Advanced Hybrid MPI/OpenMP Parallelization Paradigms for Nested Loop Algorithms onto Clusters of SMPs

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. The parallelization process of nested-loop algorithms onto popular multi-level parallel architectures, such as clusters of SMPs, is not a trivial issue, since the existence of data dependencies in the algorithm impose severe restrictions on the task decomposition to be applied. In this paper we propose three techniques for the parallelization of such algorithms, namely pure MPI parallelization, fine-grain hybrid MPI/OpenMP parallelization and coarse-grain MPI/OpenMP parallelization. We further apply an advanced hyperplane scheduling scheme that enables pipelined execution and the overlapping of communication with useful computation, thus leading almost to full CPU utilization. We implement the three variations and perform a number of micro-kernel benchmarks to verify the intuition that the hybrid programming model could potentially exploit the characteristics of an SMP cluster more efficiently than the pure message-passing programming model. We conclude that the overall performance for each model is both application and hardware dependent, and propose some directions for the efficiency improvement of the hybrid model.

1 Introduction

Clusters of SMPs have emerged as the dominant high performance computing platform. For these platforms there is an active research concern that traditional parallelization programming paradigms may not deliver optimal performance, since pure message-passing parallel applications fail to take into consideration the 2-level SMP cluster architecture. Intuitively, a parallel paradigm that uses memory access for intra-node communication and message-passing for inter-node communication seems to match better the characteristics of an SMP cluster. Since MPI has become the de-facto message passing API, while OpenMP has grown into a popular multi-threading library, there is important scientific work that addresses the hybrid MPI/OpenMP programming model.

The hybrid model has already been applied to real scientific applications ([3], [6]). Nevertheless, a lot of important scientific work enlightens the complexity of the many aspects that affect the overall performance of hybrid programs ([2], [8], [10]). Also, the need for a multi-threading MPI implementation that will efficiently support the hybrid model has been spotted by the research community ([11], [9]).

However, most of the work on the hybrid OpenMP/MPI programming paradigm addresses fine-grain parallelization, e.g. usually incremental parallelization of computationally intensive code parts through OpenMP work sharing constructs. On the other hand, programming paradigms that allow parallelization and work distribution across the entire SMP cluster would be more beneficial in case of nested loop algorithms. The pure MPI code is generic enough to apply in this case too, but as aforementioned does not take into account the particular architectural features of an SMP cluster.

In this paper we propose two hybrid MPI/OpenMP programming paradigms for the efficient parallelization of perfectly nested loop algorithms, namely a fine-grain model, as well as a coarse-grain one. We further apply an advanced pipelined hyperplane scheduling that allows for minimal overall completion time. Finally, we conduct some experiments in order to verify the actual performance of each 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 the pure MPI parallelization paradigm, while Section 4 describes the variations of the hybrid parallelization paradigm, as well as the adopted pipelined hyperplane schedule. 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 model concerns n-dimensional perfectly nested loops with uniform data dependencies of the following form:

```
FOR j_0 = min_0 TO max_0 DO
   FOR j_1 = min_1 TO max_1 DO
      ...
      FOR j_{n-1} = min_{n-1} TO max_{n-1} DO
         Computation(j_0, j_1, ..., j_{n-1});
      ENDFOR
      ...
   ENDFOR
ENDFOR
```

The loop computation is a calculation involving an n-dimensional matrix A which is indexed by $j_0, j_1, \ldots, j_{n-1}$. For the ease of the analysis to follow, we assume that we deal with rectangular iteration spaces, e.g. loop bounds min_i, max_i are constant ($0 \leq i \leq n-1$). We also assume that the loop computation imposes arbitrary constant *flow* dependencies, that is the calculation at a given loop instance may require the values of certain elements of matrix A computed at previous iterations.

Our target architecture concerns an SMP cluster 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 paradigm

Pure MPI parallelization is based on the tiling transformation. Tiling is a popular loop transformation used to achieve coarse-grain parallelism on multi-processors and enhance data locality on uni-processors. Tiling partitions the original iteration space into atomic units of execution, called tiles. Each MPI node assumes the execution of a sequence of tiles, successive along the longest dimension of the original iteration space. 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, an advanced pipelined scheduling scheme is adopted as follows: In each time step, an MPI node 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, the MPICH implementation over FastEthernet (ch_p4 ADI-2 device) 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 $\boldsymbol{tile} = (tile_0, \ldots, tile_{n-1})$ identify a tile, $\boldsymbol{nod} = (nod_0, \ldots, nod_{n-2})$ identify an MPI node in Cartesian coordinates and $x = (x_0, \ldots, x_{n-1})$ denote the tile size. For the control index j_i of the original loop it holds $j_i = tile_i * x_i + min_i$, where $min_i \leq j_i \leq max_i$ and $0 \leq i \leq n-1$. The core of the pure MPI code 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
```

4 Hybrid MPI/OpenMP paradigm

The hybrid MPI/OpenMP programming model intuitively matches the characteristics of an SMP cluster, 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 the node's memory, and appropriate synchronization must 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, that implicitly enforces node-level synchronization to ensure valid execution order as far as the different nodes are concerned.

There are two main variations of the hybrid model, namely the *fine-grain* hybrid model and the *coarse-grain* one. According to the fine-grain model, the computationally intensive parts of the pure MPI code are usually 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 node 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 thought as of a distinct time step of a node's execution sequence, and determines which threads of that node 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 will be identified by an n-dimensional vector, where the first n-1 coordinates will identify the particular MPI node $\boldsymbol{nod}$ this group refers to, while the last coordinate can be thought of as the current time step and will implicitly determine whether a given thread $\boldsymbol{th} = (th_0, \ldots, th_{n-2})$ of $\boldsymbol{nod}$ will be computing at that time step, and if so which tile $\boldsymbol{tile}$. Formally, given a group denoted by the n-dimensional vector $\boldsymbol{group} = (group_0, \ldots, group_{n-1})$
the corresponding node $\boldsymbol{nod}$ can be determined by the first n-1 coordinates of $\boldsymbol{group}$, namely
$nod_i = group_i, 0 \leq i \leq n-2$
and the tile $\boldsymbol{tile}$ to be executed by thread $\boldsymbol{th}$ of $\boldsymbol{nod}$ can be obtained by
$tile_i = group_i \times m_i + th_i, 0 \leq i \leq n-2$
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 to 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 $\boldsymbol{th}$ will compute during group $\boldsymbol{group}$: 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 $\boldsymbol{th}$ will execute tile $\boldsymbol{tile}$ at time step $\boldsymbol{group_{n-1}}$. 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)
{
   group0 = nod0;
   ...
   group_{n-2} = nod_{n-2};
   tile0 = nod0 * m0 + th0;
   ...
   tile_{n-2} = nod_{n-2} * m_{n-2} + th_{n-2};
   FOR(group_{n-1}){
      tile_{n-1} = group_{n-1} - sum_{i=0}^{n-2} tile_i;
      if(0 <= tile_{n-1} <= floor((max_{n-1} - min_{n-1}) / t))
         compute(tile);
#pragma omp barrier
   }
}
```

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

4.2 Fine-grain parallelization

The pseudo-code for the fine-grain hybrid parallelization is depicted in Table 1. 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.

Note that the hyperplane scheduling ensures that all calculations 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 re-spawning threads for each time step of the pipelined schedule.

4.3 Coarse-grain parallelization

The pseudo-code for the coarse-grain hybrid parallelization is depicted in Table 2. Threads are only spawned once and their ids are used to determine their flow of execution in the SPMD-like code. 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 compared to the fine-grain one, compensates the relatively higher programming complexity with the fact that threads are created only once, and 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). Naturally, a thread-safe MPI implementation would allow for a much more efficient communication scheme, according to which all threads would be able to call MPI routines. Alternatively, under 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.

5 Experimental results

In order to evaluate the actual performance of the different programming paradigms, we have parallelized the Alternating Direction Implicit (ADI) Integration micro-kernel ([7]) and run several experiments for different iteration spaces and tile grains. Our platform is an 8-node dual-SMP cluster. 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 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`.

We performed two series on experiments in order to evaluate the relative performance of the three parallelization methods. More specifically, in the first case we used 8 MPI processes for the MPI model (1 process per SMP node), while we used 4 MPI processes $\times$ 2 OpenMP threads for the hybrid models. In the second case, we started the pure MPI program with 16 MPI processes (2 per SMP node), while we used 8 MPI processes $\times$ 2 OpenMP threads for the hybrid models. In both cases we run the ADI micro-kernel for various iteration spaces and variable tile heights in order to obtain the minimum overall completion time. Naturally, all experimental results depend largely on the micro-kernel benchmark used, its communication pattern, the shared-memory parallelism that can be achieved through OpenMP and the hardware characteristics of the platform (CPU, memory, network).

The experimental results are graphically displayed in Figure 2. Two conclusions that can easily be drawn are that the pure MPI model is almost in all cases the fastest, while on the other hand the coarse-grain hybrid model is always better than the fine-grain one. Nevertheless, in the 16 vs 8 $\times$ 2 series of experiments, the performance of the coarse-grain hybrid model is comparable to that of the pure MPI, and in fact delivers the best results in $16 \times 16 \times 524288$ iteration space.

In order to explain the differences in the overall performance of the 3 models, we conducted some more thorough profiling of the computation and communication times (Fig 1). The computation times clearly indicate that the pure MPI model achieves the most efficient parallelization of the computational part, while the fine-grain model is always worse than the coarse-grain one as regards the time spent on the computational part of the parallel code. The communication times follow a more irregular pattern, but on average they indicate a superior performance of the pure MPI model.

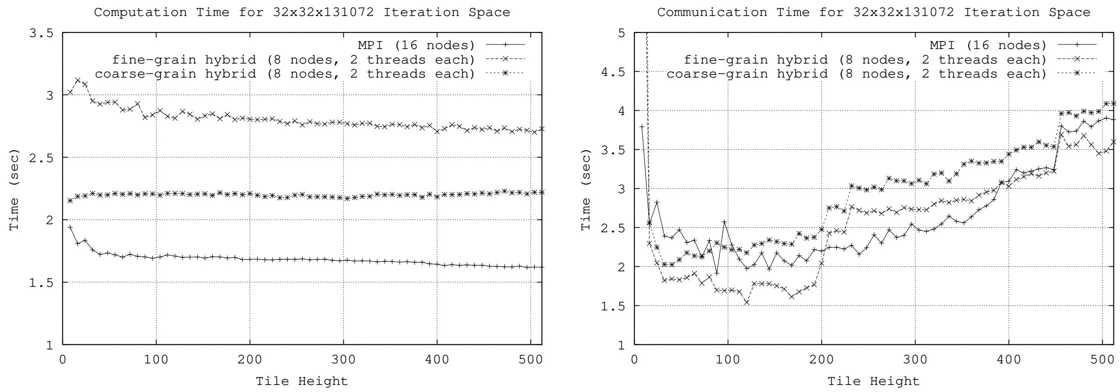

Fig. 1. Computation vs Communication Profiling for 32x32x131072 Iteration Space

The advantage of the coarse-grain model compared to the fine-grain one lies in that, according to the first model threads are only spawned once, while according to the second threads need to be re-spawned in each time step. This additional overhead accounts for the higher computation times of the fine-grain model that were experimentally verified. However, the coarse-grain model suffers from the serious disadvantage of the inefficient communication pattern, since the master thread in each node assumes more communication than an MPI node in the pure MPI model. This disadvantage could be diminished by either a more efficient load-balancing computation distribution scheme, or by a thread-safe MPI implementation that would allow all threads to call MPI routines.

6 Conclusions-Future work

In this paper we have presented three alternative parallel programming paradigms for nested loop algorithms and clusters of SMPs. We have implemented the three variations and tested their performance against the ADI Integration micro-kernel benchmark. It turns out that the performance of the hybrid coarse-grain OpenMP/MPI model looks quite promising for this class of algorithms, although the overall performance of each paradigm clearly depends on a number of factors, both application- and hardware-specific. We intend to investigate the behavior of the three paradigms more closely with a more extensive communication vs computation profiling, and apply a more efficient computation distribution, in order to mitigate the communication restrictions imposed by a non thread-safe MPI implementation.

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. 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. Em. 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*, Chicago, 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. George Em. Karniadakis and Robert M. Kirby. *Parallel Scientific Computing in C++ and MPI : A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge University Press, 2002.
8. 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.
9. B. V. Protopopov and A. Skjellum. A multi-threaded Message Passing Interface (MPI) architecture: performance and program issues. *JPDC*, 2001.
10. 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.
11. 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.

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

Table 1. Fine-grain hybrid parallelization

#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, $tile_{n-1} - 1$, $\boldsymbol{nod}$);
      */*send communication data*/*
      MPI_Isend(snd_buf, dest($\boldsymbol{nod}$));
      */*receive data for next tile*/*
      MPI_Irecv(recv_buf, src($\boldsymbol{nod}$));
    }
    */*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($\boldsymbol{tile}$);
#pragma omp master
    {
      */*wait for communication completion*/*
      MPI_Waitall;
      */*unpack communication data*/*
      Unpack(recv_buf, $tile_{n-1} + 1$, $\boldsymbol{nod}$);
    }
*/*synchronize threads for next time step*/*
#pragma omp barrier
  }
}

Table 2. Coarse-grain hybrid parallelization

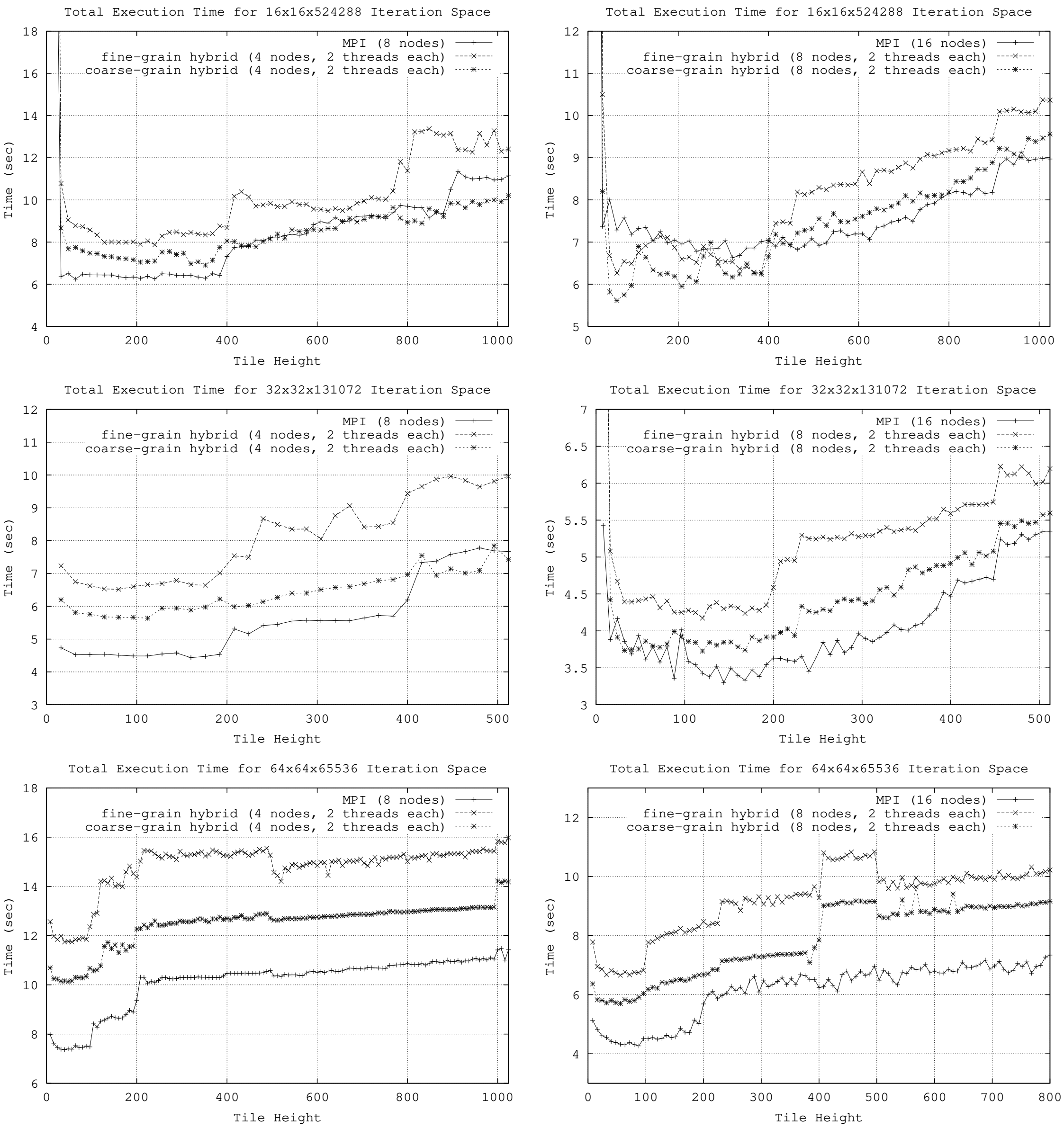

Fig. 2. Experimental Results for ADI Integration