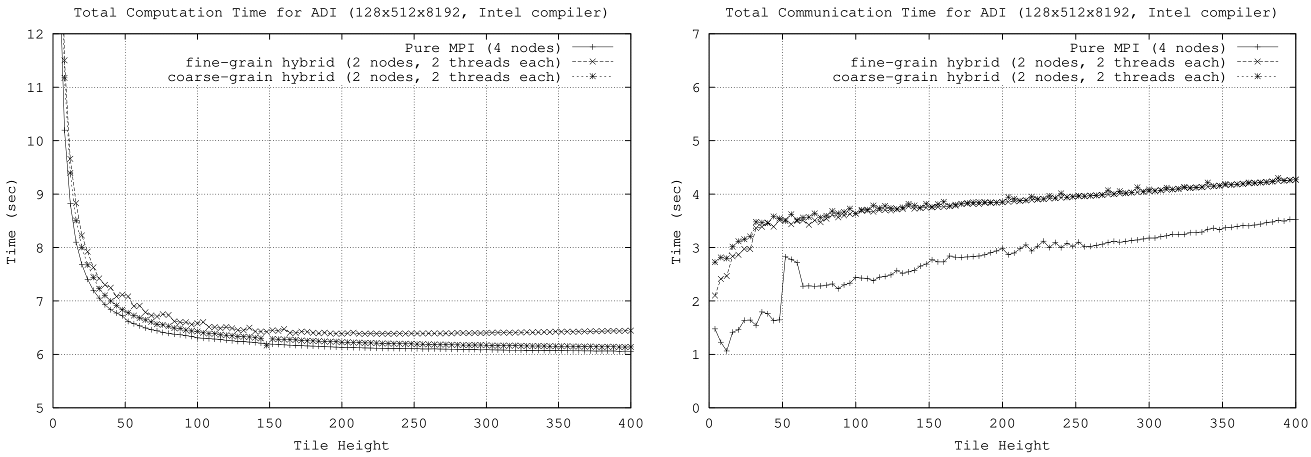

Figure 13. Computation vs communication profiling for ADI (128x512x8192)

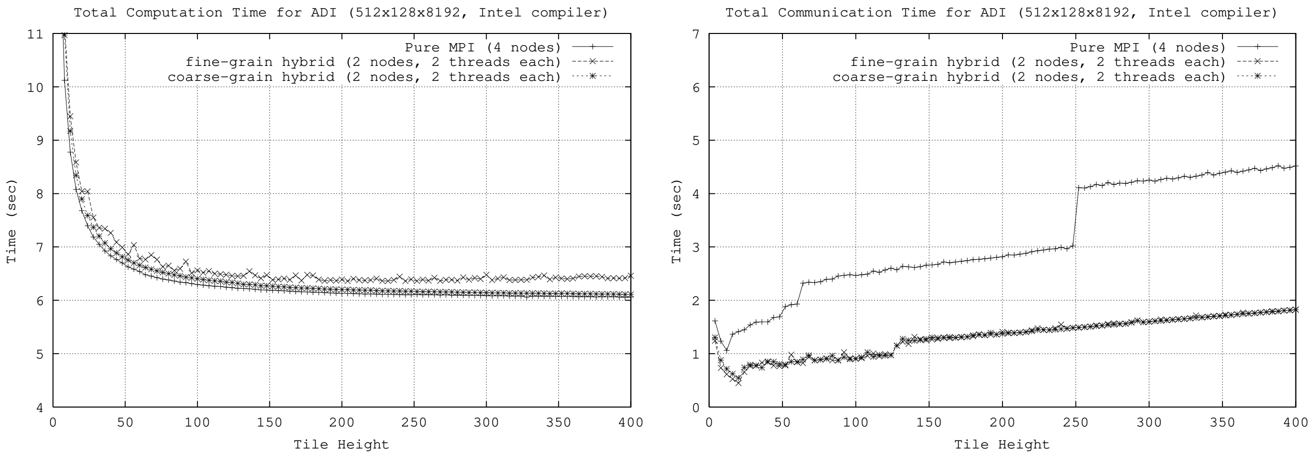

Figure 14. Computation vs communication profiling for ADI (512x128x8192)

...
$tile_{n-2}$ = nod_{n-2} * m_{n-2} + th_{n-2};
*/*calculate candidate tile for execution*/*
$tile_{n-1}$ = $group_{n-1}$ - $\sum_{i=0}^{n-2} tile_i$;
*/*if current thread is to execute a valid tile*/*
if $(0 \leq tile_{n-1} \leq \lfloor \frac{max_{n-1} - min_{n-1}}{t} \rfloor)$
*/*compute current tile*/*
Compute($\vec{tile}$);
}
*/*wait for communication completion*/*
MPI_Waitall;
*/*unpack communication data*/*
Unpack(recv_buf, $tile_{n-1}+1$, $\vec{nod}$);
}

As aforementioned, all MPI communication lies outside the OpenMP parallel construct. Also, note that the OpenMP `parallel` directive for the thread creation lies inside an iterative `FOR` loop.

4.3 Coarse-grain hybrid parallelization

According to the coarse-grain model, threads are only spawned once and their ids are used to determine their flow of execution in the SPMD-like code. That is, their OpenMP thread ids are used to determine both their work share, as well as their communication part. Inter-node communication occurs within the OpenMP `parallel` region, but is completely assumed by the master thread by means of the OpenMP `master` directive. The reason for this is that the MPICH implementation used provides at best an MPI_THREAD_FUNNELED level of thread safety, allowing only the master thread to call MPI routines. Intra-node synchronization between the threads is achieved with the aid of an OpenMP `barrier` directive.

It should be noted that the coarse-grain model, as com-

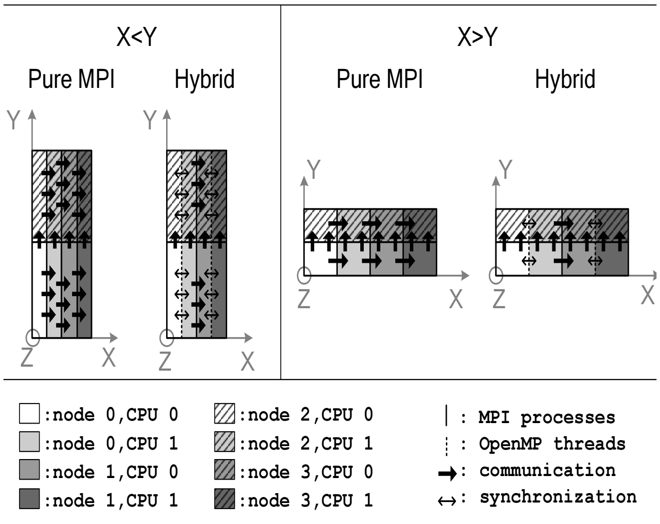

Figure 3. Processor mapping on 4 dual SMP nodes

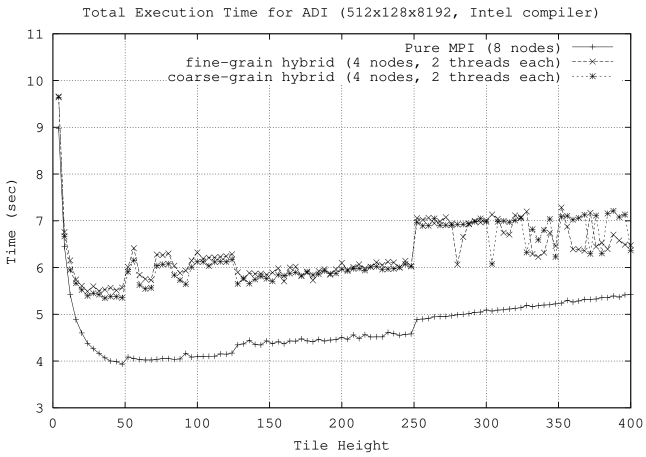

Figure 4. Total execution time for ADI (512x128x8192, 4 SMP nodes)

pared to the fine-grain one, compensates the relatively higher programming complexity with the fact that threads are initialized only once, thus the respective overhead of the fine-grain model is diminished. Furthermore, although communication is entirely assumed by the master thread, the other threads will be able to perform computation at the same time, since they have already been spawned (unlike the fine-grain model). An MPI implementation with MPI_THREAD_MULTIPLE thread support level could potentially allow for a much more efficient communication scheme, according to which all threads would be able to call MPI routines, although higher message latencies would be anticipated in such a case. Alternatively, given a non thread-safe environment, a more sophisticated load-balancing scheme, that would compensate for the master-only communication with appropriately balanced computation distribution, is being considered as future work.

The coarse-grain model can be implemented with the aid of a code structure similar to the following:

```
#pragma omp parallel
{
  group_0 = nod_0;
  ...
  group_{n-2} = nod_{n-2};
  tile_0 = nod_0 * m_0 + th_0;
  ...
  tile_{n-2} = nod_{n-2} * m_{n-2} + th_{n-2};
  /*for all time steps in current node*/
  FOR(group_{n-1}){
    /*calculate candidate tile for execution*/
    tile_{n-1} = group_{n-1} - sum_{i=0}^{n-2} tile_i;
    #pragma omp master
    {
      /*pack previously computed data*/
      Pack(snd_buf, tile_{n-1} - 1, nod);
      /*send communication data*/
      MPI_Isend(snd_buf, dest(nod));
      /*receive data for next tile*/
      MPI_Irecv(recv_buf, src(nod));
    }
    /*if current thread is to execute a valid tile*/
    if(0 <= tile_{n-1} <= floor((max_{n-1} - min_{n-1}) / t))
      /*compute current tile*/
      Compute(tile);
    #pragma omp master
    {
      /*wait for communication completion*/
      MPI_Waitall;
      /*unpack communication data*/
      Unpack(recv_buf, tile_{n-1} + 1, nod);
    }
    /*synchronize threads for next time step*/
    #pragma omp barrier
  }
}
```

5 Experimental results

We have tested the actual performance of all three programming paradigms with the aid of a micro-kernel benchmark, namely Alternating Direction Implicit (ADI - [8]). ADI is a simple three-dimensional perfectly nested

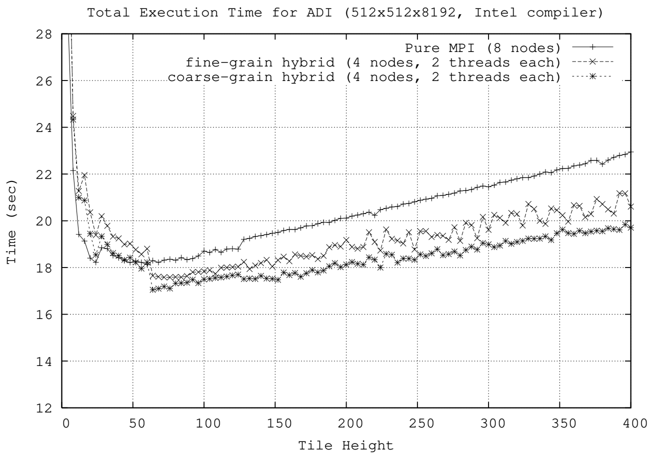

Figure 5. Total execution time for ADI (512x512x8192, 4 SMP nodes)

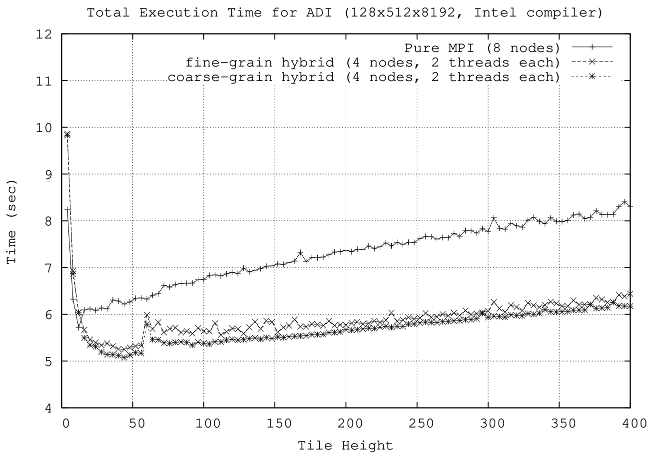

Figure 6. Total execution time for ADI (128x512x8192, 4 SMP nodes)

loop algorithm, that imposes unitary inter-tile dependencies across all three space directions. It has an iteration space of $X \times Y \times Z$, where Z is considered to be the longest algorithm dimension (Figure 2). Schematically, ADI can be implemented with the aid of following code:

```
FOR x = 0 TO X DO
  FOR y = 0 TO Y DO
    FOR z = 0 TO Z DO
      A[x,y,z]=f(A[x-1,y,z], A[x,y-1,z], A[x,y,z-1]);
    ENDFOR
  ENDFOR
ENDFOR
```

We chose to experimentally verify the efficiency of all proposed models with ADI, as it is a typical micro-kernel benchmark for tiled nested loop algorithms. Most importantly, ADI imposes communication in all three unitary directions. Consequently, all parallel implementations of ADI include a significant amount of communication, and thus allow the comparison of the three proposed programming paradigms, which substantially differ in the communication process and the particular data exchange approach (message passing as opposed to access to the shared memory of a node). Although ADI has a well defined, regular communication scheme, we expect our parallelization paradigms to deliver high performance even for algorithms with more irregular, process dependent communication patterns, as long as they span the entire iteration space (that is, some communication should be required along all unitary space dimensions). If no communication is required along one or more loop dimensions, then the proposed scheduling schemes are expected to incur an additional unnecessary overhead, as far as the overall performance is concerned.

Our experimental platform is a Pentium III dual-SMP cluster of 4 nodes. Each node has 2 Pentium III CPUs at 800 MHz, 128 MB of RAM and 256 KB of cache, and runs Linux with 2.4.20 kernel. We used Intel icc compiler version 7.0 for Linux with the following optimization flags: `-O3 -mpcu=pentiumpro -static`. Finally, we used MPI implementation MPICH v. 1.2.5, configured with the following options: `--with-device=ch_p4 --with-comm=shared`. Note that by configuring MPICH with the `--with-comm=shared` option, we ensured that all intra-node communication between MPI processes of the same SMP node would be achieved with the aid of shared memory segments in the pure MPI model.

We tried several iteration spaces for the ADI micro-kernel benchmark, and ran all parallel versions on four SMP nodes and on two SMP nodes. Moreover, for each iteration space and programming model, we tested variable tile heights in order to determine the optimal granularity for the per-tile computation to communication ratio. The following subsections illustrate the obtained experimental results for all aforementioned cases.

5.1 Performance comparison on four SMP nodes

In the first case, we ran all parallel paradigms of ADI onto four SMP nodes, that is, eight processors in total. Task decomposition and mapping to processors is applied on the XY surface (Figure 2). According to our computation distribution scheme, processors assume the execution of sequences of tiles that are successive along the dimension Z of the algorithm. We chose Z to be 8K (e.g. 8192 iterations), while X and Y ranged from 128 to 512, resulting to a total number of iterations between 500 millions and two

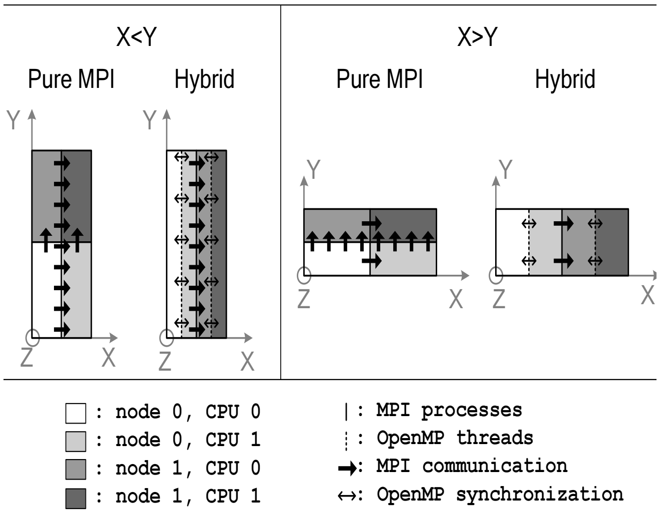

Figure 7. Processor mapping on 2 dual SMP nodes

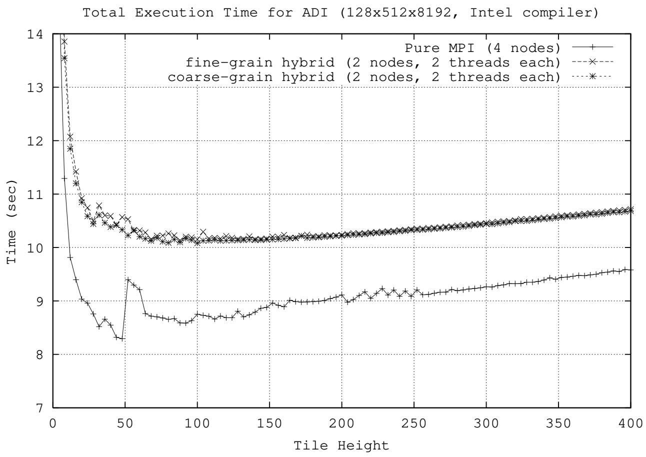

Figure 8. Total execution time for ADI (128x512x8192, 2 SMP nodes)

billions. For the pure MPI model, we considered a 4×2 cartesian topology for the eight MPI processes. On the other hand, for the hybrid models we considered a 2×2 cartesian topology for the four MPI processes, and a 2×1 topology for the OpenMP threads in each MPI process. The processor mapping for both cases is shown in Figure 3.

As shown in Figure 3, for larger values of X we expected the pure MPI model to perform better under the assumed mapping, as the hybrid models not only fail to relieve communication across the long X dimension, but further use half the MPI processes (four as opposed to eight) to perform that message passing communication. Indeed, as shown in Figure 4, for $X = 512$ and $Y = 128$, the pure MPI model outperforms the hybrid variations. For $X = Y = 512$ (Figure 5), we notice that all three implementations perform quite similarly, as the efficient message passing communication of the pure MPI model across dimension X is fully mitigated by the faster intra-node communication of the hybrid models across dimension Y. However, for large values of Y, the hybrid implementations clearly outperform the pure MPI one (Figure 6) due to the lightweight intra-node communication across the long Y dimension of the hybrid models. More specifically, the hybrid models perform intra-node communication through common access to the SMP node's memory, and therefore significantly relieve the heavy communication load across the long Y dimension.

5.2 Performance comparison on two SMP nodes

We have also performed a series of experiments on two SMP nodes (four processors in total). The reason for doing that is that generally we have chosen the cartesian topology for the pure MPI paradigm in a scheduling-efficient way, so as to minimize the overall startup latency of the parallel algorithm. In the previous case, e.g. when we had eight MPI processes, one could argue that there were two alternatives for the selection of the optimal topology, namely 4×2, as well as its symmetric one (2×4). However, with four MPI processes, there is only one way to apply the scheduling optimal cartesian topology, namely a 2×2 topology. For the hybrid models, we consider a 2×1 MPI process topology, and a 2×1 OpenMP thread topology in each MPI process (2 MPI processes $\times$ 2 OpenMP threads per MPI process). Processor mapping on the XY surface for the pure MPI model and the hybrid ones is schematically depicted in Figure 7.

As shown in Figure 7, for larger values of Y we expect the pure MPI model to perform better, since the heavy message passing communication is not relieved by the hybrid models and the pure MPI program uses twice as many MPI processes to perform that communication. In the opposite case, that is, for larger values of X, the hybrid models are expected to deliver higher performance, since they totally eliminate all message passing communication across the long X dimension.

The experimental results verified the above intuition. Starting from a relatively large value of dimension Y, the pure MPI model delivers better performance (Figure 8). However, as we move to higher values of X (Figure 9), the hybrid models are gaining on performance, as anticipated. For $X = Y$ (Figure 10), all three implementations perform

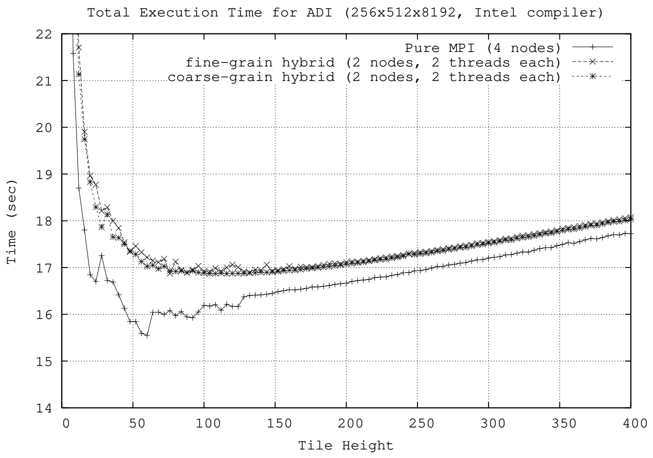

Figure 9. Total execution time for ADI (256x512x8192, 2 SMP nodes)

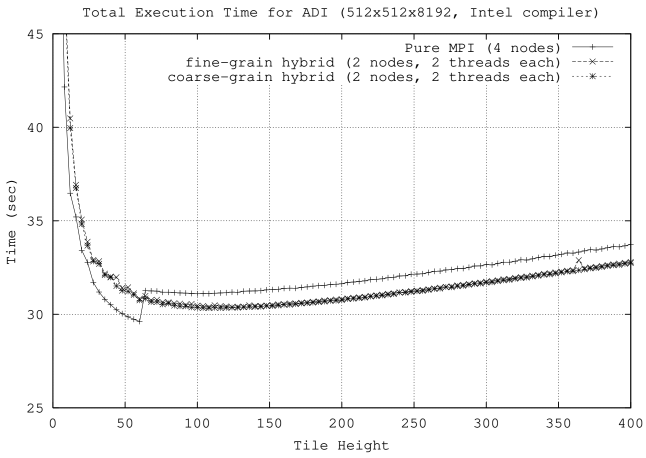

Figure 10. Total execution time for ADI (512x512x8192, 2 SMP nodes)

equally well. When X becomes larger than Y (Figure 11), we notice that the hybrid models outperform the pure MPI one, as they totally eliminate all message passing communication across the long X dimension. The performance benefits of the hybrid models become more obvious as X grows notably larger than Y (Figure 12).

In order to fortify our theoretical explanation of the obtained results, we further conducted some partial profiling of the computation and communication times. In both cases (Figure 13 for large values of Y and Figure 14 for large values of X), we noticed that partial computation times were equal in all three programming models, and thus the differences in the overall execution times can be attributed to the differences in communication, as was theoretically anticipated. Note also that the coarse-grain hybrid model performs slightly better than the fine-grain one, as far as the computational part is concerned, since it avoids the overhead of thread re-initialization. Nevertheless, the advantage of the coarse-grain model compared to the fine-grain one was relatively small, and did not deliver a significant performance gain. Last, the individual profiling of computation and communication times reveals that the overall performance degradation at certain tile heights (peaks and troughs at the graphs) can be attributed to the respective communication irregularities. For instance, the rapid performance degradation in the pure MPI model when transiting from $z = 248$ to $z = 252$ (Figure 12) is due to the respective communication bandwidth reduction (about 0,5 MB/s), according to further ping-pong bandwidth measurements that were conducted.

6 Conclusions-Future work

In this paper we have tested the performance of three programming models for the parallelization of nested loop algorithms with flow data dependencies. Depending on the particular iteration space of the algorithm, as well as the assumed processor mapping, the appropriate selection between the pure MPI model and a hybrid one must be met in order to obtain higher performance. Generally, hybrid programming models can potentially match better the architecture characteristics of an SMP cluster under an appropriate processor mapping, that would replace message passing communication with synchronized thread-level memory access. On the other hand, hybrid models are not efficiently supported by existing MPI implementations, resulting to an imbalanced message passing that is performed solely by the master thread.

We intend to apply a more efficient load balancing scheme for the coarse-grain model, that will mitigate the imbalance of the master-only message passing communication: Since the master thread assumes all message passing communication under an MPI_THREAD_FUNNELED thread support level, it should assume a smaller fraction of the computational part for a more efficient overall load balancing between threads of the same node. Last, we intend to investigate the performance of the various models on advanced interconnection networks, such as SCI and Myrinet.

References

[1] M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, and N. Koziris. Pipelined scheduling of tiled nested loops onto clusters of SMPs using memory mapped network interfaces.

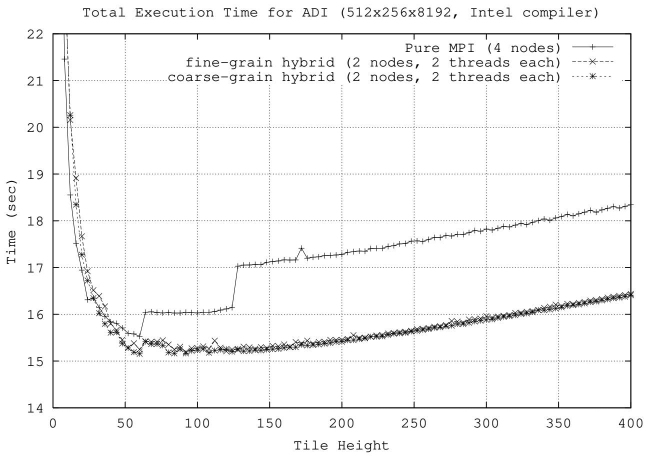

Figure 11. Total execution time for ADI (512x256x8192, 2 SMP nodes)

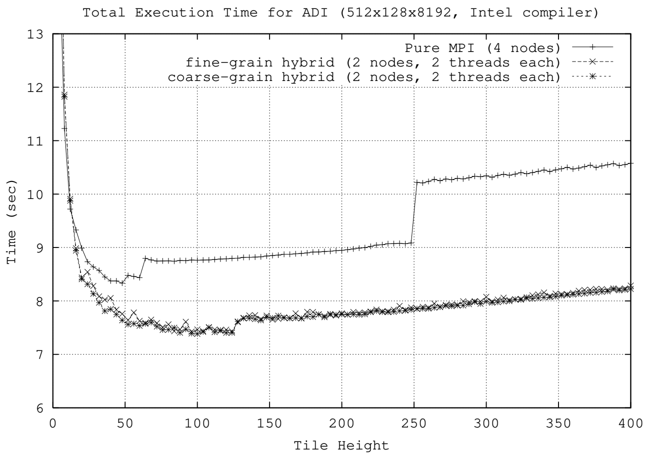

Figure 12. Total execution time for ADI (512x128x8192, 2 SMP nodes)

In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, USA, 2002. IEEE Computer Society Press.

[2] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Dallas, Texas, USA, 2000. IEEE Computer Society Press.

[3] S. Dong and G. E. Karniadakis. Dual-level parallelism for deterministic and stochastic CFD problems. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, USA, 2002. IEEE Computer Society Press.

[4] G. Goumas, M. Athanasaki, and N. Koziris. Automatic Code Generation for Executing Tiled Nested Loops Onto Parallel Architectures. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2002)*, Madrid, Mar 2002.

[5] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Compiling Tiled Iteration Spaces for Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 360–369, Chicago, Illinois, Sep 2002.

[6] Y. He and C. H. Q. Ding. MPI and OpenMP paradigms on cluster of SMP architectures: the vacancy tracking algorithm for multi-dimensional array transposition. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, USA, 2002. IEEE Computer Society Press.

[7] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 10, Dallas, Texas, United States, 2000.

[8] G. E. Karniadakis and R. M. Kirby. *Parallel Scientific Computing in C++ and MPI : A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge University Press, 2002.

[9] N. Koziris, A. Sotiropoulos, and G. Goumas. A Pipelined Schedule to Minimize Completion Time for Loop Tiling with Computation and Communication Overlapping. *Journal of Parallel and Distributed Computing*, 63(11):1138–1151, Nov. 2003.

[10] G. Krawezik and F. Cappello. Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. In *ACM SPAA 2003*, San Diego, USA, Jun 2003.

[11] R. D. Loft, S. J. Thomas, and J. M. Dennis. Terascale spectral element dynamical core for atmospheric general circulation models. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 18–18, Denver, Colorado, 2001. ACM Press.

[12] B. V. Protopopov and A. Skjellum. A multi-threaded Message Passing Interface (MPI) architecture: performance and program issues. *JPDC*, 2001.

[13] R. Rabenseifner and G. Wellein. Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures. *International Journal of High Performance Computing Applications*, 17(1):49–62, 2003.

[14] H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In *Proceedings of the 15th international conference on Supercomputing*, pages 381–392, Sorrento, Italy, 2001. ACM Press.

[15] P. Tang and J. Zigman. Reducing Data Communication Overhead for DOACROSS Loop Nests. In *Proc. of the International Conference on Supercomputing (ICS'94)*, pages 44–53, Manchester, UK, Jul. 1994.