The Effect of Process Topology and Load Balancing on Parallel Programming Models for SMP Clusters and Iterative Algorithms

Nikolaos Drosinos and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Athens, Greece, GR15780
{ndros, nkoziris}@cslab.ece.ntua.gr.

March 24, 2005

Abstract

This article focuses on the effect of both process topology and load balancing on various programming models for SMP clusters and iterative algorithms. More specifically, we consider nested loop algorithms with constant flow dependencies, that can be parallelized on SMP clusters with the aid of the tiling transformation. We investigate three parallel programming models, namely a popular message passing monolithic parallel implementation, as well as two hybrid ones, that employ both message passing and multi-threading. We conclude that the selection of an appropriate mapping topology for the mesh of processes has a significant effect on the overall performance, and provide an algorithm for the specification of such an efficient topology according to the iteration space and data dependencies of the algorithm. We also propose static load balancing techniques for the computation distribution between threads, that diminish the disadvantage of the master thread assuming all inter-process communication due to limitations often imposed by the message passing library. Both improvements are implemented as compile-time optimizations and are further experimentally evaluated. An overall comparison of the above parallel programming styles on SMP clusters based on micro-kernel experimental evaluation is further provided, as well.

1 Introduction

Distributed shared memory (DSM) architectures are becoming increasingly popular in high performance computing. The top ranking systems on the TOP500 list are essentially DSM platforms, proving that such systems can indeed sustain high performance, close to the theoretical peak figures. SMP clusters, being a typical representative of DSM platforms, are widely used in supercomputing centers, both for research and commercial purposes. SMP clusters encompass the scalability of monolithic clusters, as well as potential for shared memory multi-processing within each SMP node. For this type of architecture, there is an active research interest in considering alternative parallel programming models.

Traditionally, the *pure message passing model* has prevailed as the dominant parallelization technique for numerous high performance architectures, mainly due to its success in achieving both high performance and scalability in working environments. A significant amount of scientific algorithms have already been successfully parallelized with the aid of message passing libraries, most notably MPI, PVM etc., often delivering almost linear or even super-linear speedups. A large fraction of the code, that has been submitted to message passing parallelization, concerns nested loops, that traverse a specific iteration space performing computations. The message passing parallelization of such algorithms can be achieved with the aid of a coarse-grain task partitioning strategy, such as the *tiling* transformation, and has proved to be quite beneficial in many cases.

However, for SMP clusters, *hybrid* programming models are being considered, as well, although currently they are not as widespread as pure message passing ones. Generally, hybrid models retain message passing communication between different SMP nodes, while resorting to shared memory multi-threaded processing within each SMP node. Intuitively, hybrid models appear to map more closely to the architectural infrastructure of an SMP cluster than the monolithic paradigm, since they avoid using message passing communication within an SMP node, which substantially is a shared memory sub-system. Nevertheless, in practice, hybrid parallelization is a very open subject, as the efficient use of an SMP cluster calls for appropriate scheduling methods and load balancing techniques.

In this article we address two important issues regarding the parallelization of iterative algorithms onto SMP clusters, namely the specification of a suitable process topology, as well as the efficient thread load balancing. We assume tiled loop $N+1$-dimensional algorithms, which are parallelized across the outermost N dimensions and perform sequential execution along the innermost one in a pipeline fashion, interleaving computation and communication phases. We prove that the selection of an appropriate topology for the N-dimensional mesh of processes can significantly reduce the overall completion time, and provide a heuristic method for the determination of such a topology. Furthermore, we compare the *coarse-grain* hybrid model against the more popular *fine-grain* one, and emphasize on the need for an efficient load balancing strategy, in order to overcome the limited multi-threading support provided by existing standard message passing libraries, such as MPICH. Both improvements can be easily implemented as compiler optimizations or library routines, and are experimentally verified to deliver superior performance. Last, we compare all three models (pure message passing, fine-grain hybrid and coarse-grain hybrid) against micro-kernel benchmarks, in order to draw some more generic conclusions.

The rest of this article is organized as follows: Section 2 briefly discusses our target algorithmic model. Section 3 presents the various parallel programming models, while Sections 4 and 5 describe the proposed optimizations regarding process topology selection and thread load balancing, respectively. Section 6 displays results from the experimental evaluation of the above optimizations, whereas Section 7 refers to related scientific work. Finally, Section 8 summarizes the article and states some conclusions which can be drawn from the experimental results.

2 Algorithmic Model

Our algorithmic model concerns iterative algorithms formulated as perfectly nested loops with uniform data dependencies. Schematically, we consider tiled iterative algorithms of the form depicted in Alg. 1. Alg. 1 is the equivalent tiled code form of a $N+1$-

Algorithm 1: iterative algorithm model

```
foracross $tile_1 \leftarrow 1$ to H($X_1$) do
    ...
    foracross $tile_N \leftarrow 1$ to H($X_N$) do
        for $tile_{N+1} \leftarrow 1$ to H($Z$) do
            Receive($\overrightarrow{tile}$);
            Compute($A, \overrightarrow{tile}$);
            Send($\overrightarrow{tile}$);
        endfor
    endforacross
    ...
endforacross
```

dimensional iterative algorithm with an iteration space of $X_1 \times \cdots \times X_N \times Z$, which has been partitioned using a tiling transformation H. Z is considered the longest dimension of the iteration space, and should be brought to the innermost loop through permutation in order to simplify the generation of efficient parallel tiled code ([20]). In the above code, tiles are identified by an $N+1$-dimensional vector $(tile_1, \ldots, tile_{N+1})$. **foracross** implies parallel execution, as opposed to sequential execution (**for**). Generally, tiled code is associated with a particular tile-to-processor distribution strategy, that enforces explicit data distribution and implicit computation distribution, according to the computer-owns rule. For homogeneous platforms and fully permutable iterative algorithms, related scientific literature ([4], [1]) has proved the optimality of the columnwise allocation of tiles to processors, as long as sequential execution along the longest dimension is assumed. Therefore, all parallel algorithms considered in this article implement computation distribution across the N outermost dimensions, while each processor computes a sequence of tiles along the innermost $N+1$-th dimension.

The computational part involves calculations on one or more matrices (A in Alg. 1) and imposes uniform flow data dependencies, that result to a need for communication in order for each process to exchange boundary data with its neighboring processes. In most practical cases, the data dependencies of the algorithm are of several orders of magnitude smaller compared to the iteration space dimensions. Consequently, only neighboring processes need to communicate assuming reasonably coarse parallel granularities, taking into account that distributed memory architectures are addressed. According to the above, we only consider unitary process communication directions for our analysis, since all other non-unitary process dependencies can be satisfied according to indirect message passing techniques, such as the ones described in [19]. However, in order to preserve the communication pattern of the application, we consider a weight factor d_i for each process dependence direction i, implying that if iteration $\vec{j} = (j_1, \ldots, j_i, \ldots, j_{N+1})$ is assigned to a process $\vec{p}$, and iteration $\vec{j'} = (j_1, \ldots, j_i + d_i, \ldots, j_{N+1})$ is assigned to a different process $\vec{p'}$, $\vec{p} \neq \vec{p'}$, then data calculated at $\vec{j}$ from process $\vec{p}$ need to be sent to $\vec{p'}$, since they will be required for the computation of data at iteration $\vec{j'}$. Algorithms considered in this article impose data dependencies, that span the entire iteration space, and therefore essentially interleave computation and communication phases.

Summarizing, following restrictions are assumed for our algorithmic model:

- *fully permutable nested loops* This restriction constitutes the most important simplification adopted here.

- *unitary inter-process dependencies* This assumption holds for many real applications, whereas the non-unitary case can be similarly satisfied through indirect message passing techniques.

3 Parallel Programming Models for SMP Clusters

SMP clusters provide programming flexibility to the application developer, as they combine both shared and distributed memory architecture. One can think of SMP clusters as traditional monolithic MPP platforms, and transparently use a pure message passing programming style across the entire cluster, both for intra-node and inter-node communication. Alternatively, a parallel application could be designed in an hierarchical, two-level manner, so that it distinguishes between inter- and intra-node communication. In the first case, we refer to a *pure message passing* parallel programming paradigm, while the second approach is often addressed as *hybrid* parallelization. In this Section, we emphasize on the application of these two parallelization models on tiled iterative algorithms. Naturally, other parallelization approaches also exist, such as using a parallel programming language, like HPF or UPC, and relying on the compiler for efficiency, or even assuming a single shared memory system image across the entire SMP cluster, implemented with the aid of a Distributed Shared Virtual Memory (DSVM) software ([16], [11]). Nevertheless, these techniques are not as popular as either the message passing approach, or even hybrid parallel programming, hence they will not be considered in this article.

In the following, $P_1 \times \cdots \times P_N$ and $T_1 \times \cdots \times T_N$ denote the process and thread topology, respectively. $P = \prod_{i=1}^{N} P_i$ is the total number of processes, while $T = \prod_{i=1}^{N} T_i$ the total number of threads. Also, vector $\vec{p} = (p_1, \ldots, p_N)$, $0 \leq p_i \leq P_i - 1$ identifies a specific process, while $\vec{t} = (t_1, \ldots, t_N)$, $0 \leq t_i \leq T_i - 1$ refers to a particular thread. Throughout the text, we will use MPI and OpenMP notations in the proposed parallel algorithms.

3.1 Pure Message-passing Model

The proposed pure message passing parallelization for the algorithms described above is based on the tiling transformation. Tiling is a popular loop transformation and can be applied in the context of implementing coarser granularity in parallel programs, or even in order to exploit memory hierarchy by enforcing data locality. Tiling partitions the original iteration space of an algorithm into atomic units of execution, called tiles. Each process 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 [9].

The message passing parallelization paradigm for tiled nested loops is schematically depicted in Alg. 2. Each process is identified by N-dimensional vector $\vec{p}$, while different tiles correspond to different instances of $N+1$-dimensional vector $\overrightarrow{tile}$. The N outermost coordinates of a tile specify its owner process $\vec{p}$, while the innermost coordinate $tile_{N+1}$

Algorithm 2: pure message passing model

```
Data: Algorithm (Compute), space $\prod_{i=1}^{N} X_i Z$, process $\vec{p}$
for $i \leftarrow 1$ to $N$ do
    $tile_i = p_i$;
endfor
for $tile_{N+1} \leftarrow 1$ to $\lceil \frac{Z}{z} \rceil$ do
    foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ do
        Pack(snd_buf [$\overrightarrow{dir}$],$tile_{N+1} - 1$,$\vec{p}$);
        MPI_Isend(snd_buf [$\overrightarrow{dir}$],dest($\vec{p} + \overrightarrow{dir}$));
        MPI_Irecv(recv_buf [$\overrightarrow{dir}$],src($\vec{p} - \overrightarrow{dir}$));
    endforeach
    Compute($\overrightarrow{tile}$);
    MPI_Waitall ;
    foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ do
        Unpack(recv_buf [$\overrightarrow{dir}$],$tile_{N+1} + 1$,$\vec{p}$);
    endforeach
endfor
```

iterates over the set of tiles assigned to that process. z denotes the tile height along the sequential execution dimension, and determines the granularity of the achieved parallelism: higher values of z (in respect to Z) imply less frequent communication and coarser granularity, while lower values of z call for more frequent communication and lead to finer granularity. The investigation of the effect of granularity on the overall completion time of the algorithm and the selection of an appropriate tile height z are beyond the scope of this article. Generally, we consider z to be a user-defined parameter, and perform measurements for various granularities, in order to experimentally determine the value of z that delivers minimal execution time.

Furthermore, advanced pipelined scheduling is adopted as follows: In each time step, a process $\vec{p} = (p_1, \ldots, p_N)$ concurrently computes a tile $(p_1, \ldots, p_N, tile_{N+1})$, receives data required for the computation of the next tile $(p_1, \ldots, p_N, tile_{N+1} + 1)$ and sends data computed at the previous tile $(p_1, \ldots, p_N, tile_{N+1} - 1)$. $\mathbb{C}_{\vec{p}}$ denotes the set of valid communication directions of process $\vec{p}$, that is, if $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ for a non-boundary process identified by $\vec{p}$, then $\vec{p}$ needs to send data to process $\vec{p} + \overrightarrow{dir}$ and also receive data from process $\vec{p} - \overrightarrow{dir}$. $\mathbb{C}_{\vec{p}}$ is determined both by the data dependencies of the original algorithm, as well as by the selected process topology of the parallel implementation.

For the true overlapping of computation and communication, as theoretically implied by the above scheme by combining non-blocking message passing primitives with the overlapping scheduling, the usage of advanced CPU offloading features is required, such as zero-copy and DMA-driven communication. Unfortunately, experimental evaluation over a standard TCP/IP based interconnection network, such as Ethernet, combined with the ch_p4 ADI-2 device of the MPICH implementation, restricts such advanced non-blocking communication, but nevertheless the same limitations hold for our hybrid models, and are thus not likely to affect the relative performance comparison. However, this fact does complicate our theoretical analysis, since we will assume in general distinct, non-overlapped computation and communication phases, and thus to some extent underestimate the efficiency of the message passing communication primitives.

3.2 Hybrid Models

The potential for hybrid parallelization is mainly limited by the multi-threading support provided by the message passing library. From that perspective, there are mainly five levels of multi-threading support addressed in related scientific literature:

1. *single* No multi-threading support.

2. *masteronly* Message passing routines may be called, but only outside of multi-threaded parallel regions.

3. *funneled* Message passing routines may be called even within the dynamic extent of multi-threaded parallel regions, but only by the master thread. Other threads may run application code at this time.

4. *serialized* All threads are allowed to call message passing routines, but only one at a time.

5. *multiple* All threads are allowed to call message passing routines, without restrictions.

Each category is a superset of all previous ones. Currently, popular non-commercial message passing libraries provide support up to the third level (funneled), while only some proprietary libraries allow for full multi-threading support. Due to this fact, most attempts for hybrid parallelization of applications, that have been proposed or implemented, are mostly restricted to the first three thread support levels.

In this Subsection, we propose two hybrid implementations for iterative algorithms, namely both fine- and coarse-grain hybrid parallelization. Both models implement the advanced hyperplane scheduling presented in [2], that allows for minimal overall completion time.

3.2.1 Fine-grain Hybrid Model

The fine-grain hybrid programming paradigm, also referred to as masteronly in related literature, is the most popular hybrid programming approach, although it raises a number of performance deficiencies. The popularity of the fine-grain model over the coarse-grain one is mainly attributed to its programming simplicity: in most cases, it is a straightforward incremental parallelization of pure message-passing code by applying block distribution work sharing constructs to computationally intensive code parts (usually loops). Because of this fact, it does not require significant restructuring of the existing message passing code, and is relatively simple to implement by submitting the application to performance profiling and further parallelizing performance critical parts with the aid of a multi-threading API. Also, fine-grain parallelization is the only feasible hybrid approach for message passing libraries supporting only masteronly multi-threading.

However, the fine-grain model imposes significant restrictions in terms of achieving good performance. Most notably, its efficiency is directly associated with the fraction of the code that is incrementally parallelized, according to Amdahl's law. Since message passing communication can be applied only outside of parallel regions, other threads are essentially sleeping when such communication occurs, resulting to waste of CPU and poor overall load balancing. Also, this paradigm suffers from the overhead of re-initializing the thread structures every time a parallel region is encountered, since threads are continually spawned and terminated. This thread management overhead can be substantial, especially in the case of a poor implementation of the multi-threading library, and generally increases with the number of threads. Moreover, incremental loop parallelization is a very restrictive multi-threading parallelization approach for many real algorithms, where such loops either do not exist or cannot be directly enclosed by parallel regions. After all, the success of the message passing paradigm with all its programming complexity can be largely attributed to the programming potential of the SPMD model, for which a fine-grain multi-threading parallelization approach provides a poor substitute.

The proposed fine-grain hybrid implementation for iterative algorithms is depicted in Alg. 3. The hyperplane scheduling is implemented as follows: Each group of tiles is identified by a $N+1$-dimensional vector $\overrightarrow{group}$, where the N outermost coordinates denote the owner process $\vec{p}$, and the innermost one iterates over the distinct time steps. $\mathbb{G}_{\vec{p}}$ corresponds to the set of time steps of process $\vec{p}$, and depends both on the process and thread topology. For each instance of vector $\overrightarrow{group}$, each thread determines a candidate tile $\overrightarrow{tile}$ for execution, and further evaluates an **if**-clause to check whether that tile is valid and should be computed at the current time step. A barrier directive ensures that all threads are synchronized after the execution of the valid tiles, so that the next time step can begin. The hyperplane scheduling is more extensively presented in [2].

All message passing communication is performed outside of the parallel region (lines 5-9 and 20-23), while the multi-threading parallel computation occurs in lines 10-19. Note that no explicit barrier is required for thread synchronization, as this effect is implicitly achieved by exiting the multi-threading parallel region. Note also that only the code fraction in lines 10-19 fully exploits the underlying processing infrastructure, thus effectively limiting the parallel efficiency of the algorithm.

3.2.2 Coarse-grain Hybrid Model

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. Inter-node message passing communication occurs within the extent of the multi-threaded parallel region, but is completely assumed by the master thread, as dictated by the funneled thread support level. Intra-node synchronization between threads of the same SMP node is achieved with the aid of a barrier directive of the multi-threading API.

The coarse-grain model compensates the relatively higher programming complexity with potential for superior performance. The additional promising feature of this approach, as opposed to the fine-grain alternative, is the overlapping of multi-threaded computation with message passing communication. However, due to the restriction that only the master thread is allowed to perform message passing, a naive straightforward implementation of the coarse-grain model suffers from load imbalance between the

Algorithm 3: fine-grain hybrid model

```
Data: Algorithm (Compute), space $\prod_{i=1}^{N} X_i Z$, process $\vec{p}$, thread $\vec{t}$
for $i \leftarrow 1$ to $N$ do
    $group_i = p_i$;
endfor
foreach $group_{N+1} \in \mathbb{G}_{\vec{p}}$ do
    foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ do

        Pack(snd_buf [$\overrightarrow{dir}$],$group_{N+1} - 1$,$\vec{p}$);
        MPI_Isend(snd_buf [$\overrightarrow{dir}$],dest($\vec{p} + \overrightarrow{dir}$));
        MPI_Irecv(recv_buf [$\overrightarrow{dir}$],src($\vec{p} - \overrightarrow{dir}$));
    endforeach
    #pragma omp parallel
    begin
        for $i \leftarrow 1$ to $N$ do
            $tile_i = p_i T_i + t_i$;
        endfor
        $tile_{N+1} = group_{N+1}$ - $\sum_{i=1}^{N} tile_i$;
        if $1 \leq tile_{N+1} \leq \lceil \frac{Z}{z} \rceil$ then
            Compute($\overrightarrow{tile}$);
        endif
    end
    MPI_Waitall ;
    foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ do
        Unpack(recv_buf [$\overrightarrow{dir}$],$group_{N+1} + 1$,$\vec{p}$);
    endforeach
endforeach
```

threads, if equal portions of the computational load are assigned to all threads. Therefore, additional load balancing must be applied, so that the master thread will assume a relatively smaller computational load compared to the other threads, thus equalizing the per tile execution times of all threads. Moreover, the coarse-grain model avoids the overhead of re-initializing thread structures, since threads are spawned only once. Additionally, the coarse-grain hybrid approach can potentially implement more generic parallelization schemes, as opposed to its limiting fine-grain counterpart.

The pseudo-code for the coarse-grain parallelization of the fully permutable iterative algorithms is depicted in Alg. 4. Note that the inter-node communication (lines 11-17 and 24-29) is conducted by the master thread, per communication direction and per owner thread, incurring additional complexity compared to both the pure message passing and the fine-grain model. Also, note the `bal` parameter in the computation, that optionally implements load balancing between threads, as will be described in Section 5. Finally, note that the coarse-grain model requires explicit barrier synchronization, which is however expected to incur a smaller overhead compared to the re-initialization of the thread structures of the fine-grain model.

4 Process Topology

For all parallel programming models, the selection of an appropriate process topology is critical in order to achieve good performance. Usually, given P processes for the parallelization of an $N+1$-dimensional iterative algorithm according to the pipelined columnwise allocation, a N-dimensional mesh $P_1 \times \cdots \times P_N$ is selected, that constitutes a feasible solution to the following optimization problem:

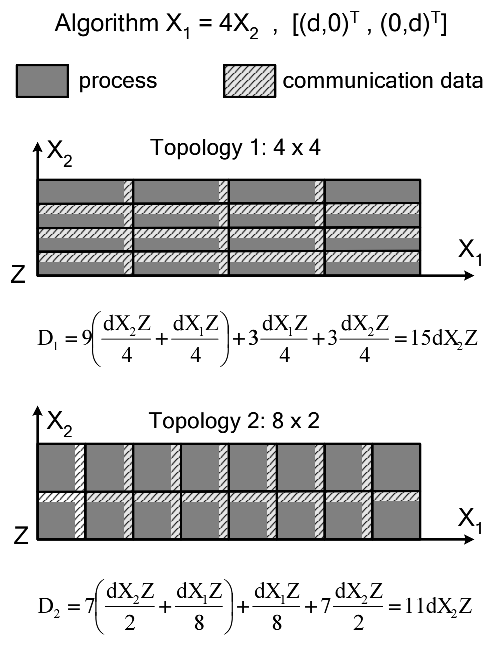
$$\left.\begin{aligned} P_i &\rightarrow \sqrt[N]{P} \\ P &= \prod_{i=1}^{N} P_i \\ P_i &\in \mathbb{N} \end{aligned}\right\}$$

The advantage of such a process topology is minimizing the latency of the parallel program, or equivalently ensuring that the most distant process will start executing its work share at the earliest possible time step. Although this is the most common approach for the selection of the process topology, is has a significant drawback for the type of iterative algorithms we study: it fails to adjust to either the iteration space or the data dependencies of the algorithm. However, by doing so, it is possible that it imposes higher communication needs compared to the ones that would be required, if a more appropriate topology was selected.

For instance, assuming that 16 processes are available for the parallel implementation of a 3D algorithm, the standard approach would be to consider a 4×4 topology, since

Algorithm 4: coarse-grain hybrid model

```
Data: Algorithm (Compute), space $\prod_{i=1}^{N} X_i Z$, process $\vec{p}$, thread $\vec{t}$
#pragma omp parallel
begin
    for $i \leftarrow 1$ to $N$ do
        $group_i = p_i$;
        $tile_i = p_i T_i + t_i$;
    endfor
    foreach $group_{N+1} \in \mathbb{G}_{\vec{p}}$ do
        $tile_{N+1} = group_{N+1}$ - $\sum_{i=1}^{N} tile_i$;
        #pragma omp master
        begin
            foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ do
                for $th \leftarrow 1$ to $M$ do
                    Pack(snd_buf [$\vec{dir}$],$group_{N+1} - 1$,$\vec{p}$,$th$);
                endfor
                MPI_Isend(snd_buf [$\overrightarrow{dir}$],dest($\vec{p} + \overrightarrow{dir}$));
                MPI_Irecv(recv_buf [$\overrightarrow{dir}$],src($\vec{p} - \overrightarrow{dir}$));
            endforeach
        end
        if $1 \leq tile_{N+1} \leq \lceil \frac{Z}{z} \rceil$ then
            Compute($\overrightarrow{tile}$,bal($\vec{p}$,$\vec{t}$));
        endif
        #pragma omp master
        begin
            MPI_Waitall ;
            foreach $\overrightarrow{dir} \in \mathbb{C}_{\vec{p}}$ do
                for $th \leftarrow 1$ to $M$ do
                    Unpack(recv_buf [$\overrightarrow{dir}$],$group_{N+1} + 1$,$\vec{p}$,$th$);
                endfor
            endforeach
        end
        #pragma omp barrier
    endforeach
end
```



$$D_1 = 9\left(\frac{dX_2Z}{4} + \frac{dX_1Z}{4}\right) + 3\frac{dX_1Z}{4} + 3\frac{dX_2Z}{4} = 15dX_2Z$$

$$D_2 = 7\left(\frac{dX_2Z}{2} + \frac{dX_1Z}{8}\right) + \frac{dX_1Z}{8} + 7\frac{dX_2Z}{2} = 11dX_2Z$$

Figure 1: Comparison of communication data for two alternative process topologies and 3D algorithm

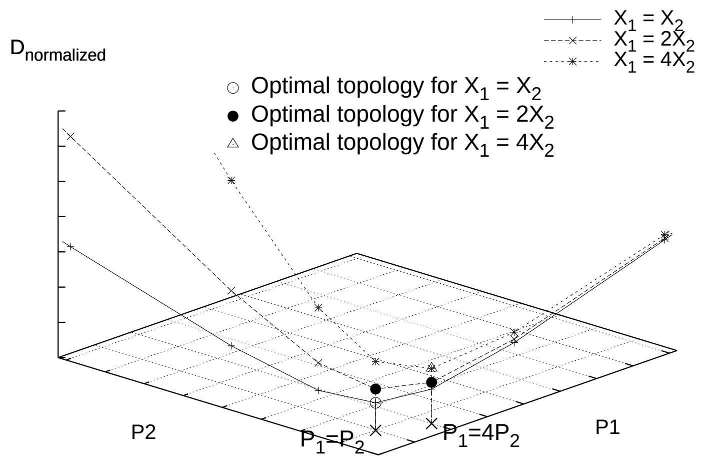


Figure 2: Selection of optimal process topology (P_1, P_2) for 3D algorithm according to normalized communication volume

it allows the most distant process to begin execution at the 7th time step. However, given an iteration space $X_1 \times X_2 \times Z$ with $X_1 = 4X_2$ and data dependencies $(d, 0)^T$ and $(0, d)^T$ (the $N+1$-th dimension is omitted in our dependence analysis, since it incurs no communication according to our scheduling scheme), a topology of 8×2 would be more appropriate for a distributed memory platform, as it reduces the communication volume about 27% (see Fig. 1). Note though that the latter topology does not allow the most distant process to begin execution until the 9th time step, therefore the parallel algorithm requires two additional execution steps under the 8×2 topology. Nevertheless, since each process will have to execute a sufficient number of tiles in order to ensure a satisfactory degree of pipelining, this overhead is expected to be negligible compared to the communication benefits attained at each execution step, if the 8×2 topology is selected.

Motivated by this observation, we developed a heuristic for the specification of an appropriate process topology, given an iterative algorithm with specific iteration space and data dependencies. The methodology is based on the following lemma:

Lemma 1. *Let $X_1 \times \cdots \times X_N \times Z$ be the iteration space of an $N+1$-dimensional iterative algorithm, that imposes data dependencies $[d_1, \ldots, 0]^T, \ldots, [0, \ldots, d_{N+1}]^T$. Let P be the number of processes available for the parallel execution of the algorithm. If there exist $P_i \in \mathbb{N}$, such that*

$$P = \prod_{i=1}^{N} P_i \tag{1}$$

and

$$\frac{d_i P_i}{X_i} = \frac{d_j P_j}{X_j}, 1 \leq i, j \leq N \tag{2}$$

then process topology $P_1 \times \cdots \times P_N$ minimizes inter-process communication for the tiled algorithm on P processes. Also, (2) is equivalent to

$$P_j = \frac{X_j}{d_j} \sqrt[N]{\frac{P \prod_{i=1}^{N} d_i}{\prod_{i=1}^{N} X_i}}, 1 \leq j \leq N \tag{3}$$

Note that (3) does not always define a valid topology. In fact, this is the case when there is at least one j with $1 \leq j \leq N$, such that $P_j \notin \mathbb{N}$. However, the monotonicity of function $\overline{D}$ (see Appendix A) ensures that if we can obtain a feasible integer solution in the neighborhood of the minimum of $\overline{D}$, as determined by (3), then the respective topology is likely to minimize the communication volume or at least provide an efficient topology for the specific algorithm. According to this observation, we have implemented

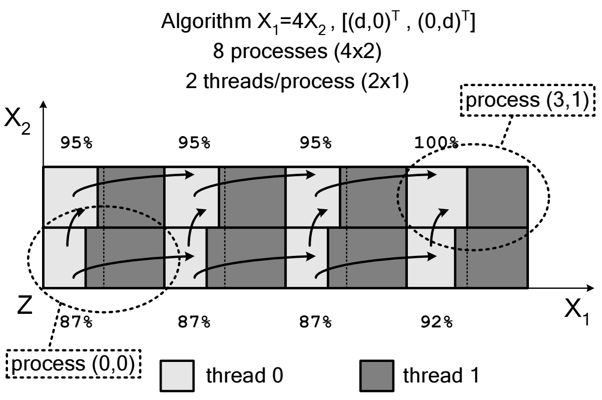


Figure 3: Variable balancing for 8 processes × 2 threads and 3D algorithm with $X_1 = 4X_2$

a heuristic function, that exhaustively searches for the optimal topology in the neighborhood of the minimum of $\overline{D}$. Since N is reasonably small for most practical cases, the complexity of the function is not prohibitive in practice.

Geometrically, the procedure is equivalent to determining an N-dimensional point $(P_1, \ldots, P_N)$, for which the communication volume `D` is minimized. Fig. 2 provides an illustrative example of the specification of a 2D process topology for a 3D algorithm with iteration space $X_1 \times X_2 \times Z$ and data dependencies $(d, 0)^T$, $(0, d)^T$. For instance, assuming 64K processes on a large scale system, a 3D iteration space with $X_1 = 4X_2$ would benefit most from a 512×128 topology. On the other hand, an iteration space with $X_1 = X_2$ would be more appropriately suited by a 256×256 topology, while the case $X_1 = 2X_2$ would call for either of these two topologies, from the communication point of view. In the latter case, the latency optimal $X_1 = X_2$ topology would be selected.

5 Load Balancing for Hybrid Model

The applied hyperplane scheduling enables for a more efficient load balancing between threads: Since the computations of each time step are essentially independent of the communication data exchanged at that step, they can be arbitrarily distributed among threads. Thus, under the funneled thread support level, it would be meaningful for the master thread to assume a smaller part of computational load, so that the total computation and the total communication associated with the owner process is evenly distributed among all threads.

We have implemented two alternative static load balancing schemes. The first one (*constant balancing*) requires the calculation of a constant balancing factor, which is common for all processes. For this purpose, we consider a non-boundary process, that performs communication across all N process topology dimensions, and determine the computational fraction of the master thread, that equalizes tile execution times on a per thread basis. The second scheme (*variable balancing*) requires further knowledge of the process topology, and calculates a different balancing factor for each process by ignoring communication directions cutting the iteration space boundaries, since these do not result to actual message passing. For both schemes, the balancing factor(s) can be obtained by the following lemma:

Lemma 2. *Let $X_1 \times \cdots \times X_N \times Z$ be the iteration space of an $N+1$-dimensional iterative algorithm, that imposes data dependencies $[d_1, \ldots, 0]^T$, …, $[0, \ldots, d_{N+1}]^T$. Let $P = P_1 \times \cdots \times P_N$ be the process topology and T the number of threads available for the parallel execution of the hybrid funneled implementation of the respective tiled algorithm. The overall completion time of the algorithm is minimal if the master thread assumes a portion $\frac{bal}{T}$ of the process's computational load, where*

$$bal = 1 - \frac{T-1}{t_{comp}\left\{\frac{Xz}{P}\right\}} \sum_{\substack{i=1 \\ i \in \mathbb{C}_{rank}}}^{N} t_{comm}\left\{\frac{d_i P_i X z}{X_i P}\right\} \tag{4}$$

$t_{comp}\{x\}$ *The computation time required for x iterations*

$t_{comm}\{x\}$ *The time required for the transmission of an x-sized message*

z *The tile height for each execution step of the tiled algorithm*

$\mathbb{C}_{rank}$ *Valid communication directions of process rank*

X *Equal to $\prod_{i=1}^{N} X_i$*

Note that if condition $i \in \mathbb{C}_{rank}$ is evaluated for each communication direction i, variable balancing is enforced. Otherwise, if the above check is omitted, (4) delivers the constant balancing factor.

The constant balancing scheme can be applied at compile-time, since it merely requires knowledge of the underlying computational and network infrastructure, but also tends to overestimate the communication load for boundary processes. On the other hand, the variable balancing scheme can be applied only after selecting the process topology, as it uses that information to calculate a different balancing factor for each process. Fig. 3 demonstrates the variable load balancing scheme, for a dual SMP cluster. A balancing factor of $X\%$ implies that the master thread will assume $\frac{X}{2}\%$ of the process's computational part, while the second thread will execute the remaining $\frac{200-X}{2}\%$.

Generally, a factor bal, $0 \leq bal \leq 1$, for load balancing T threads means that the master thread assumes $\frac{bal}{T}$ of the process's computational share, while all other threads are assigned a fraction of $\frac{T-bal}{T(T-1)}$ of that share. Note that according to the variable scheme, the balancing factor is slightly smaller for non-boundary processes. However, as the active communication directions decrease for boundary processes, the balancing factor increases, in order to preserve the desirable thread load balancing.

Clearly, the most important aspect for the effectiveness of both load balancing schemes is the accurate modeling of basic performance parameters concerning the system components, such as the sustained network bandwidth and latency, as well as and the requirements imposed by the specific algorithm in terms of processing power. In order to preserve the simplicity and applicability of the methodology, we avoid complicated in-depth software and hardware modeling, and adopt simple, yet partly crude, approximations for the system's behavior. Obviously, the more thoroughly the various system components are integrated into the theoretical calculation of t_{comp} and t_{comm} of (4), the more efficient the load balancing that is obtained.

6 Experimental Results

In order to test the topology and load balancing optimizations, we use both a micro-kernel benchmark, namely Alternating Direction Implicit integration (ADI), as well as a synthetic communication-intensive benchmark. ADI is a stencil computation used for solving partial differential equations ([12]). Essentially, ADI is a simple three-dimensional perfectly nested loop algorithm, that imposes unitary data dependencies across all three space directions. It has an iteration space of $X_1 \times X_2 \times Z$, where Z is considered to be the longest algorithm dimension. On the other hand, the synthetic benchmark uses the same algorithmic pattern as ADI, but imposes data dependencies $(3,0,0)^T$, $(0,3,0)^T$ and $(0,0,3)^T$, therefore requiring the exchange of notably more communication data compared to ADI.

We use MPI as the message passing library and OpenMP as the multi-threading API. Our experimental platform is an 8-node Pentium III dual-SMP cluster interconnected with 100 Mbps FastEthernet. Each node has two Pentium III CPUs at 800 MHz, 256 MB of RAM, 16 KB of L1 I Cache, 16 KB L1 D Cache, 256 KB of L2 cache, and runs Linux with 2.4.26 kernel. For the support of OpenMP directives, we use Intel C++ compiler v.8.1 with the following flags: `-O3 -mcpu=pentiumpro -openmp -static`. Finally, we use MPI implementation MPICH v.1.2.6, appropriately configured for an SMP cluster, so as to perform intra-node communication through SYS V shared memory. This version of the MPICH implementation asserts a funneled thread support level, and is thus capable of supporting all programming models discussed above. Some fine-tuning of the MPICH communication performance for our experimental platform indicated using a

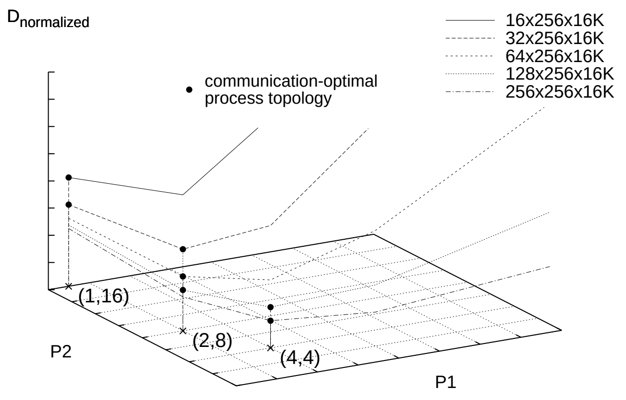


Figure 4: Selection of communication-optimal process topology (P_1, P_2) for five experimental iteration spaces

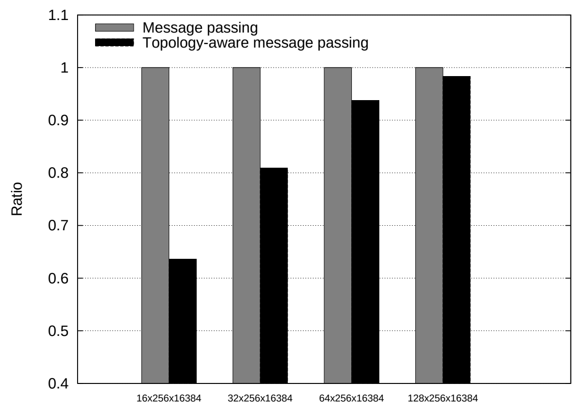
maximum socket buffer size of 104KB, so the respective `P4_SOCKBUFSIZE` environment variable was appropriately set to that value for all cluster nodes.

In all cases, we use 16 processes for the pure message passing experiments, and 8 processes with 2 threads per process for all hybrid programs. For the pure message passing case, an appropriate machine file is used to ensure that two MPI processes residing on the same SMP node will communicate through shared memory segments. Also, all experimental results are averaged over at least three independent executions for each case. Although effort was made, so as to take measurements in the absence of other users, our experimental platform is a non-dedicated cluster, and certain spikes in our curves could not be avoided, though most of them can be ascribed to the message passing communication performance.

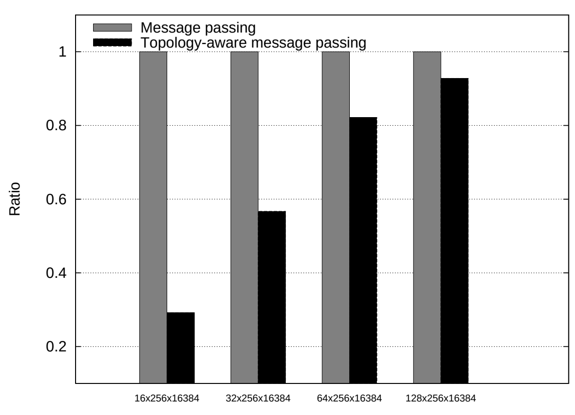
6.1 Effect of Process Topology

The proposed process topology optimization was tested against various iteration spaces and parallelization grains for both algorithms (ADI and the synthetic benchmark). More specifically, we performed measurements for four iteration spaces, namely $16 \times 256 \times 16K$, $32 \times 256 \times 16K$, $64 \times 256 \times 16K$ and $128 \times 256 \times 16K$. For the first two iteration spaces, the proposed communication-optimal topology is 1×16, while for the last two 2×8. On the other hand, in all cases the standard latency-minimal process topology is 4×4. The proposed topology for all iteration spaces and both algorithms is schematically depicted in Fig. 4.

Note that for the iteration spaces $16 \times 256 \times 16K$ and $64 \times 256 \times 16K$, the applied 1×16



(a) ADI integration



(b) synthetic benchmark

Figure 5: Comparison of simple and topology-aware MPI model (various iteration spaces, 8 dual SMP nodes, 16 processes)

and 2×8 topologies are communication optimal, and can be directly derived from (3). For the other two cases, (3) does not define a valid integer topology, so a valid one was determined according to our heuristic method by exhaustively examining feasible integer solutions in the neighborhood of the non-integer point defined by (3). Furthermore, we omitted the $256 \times 256 \times 16K$ iteration space, as in that case the communication optimal topology coincides with the standard 4×4 one. Also, we do not provide results for iteration spaces $X_1 \times X_2 \times Z$ with $X_1 > X_2$, since these results are very similar to their $X_2 \times X_1 \times Z$ counterparts. Figures 5(a) and 5(b) summarize the minimum total execution times, that were obtained for ADI and the synthetic benchmark, respectively. All execution times are normalized in respect to the times corresponding to the latency-minimal 4×4 topology.

The experimental results demonstrate an impressive performance gain, if an appropriate communication optimal process topology is selected. The improvement percentage increases, as we move away from relatively symmetric iteration spaces to more asymmetric ones. For example, ADI exhibits an improvement more than 35% for the $16 \times 256 \times 16K$ iteration space, while around 2% for the $128 \times 256 \times 16K$ space. This is due to the fact that the relative reduction of communication data in more asymmetric cases is more significant compared to more symmetric ones. Conclusively, the topology optimization is particularly important for algorithms with either asymmetric iteration spaces or asymmetric data dependencies.

The results for the synthetic benchmark are even more impressive, as was anticipated, since the communication volume in this case is significantly higher compared to ADI.

For instance, we observe an improvement about 70% for the $16 \times 256 \times 16K$ iteration space, which drops to about 6% for the $128 \times 256 \times 16K$ space. Consequently, given a specific hardware and network infrastructure, the proposed topology optimization should be applied to algorithms with intrinsic high communication to computation demands. Also, in both algorithms the improvement percentage is proportional to the relative reduction of the communication volume, while the cost of the additional execution steps under asymmetric process topologies is negligible.

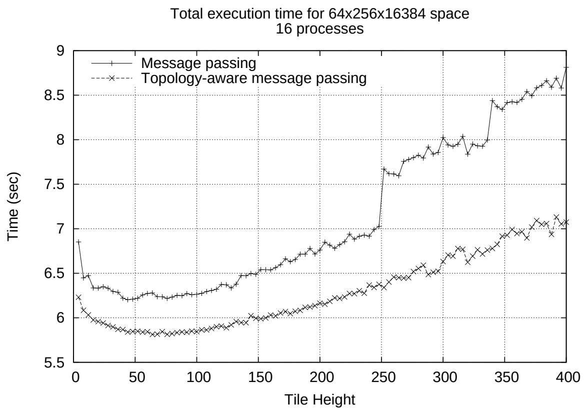
For each iteration space and parallelization grain, we also performed further profiling of the computation and communication times. In all cases, the results are quite similar, therefore we only portray partial profiling curves for one iteration space ($64 \times 256 \times 16K$). For various granularities, or equivalently for different tile heights z, we measured maximum total execution times (Fig. 6(a)), maximum computation times (Fig. 6(b)) and maximum communication times (Fig. 6(c)), by reducing the respective times of each process over all processes and acquiring the maximum for each case. By partial communication times we imply all processing times related to message passing communication, e.g. data transmission and reception MPICH non-blocking primitives, communication completion calls, as well as packing and unpacking communication data. Fig. 6(a) demonstrates that the topology-aware message passing implementation outperforms the standard one for the entire range of considered granularities. The similar pattern of the curves of Figures 6(a) and 6(c) imply that the improvements in the total execution time can indeed be ascribed to the reduction of the message passing communication time.

Finally, the fact that the overall performance deteriorates significantly at $z = 250$ for the standard 4×4 topology can be ascribed to the MPICH communication protocol transition: The MPICH implementation distinguishes between three communication protocols, namely short, eager and rendezvous, each of which implements a different message passing approach, thus requiring a different number of transmitted packets and possibly copies. For $z = 250$, the assumed mapping results to an amount of $1 \times 64 \times 250 \times 8 = 128000$ bytes of communication data along the X_1 dimension, which coincides with the threshold value for the transition from the eager to the rendezvous MPICH communication protocol. Generally, there is a trade-off between process synchronization and intermediate data buffering, therefore when transitioning from the eager to the rendezvous protocol a performance variation is not surprising.

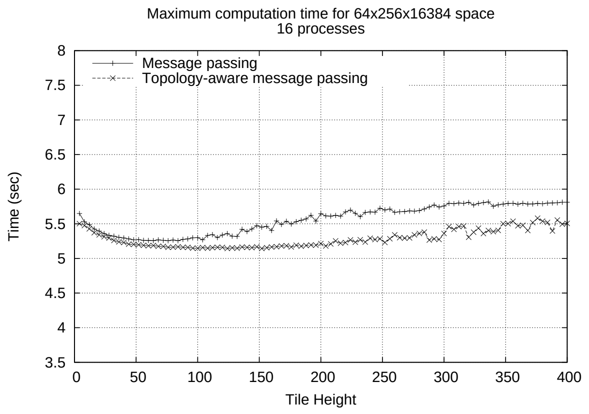
Summarizing, taking into account the iteration space and data dependencies of the algorithm when specifying the virtual process topology of its parallel tiled implementation is particularly important when

- the algorithm exhibits asymmetric data dependencies and/or iteration space dimensions

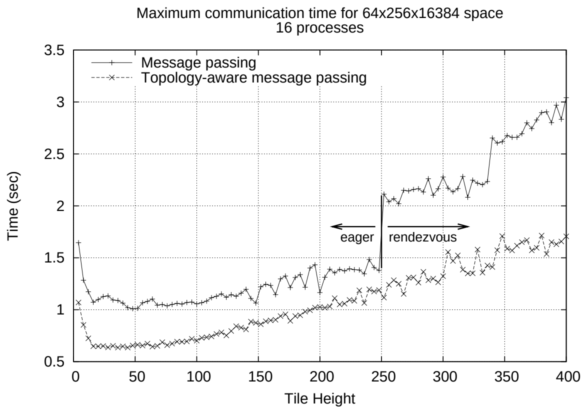
- the algorithm imposes relatively high communication-to-computation demands for



(a) total execution time



(b) maximum process computation time



(c) maximum process communication time

Figure 6: Comparison of simple and topology-aware MPI model (ADI integration, iteration space $64 \times 256 \times 16384$, 8 SMP nodes)

the underlying hardware/network parallel infrastructure and lower level message passing software

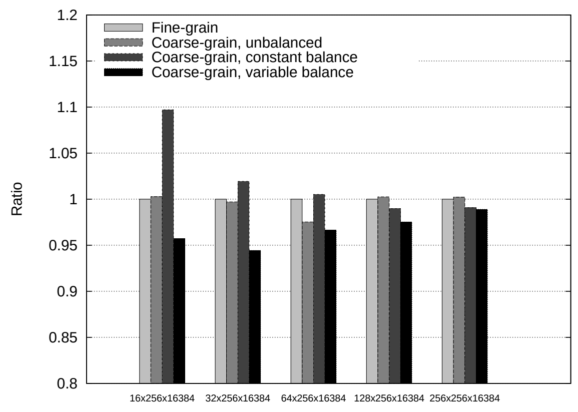
6.2 Effect of Load Balancing

We intend to experimentally verify the effectiveness of both balancing schemes, as well as their relative performance. A simplistic approach is adopted in order to model the behavior of the underlying infrastructure, so as to approximate quantities t_{comp} and t_{comm} of (4). As far as t_{comp} is concerned, we assume the computational cost involved with the calculation of x iterations to be x times the average cost required for a single iteration. On the other hand, the communication cost is considered to consist of a constant start-up latency term, as well as a term proportional to the message size, that depends upon the sustained network bandwidth on application level. Formally, we define

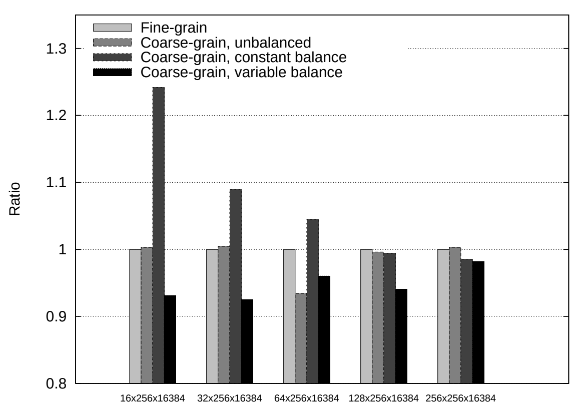
$$t_{comp}\{x\} = x t_{comp}\{1\} \tag{5}$$

$$t_{comm}\{x\} = t_{startup} + \frac{x}{B_{sustained}} \tag{6}$$

Since our primary objective was preserving simplicity and applicability in the modeling of environmental parameters, we intentionally overlooked at more complex phenomena, such as cache effects or precise communication modeling. Despite having thoroughly studied the MPICH source code in order to acquire an in-depth understanding of the ch_p4 ADI-2 device, we decided not to integrate implementation-specific protocol semantics into our theoretical model in order to preserve generality and simplicity. The same holds for cache effects, which would require a memory access pattern analysis of the tiled application in respect to the memory hierarchy configuration of the underlying architecture. Naturally, a more accurate representation of such hardware and software issues involved would probably lead to more efficient load balancing. Also, a major difficulty we encountered was modeling the TCP/IP socket communication performance and incorporating that analysis in our load balancing scheme. Assuming distinct, non-overlapping computation and communication phases and relatively high sustained network bandwidth allowed bypassing this restriction. However, this hypothesis underestimates the communication cost for short messages, which are mostly latency-bound and sustain relatively low throughput, while on the other hand it overestimates the respective cost in the case of longer messages, where DMA transfers alleviate the CPU significantly. Our main goal was providing some intuition as to the merit of these load balancing schemes, even under the most simple and straightforward implementation. For our analysis, we considered following values for the parameters of (5) and (6):



(a) ADI integration



(b) synthetic benchmark

Figure 7: Comparison of hybrid models (various iteration spaces, 8 dual SMP nodes)

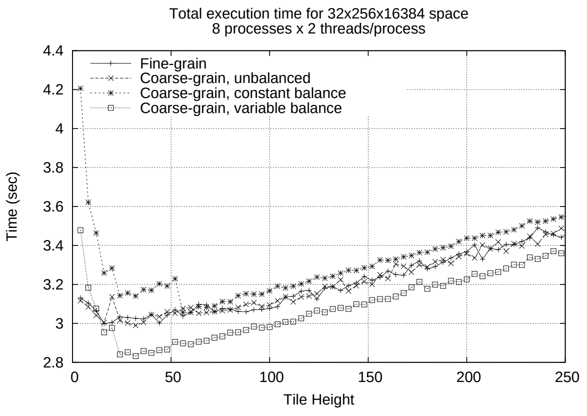
$$\begin{aligned} t_{comp}\{1\} &= 288nsec \\ t_{startup} &= 107usec \\ B_{sustained} &= 10MB/sec \end{aligned}$$

The overall experimental results for ADI integration and various iteration spaces are depicted in Fig. 7(a), while the respective results for the synthetic benchmark are summarized in Fig. 7(b). These results are normalized in respect to the fine-grain execution times. Granularity measurements for various iteration spaces and the ADI integration are depicted in Figures 8(a), 8(b), 8(c) and 8(d).

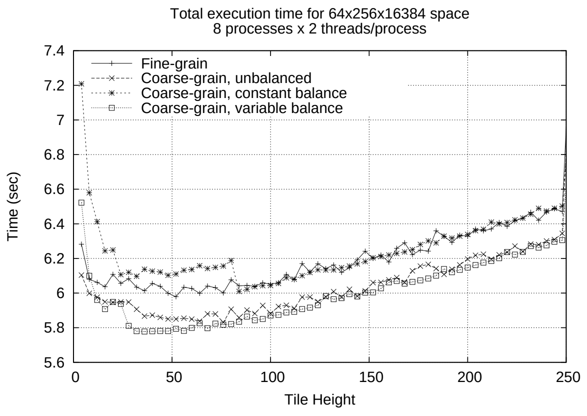
Although the experimental results are more complex compared to the pure message passing case, following conclusions can be drawn from the thorough investigation of the obtained performance measurements:

- The coarse-grain hybrid model is not always more efficient than its fine-grain counterpart. This observation reflects the fact that the poor load balancing of the simple coarse-grain model annuls its advantages compared to the fine-grain alternative (e.g. overlapping computation with communication, no thread re-initialization overhead etc.).

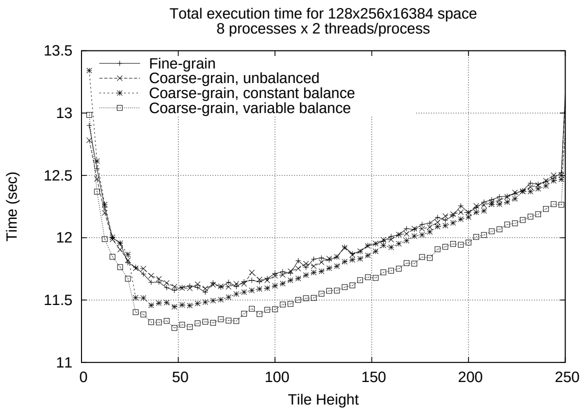
- When applying constant balancing, in some cases the balanced coarse-grain implementation is less effective than the unbalanced alternatives (either fine- or coarse-grain). This can be attributed both to inaccurate theoretical modeling of the



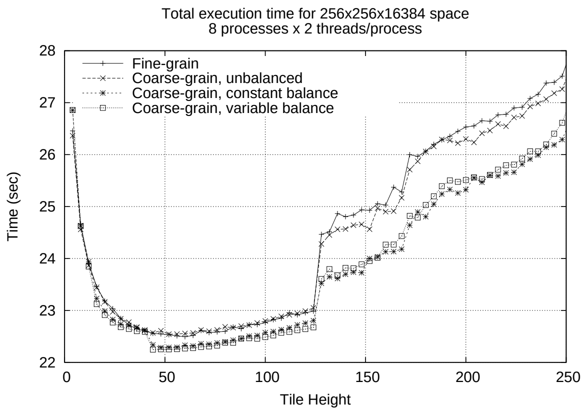
(a) $32 \times 256 \times 16K$



(b) $64 \times 256 \times 16K$



(c) $128 \times 256 \times 16K$



(d) $256 \times 256 \times 16K$

Figure 8: Comparison of hybrid models for ADI integration (8 dual SMP nodes)

system parameters for the calculation of the balancing factors, as well as to the inappropriateness of the constant balancing scheme for boundary processes. Interestingly, the constant load balancing approach falls short particularly in some cases for the synthetic benchmark. This can be ascribed to the fact that not distinguishing between boundary and non-boundary processes when load balancing has a more apparent effect at larger communication volumes.

- When applying variable balancing, the coarse-grain hybrid model was able to deliver superior performance compared to the fine-grain alternative in all cases. The performance improvement lies in the range of 2-8%. This observation complies with our theoretical intuition that, under appropriate load balancing, the coarse-grain model should always be able to perform better than the fine-grain equivalent.

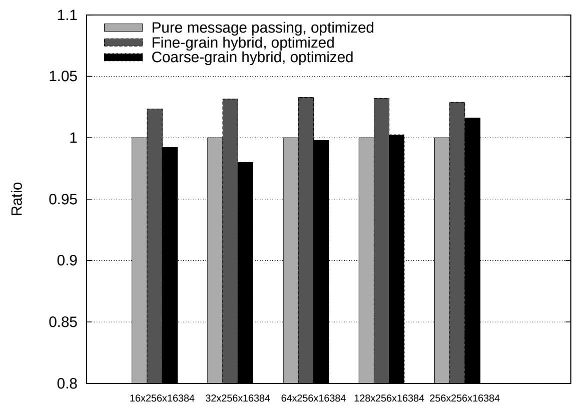
- Variable balancing appears to be in almost all iteration spaces the most efficient approach for the hybrid implementation. This observation also holds in terms of granularity, as Figures 8(a)-8(d) reveal that variable balancing performs better than any other implementation for almost all granularities. This remark intuitively implies that by more accurate inclusion of all related system parameters, variable balancing could in all cases deliver the best results.

6.3 Overall Comparison

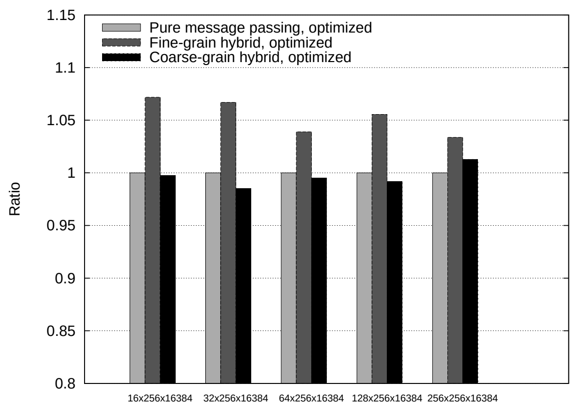
Finally, we also performed an overall comparison of the message passing model, the fine-grain hybrid one and the coarse-grain paradigm. All optimizations of the previous Sections were applied in this comparison, that is, we considered communication optimal process topologies and variable balancing for the coarse-grain model. Even though the fine-grain model has proved to be the most inefficient hybrid approach, we nonetheless included it in this comparison, mainly due to its popularity in related literature. We used both benchmarks for the comparison, and display the obtained results in Figures 9(a) and 9(b), normalized to the pure message passing execution times.

It should be noted that although this comparison may be useful towards the efficient usage of SMP clusters, it is largely dependent on the comparative performance of the two programming APIs (MPICH vs OpenMP support on Intel compiler). Also, it relies heavily on how efficiently multi-threading hybrid programming is supported by the MPICH library. Because of these reasons, the conclusions of the overall comparison cannot be generalized beyond the chosen hardware-software combination of our experimental testbed.

That said, we observe that fine-grain hybrid parallelization is always 3-8% worse in terms of performance compared to pure message passing. This observation holds for all iteration spaces, parallelization grains and both considered algorithms, and reflects



(a) ADI integration

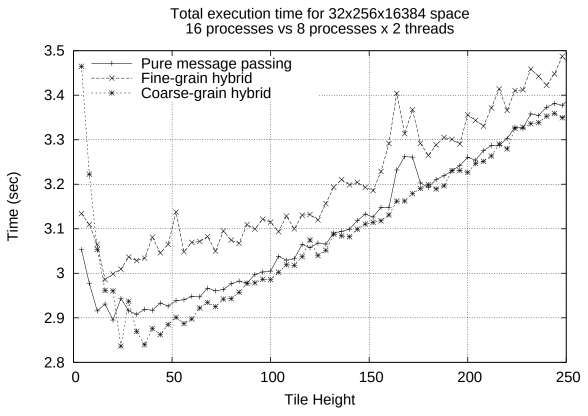


(b) synthetic benchmark

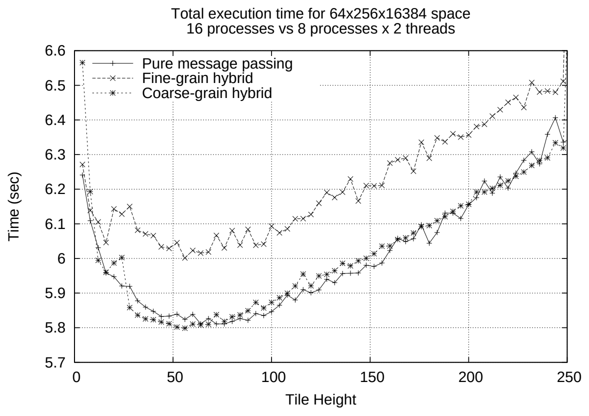
Figure 9: Overall comparison of optimized programming models (8 dual SMP nodes)

the fact that unoptimized hybrid programming suffers from serious disadvantages in respect to pure message passing, and fails to exploit its structural similarities with the architecture of SMP clusters. Limiting parallelization due to the masteronly thread support level according to Amdahl's law, as well as the overhead of re-initializing the thread structures by repeatedly entering and exiting parallel regions, are the main causes for the poor performance of the fine-grain hybrid model. The performance gap between the pure message passing and the fine-grain hybrid implementations widens in the case of the communication intensive synthetic benchmark, proving that, instead of relieving the communication part, fine-grain hybrid parallelization of iterative algorithms is even less beneficial than the pure message passing approach as the communication needs of the algorithm increase.

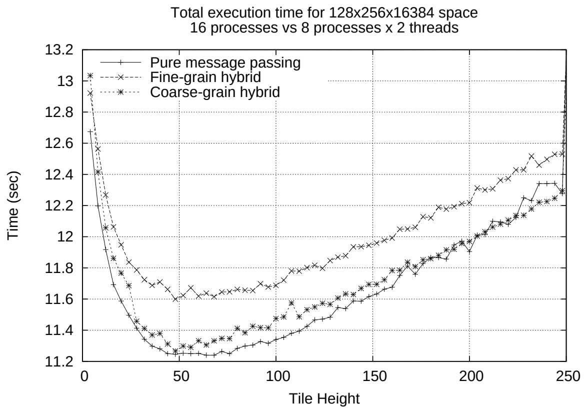
On the other hand, the combination of the coarse-grain model with an efficient load balancing technique seems very promising, as it significantly improves the obtained execution times. In fact, for most cases the optimized coarse-grain hybrid model outperforms the pure message passing one, although only by a small fraction. However, some drawbacks incurred by the coarse-grain model cannot be avoided, even under all proposed optimizations, the two most fundamental of which involve the additional communication overhead associated with having the master thread assume all message passing, as well as the difficulties in accurately estimating the impact of the various system parameters in order to apply load balancing. As a result, the message passing and the coarse-grain model perform quite similarly, and only at specific cases appear slight performance differences. Finally, we also provide granularity performance results for the ADI integration and various iteration spaces in Figures 10(a)-10(d).



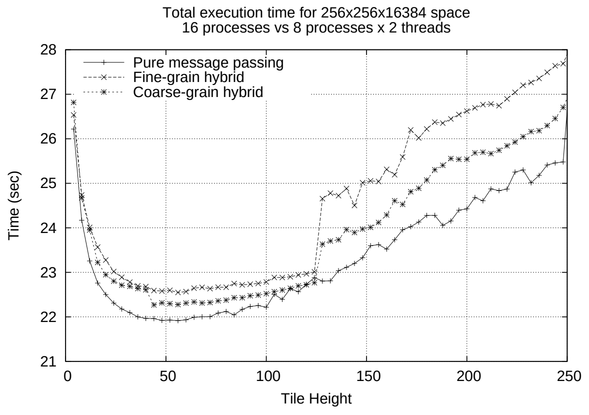
(a) $32 \times 256 \times 16K$



(b) $64 \times 256 \times 16K$



(c) $128 \times 256 \times 16K$



(d) $256 \times 256 \times 16K$

Figure 10: Overall comparison for ADI integration (8 dual SMP nodes)

7 Related Work

Hybrid programming models have arisen significant research interest. In [5] and [13], fine-grain hybrid MPI-OpenMP implementations perform worse than pure MPI for the popular NPB 2.3 benchmarks, mainly due to the relatively lower parallelization degree achieved with the hybrid model, and also because of different communication and memory access patterns. Similarly, the application of the hybrid model on specific scientific algorithms has delivered some controversial results (discrete element modeling [10], computational fluid dynamics [7], spectral climate modeling [15] etc). In [8], we have conducted a preliminary evaluation of hybrid programming models for nested loop algorithms, but without considering the important process topology and load balancing issues discussed here.

Load balancing techniques and theoretical inspection of the hybrid model in general have also been analyzed and discussed in literature. In [18], the authors distinguish between two main hybrid paradigms, namely masteronly (fine-grain) and overlapping (either funneled or multiple, coarse-grain). They investigate the deficiencies of the masteronly hybrid model (e.g. inter-node bandwidth saturation, CPU waste at MPI communication etc.), and claim that the coarse-grain hybrid model diminishes these effects with overlapping of computation and communication, as well as thread load balancing balancing techniques. From a different perspective, [14] addresses the problem of optimally mapping iterative algorithms interleaving computation and communication phases onto a heterogeneous cluster, taking into account network links contention and provides a heuristic method for efficient mapping. Load balancing is proposed here in order to mitigate varying processing power, although only task distribution to unidimensional processor arrays is considered. Nevertheless, no concrete load balancing techniques have been implemented and more importantly evaluated for the hybrid programming model.

Scheduling iterative algorithms on parallel platforms has been extensively studied, although no complete analysis for the specification of communication minimal process topology has been proposed to our knowledge. In [4] a cyclic columnwise allocation is considered the optimal approach among all possible distributions of tiles to equally powerful processors. In [3] the authors propose an efficient columnwise tile to processor distribution in heterogeneous environments. Based on the idea that block sizes should be proportional to the processor power, the authors develop a heuristic to determine efficient tile to processor distributions with bounded chunk size. The problem is to some extent equivalent to that of calculating an optimal process topology for specific iterative algorithms, for which we propose a heuristic method in this article. In [6], the authors consider a multipartioning parallelization strategy for line-sweep computations, and propose optimal domain decomposition by enforcing proper load balancing and minimizing the number of communication messages.

Summarizing, programming SMP clusters and hybrid parallel programming pose new challenges and also shed new light on older high performance related considerations. Although in theory hybrid programming models seem to be more appropriate for SMP clusters, experimental evaluation has eloquently demonstrated that achieving superior performance with hybrid programming is an intricate task, calling for efficient scheduling and load balancing parallelization techniques. Solid multi-threading support from the message passing library is also a very important issue; often, achieving high performance under hybrid programming is directly associated with overcoming the limitations imposed by the message passing library. Towards this direction, thread-safe extensions to popular existing message passing libraries, that would allow for portability, overlapping of computation and communication, and preserving latencies of individual networks, such as the approach presented in [17], could be helpful.

8 Conclusions

In this article we have discussed various programming alternatives for the parallelization of fully permutable iterative algorithms onto SMP clusters. We proposed a pure message passing implementation, as well as hybrid parallelization, and considered both major hybrid parallelization approaches, namely fine-grain and coarse-grain. We investigated the effect of the selected process topology in terms of overall completion time, proposed a heuristic method for the specification of a communication optimal process topology according to the iteration space and data dependencies of the algorithm and experimentally evaluated the performance gain attained when assuming the proposed optimization. Furthermore, experimental profiling of the hybrid model confirmed that a naive, simple fine-grain implementation suffers from major disadvantages compared to the pure message passing model, most notably the overhead of re-initializing parallel regions and also restricting parallelization according to Amdahl's law due to the masteronly thread support scheme. The poor load balancing of the funneled thread support scheme results to the unoptimized coarse-grain model also exhibiting comparatively low performance, although it does help in overcoming some of the limitations of the fine-grain alternative. We provide some guidelines for a variable load balancing scheme of the coarse-grain model, that delivers superior performance compared to the other hybrid programming alternatives. Last, we performed an overall comparison between the optimized versions of the three most popular implementations (message passing, fine-grain hybrid, coarse-grain hybrid), and concluded that despite the poor multi-threading support on behalf of the message passing library, when adopting all optimizations the coarse-grain model can be competitive to or even better than the pure message passing model.

A Proof of Communication-optimal Process Topology

Proof. According to (1), it holds

$$P_N = \frac{P}{P_1 \times \cdots \times P_{N-1}} \tag{7}$$

Each process assumes $\left\lceil \frac{X_i}{P_i} \right\rceil$ iterations in direction i under the proposed mapping, where $1 \leq i \leq N$. For the sake of simplicity, we assume that

$$\left\lceil \frac{X_i}{P_i} \right\rceil \simeq \frac{X_i}{P_i} \tag{8}$$

Due to the data dependences of the algorithm, each process is required to send $d_i \prod_{\substack{j=1 \\ j \neq i}}^{j=N} \frac{X_j}{P_j} Z$ data to direction i. Thus, the total communication volume D of a process can be obtained by the following expression:

$$\begin{aligned} D &= d_1 Z \prod_{\substack{i=1 \\ i \neq 1}}^{N} \frac{X_i}{P_i} + \cdots + d_N Z \prod_{\substack{i=1 \\ i \neq N}}^{N} \frac{X_i}{P_i} \\ &= \frac{d_1 Z P_1}{X_1} \prod_{i=1}^{N} \frac{X_i}{P_i} + \cdots + \frac{d_N Z P_N}{X_N} \prod_{i=1}^{N} \frac{X_i}{P_i} \\ &= \frac{Z \prod_{i=1}^{N} X_i}{P} \left(\frac{d_1 P_1}{X_1} + \cdots + \frac{d_N P_N}{X_N} \right) \end{aligned} \tag{9}$$

Using (7), (9) can be written by substituting P_N as follows:

$$\begin{aligned} D &= D(P_1, \ldots, P_{N-1}) \\ &= \frac{Z \prod_{i=1}^{N} X_i}{P} \sum_{i=1}^{N-1} \frac{d_i P_i}{X_i} + \frac{d_N Z \prod_{i=1}^{N} X_i}{X_N P_1 \ldots P_{N-1}} \end{aligned} \tag{10}$$

Note that D is substantially a function of $P_1, \ldots, P_{N-1}$ (formally: $D : \mathbb{N}^{N-1} \to \mathbb{R}$). Let $\overline{D}$ be the real extension of D, defined by (10) for $P_j \in \mathbb{R}, 1 \leq j \leq N$ ($\overline{D} : \mathbb{R}^{N-1} \to \mathbb{R}$).

For a stationary point $(P_1, \dots, P_{N-1})$ of $\overline{D}$ and $1 \le j \le N-1$ it holds:

$$\frac{\partial \overline{D}}{\partial P_j} = 0 \tag{11}$$

$$\frac{d_j Z \prod_{i=1}^{N} X_i}{P X_j} - \frac{d_N Z \prod_{i=1}^{N} X_i}{X_N P_1 \dots P_j^2 \dots P_{N-1}} = 0$$

$$\frac{d_j \prod_{i=1}^{N} X_i}{P X_j} = \frac{d_N P_N \prod_{i=1}^{N} X_i}{X_N P_j P}$$

$$\frac{d_j P_j}{X_j} = \frac{d_N P_N}{X_N} \tag{12}$$

Also,

$$\frac{\partial^2 \overline{D}}{\partial {P_j}^2} = \frac{2 d_N \prod_{i=1}^{N} X_i}{X_N P_1 \dots P_j^3 \dots P_{N-1}} > 0 \tag{13}$$

Because of (11) and (13), $\overline{D}$ has a minimum at $(P_1, \dots, P_{N-1})$, and as $P_i \in \mathbb{N}, 1 \le i \le N-1$, this will be the minimum of D, as well. Therefore, the communication data is minimal when a topology $P_1 \times \dots \times P_N$ satisfying (12) is assumed. Finally, it holds

$$\frac{P \prod_{i=1}^{N} d_i}{\prod_{i=1}^{N} X_i} = \frac{d_1 P_1}{X_1} \dots \frac{d_N P_N}{X_N} \tag{14}$$

and assuming D is minimal at $(P_1, \dots, P_N)$, (14) can be written because of (12) as follows:

$$\frac{P \prod_{i=1}^{N} d_i}{\prod_{i=1}^{N} X_i} = \left(\frac{d_j P_j}{X_j} \right)^N$$

$$P_j = \frac{X_j}{d_j} \sqrt[N]{\frac{P \prod_{i=1}^{N} d_i}{\prod_{i=1}^{N} X_i}} \tag{15}$$

$\square$

B Proof of Optimal Load Balancing for Hybrid Funneled Model

Proof. For the sake of simplicity, and without loss of generality, we assume that all divisions result to integers, in order to avoid over-complicating our equations with *ceil* and *floor* operators. Thus, each process will be assigned $\frac{Z}{z}$ tiles, each containing $\frac{Xz}{P}$ iterations. Furthermore, as the master thread assumes the execution of a fraction $\frac{bal}{T}$ of the process's computational load, each of the other $T-1$ threads will be assigned a fraction of $\frac{T-bal}{T(T-1)}$ of the above computations. Note that under the funneled hybrid model, only the master thread is allowed to perform inter-node communication, and should therefore take care of both its own communication data, as well as the communication data of the other threads. The overall completion time of the algorithm can be approximated by multiplying the total number of execution steps with the execution time required for each tile (step) T_{tile}. It holds

$$T_{tile} = max\{T_{master}, T_{other}\} \tag{16}$$

where

$$\begin{aligned} T_{master} = t_{comp}\left\{\frac{bal}{T}\frac{Xz}{P}\right\} + \\ \sum_{\substack{i=1 \\ i\in\mathbb{C}_{rank}}}^{N} t_{comm}\left\{\frac{d_i P_i X z}{X_i P}\right\} \end{aligned} \tag{17}$$

$$T_{other} = t_{comp}\left\{\frac{T-bal}{T(T-1)}\frac{Xz}{P}\right\} \tag{18}$$

In order to minimize the overall completion time, or equivalently the execution time for each tile (since the number of execution steps does not depend on the load distribution between threads), T_{master} must be equal to T_{other}. If this is not the case, that is if $T_{master} \neq T_{other}$, there can always be a more efficient load balancing by assigning more work to the more lightly burdened thread(s). Consequently, for minimal completion time under the assumed mapping, it holds

$$T_{master} = T_{other} \tag{19}$$

(17), (18) and (19) can be easily combined to deliver (4). □

References

[1] T. Andronikos, N. Koziris, G. Papakonstantinou, and P. Tsanakas. Optimal Scheduling for UET/UET-UCT Generalized N-Dimensional Grid Task Graphs. *Journal of Parallel and Distributed Computing*, 57(2):140–165, May 1999.

[2] 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.

[3] P. Boulet, J. Dongarra, Y. Robert, and F. Vivien. Static Tiling for Heterogeneous Computing Platforms. *Journal of Parallel Computing*, 25(5):547–568, May 1999.

[4] P. Calland, J. Dongarra, and Y. Robert. Tiling on Systems with Communication/Computation Overlap. *Journal of Concurrency: Practice and Experience*, 11(3):139–153, 1999.

[5] 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 (CDROM)*, page 12, Dallas, Texas, USA, 2000. IEEE Computer Society.

[6] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda. Generalized Multipartitioning of Multi-dimensional Arrays for Parallelizing Line-sweep Computations. *Journal of Parallel and Distributed Computing*, 63(9):887–911, 2003.

[7] S. Dong and G. Em. Karniadakis. Dual-level Parallelism for High-order CFD Methods. *Journal of Parallel Computing*, 30(1):1–20, 2004.

[8] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium 2004 (CDROM)*, page 10, Santa Fe, New Mexico, Apr 2004.

[9] 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.

[10] 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. IEEE Computer Society.

[11] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.

[12] G. Em. 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.

[13] G. Krawezik and F. Cappello. Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. *Journal of Concurrency and Computation: Practice and Experience*, 2003.

[14] A. Legrand, H. Renard, Y. Robert, and F. Vivien. Mapping and Load-balancing Iterative Computations on Heterogeneous Clusters with Shared Links. *IEEE Trans. on Parallel and Distributed Systems*, 15(6):546–558, Jun 2004.

[15] 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)*, page 18, Denver, Colorado, 2001. ACM Press.

[16] C. Morin and I. Puaut. A Survey of Recoverable Distributed Shared Virtual Memory Systems. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):959–969, 1997.

[17] B. V. Protopopov and A. Skjellum. A Multi-threaded Message Passing Interface (MPI) Architecture: Performance and Program Issues. *Journal of Parallel and Distributed Computing*, 61(4):449–466, Apr 2001.

[18] 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.

[19] P. Tang and J. Zigman. Reducing Data Communication Overhead for DOACROSS Loop Nests. In *Proceedings of the 8th International Conference on Supercomputing (ICS'94)*, pages 44–53, Manchester, UK, Jul 1994.

[20] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.