Efficient hybrid parallelization of tiled algorithms on SMP clusters

Nikolaos Drosinos and Nectarios Koziris
National Technical University of Athens,
School of Electrical and Computer Engineering,
Computing Systems Laboratory,
Iroon Polytexneioy 9, 15780 Zografou Campus, Athens, Greece
E-mail: {ndros,nkoziris}@cslab.ece.ntua.gr

Abstract: This article emphasizes on load balancing issues associated with hybrid programming models for the parallelization of tiled algorithms onto SMP clusters. Although tiled algorithms usually account for relatively regular computation and communication patterns, their hybrid parallelization often suffers from intrinsic load imbalance between threads. This observation mainly reflects the fact that most existing message passing libraries generally provide limited multi-threading support, thus allowing only the master thread to perform inter-node message passing communication. In order to mitigate this effect, we propose a generic method for the application of load balancing on the coarse-grain hybrid model for the appropriate distribution of the computational load to the working threads. We adopt both a static, as well as a dynamic load balancing approach, and implement three alternative balancing variations. All implementations are experimentally evaluated against kernel benchmarks, in order to demonstrate the potential of such load balancing schemes for the extraction of maximum performance out of hybrid parallel programs.

Keywords: nested loops; tiling transformation; load balancing; hybrid parallelization; SMP clusters; MPI; OpenMP

Reference to this article should be made as follows: Drosinos, N. and Koziris, N. (2006) 'Efficient hybrid parallelization of tiled algorithms on SMP clusters', *International Journal of Computational Science and Engineering*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Nikolaos Drosinos is currently a Ph.D. candidate in the School of Electrical and Computing Engineering at the National Technical University of Athens (NTUA). He received his Diploma in Electrical Engineering from the NTUA in 2001. His research interests include parallel processing (automatic parallelization, loop scheduling, hybrid parallel programming models, load balancing), parallel architectures and interconnection networks. He is a student member of the IEEE, and also a member of the IEEE Computer Society.
Nectarios Koziris received his Diploma in Electrical Engineering from the National Technical University of Athens (NTUA) and his Ph.D. in Computer Engineering from NTUA (1997). He is currently an Assistant Professor in the Computer Science Department, School of Electrical and Computer Engineering at NTUA. His research interests include parallel architectures, loop code optimizations, interaction between compilers, OS and architectures, communication architectures for clusters (OS and compiler support) and resource scheduling (CPU and storage) for large scale computer systems. He has published more than 60 research papers in international refereed journals and conferences/workshops, as well as two Greek textbooks "Mapping Algorithms into Parallel Processing Architectures", and "Computer Architecture and Operating Systems". He is a recipient of the IEEE IPDPS 2001 best paper award. He serves as a reviewer in International Journals and various HPC Conferences (IPDPS, ICPP etc). He served as a Program Committee member in HiPC-2002 & ICPP-2005 conferences, CAC03 & CAC04 workshops (organized with IPDPS), and Program Committee co-Chair for the ACM SAC03-PDS, SAC04-PDSN and SAC05-DSGC Tracks. He is a member of IEEE Computer Society, member of IEEE-CS TCPP and TCCA, ACM and chairs the Greek IEEE Computer Society Chapter. He also serves as a Board Member and Deputy Vice-Chair for the Greek Research and Education Network.

1 INTRODUCTION

SMP clusters have dominated the high performance computing domain by providing a reliable, cost-effective solution to both the research and the commercial communities. An immediate consequence stemming from the emersion of this new architecture is the consideration of new parallel programming models, which might exploit the underlying infrastructure more efficiently. Currently, message passing parallelization via the MPI library has become the de-facto programming approach for the development of portable code for a variety of high performance platforms. Although message passing parallel programs are generic enough, so as to be directly adapted to multi-layered, hierarchical architectures in a straightforward manner, there is an active research interest in considering alternative parallel programming models, that could be more appropriate for such platforms.

Hybrid programming models on SMP clusters resort to both message passing and shared memory access for inter- and intra-node communication, respectively, thus implementing a two-level hierarchical communication pattern. Usually, MPI is employed for the inter-node communication, while a multi-threading API, such as OpenMP, is used for the intra-node processing and synchronization. There are mainly two hybrid programming variations addressed in related work, namely the fine-grain incremental hybrid parallelization, as well as the coarse-grain SPMD-like alternative. Naturally, other cluster parallelization approaches also exist, such as using a parallel programming language, like HPF (Merlin and Hey (1995)) or UPC (El-Ghazawi et al. (2002)), and relying on the compiler for efficiency, or even visualizing a single shared memory system image across the entire SMP cluster, implemented with the aid of a Distributed Shared Virtual Memory (DSVM) software (Morin and Puaut (1997), Hu et al. (2000)). Nevertheless, these techniques are not as popular as either the message passing approach, or even hybrid parallel programming, hence they will not be discussed here.

Tiled loop algorithms account for a large fraction of the computational intensive part of many existing scientific codes. A typical representative of this application model stems from the discretization of Partial Differential Equations (PDEs). We consider generic $N+1$-dimensional tiled loops, which are parallelized across the outermost N dimensions, so as to perform sequential execution along the innermost dimension in a pipeline fashion, interleaving computation and communication phases. These algorithms impose significant communication demands, thus rendering communication-efficient parallelization schemes critical in order to obtain high performance. Moreover, the hybrid parallelization of such algorithms is a non-trivial issue, as there is a trade-off in programming complexity and parallel efficiency.

Hybrid parallelization is a popular topic in related literature, although it has admittedly delivered controversial results (Cappello and Etiemble (2000), Drosinos and Koziris (2004), Henty (2000), Dong and Karniadakis (2004), Loft et al. (2001), Ayguadé et al. (2004), Majumdar (2000), Nakajima (2003), Su et al. (2004) etc). In practice, it is still a very open subject, as the efficient use of an SMP cluster calls for appropriate scheduling methods and load balancing techniques. Most message passing libraries provide a limited multi-threading support level (e.g. funneled), allowing only the master thread to perform message passing communication. Even under multiple thread support, the additional complexity associated with ensuring thread safety can potentially diminish the load balancing advantages of full multi-threading support. Therefore, additional load balancing must be applied, so as to equalize the per tile execution times of all threads. This effect has been theoretically spotted in related literature (Rabenseifner and Wellein (2003), Legrand et al. (2004), Chavarría-Miranda and Mellor-Crummey (2003), Darte et al. (2003)), but to our knowledge no generic load balancing technique has been proposed and, more importantly, evaluated.

In this article we propose both static and dynamic load balancing for the coarse-grain funneled hybrid parallelization of tiled loop algorithms. The computational load associated with a tile is appropriately distributed among threads, either statically, based upon the estimation and modeling of basic system parameters, or at run-time, by sampling relative computation and communication times and dynamically applying appropriate load balancing. We distinguish between two variations of static load balancing, namely both a constant and a variable scheme, depending on whether the same task distribution is applied on a global or a per process base, respectively. We emphasize on the elements of applicability and simplicity, and evaluate the efficiency of the proposed scheme against two popular kernel benchmarks, namely ADI integration and a backward discretization of the Diffusion Equation. The experimental evaluation eloquently demonstrates the merit of load balancing when following the hybrid parallelization approach, and yields to significant overall performance improvement over the dominant message passing model.

The rest of this article is organized as follows: Section 2 discusses our target algorithmic model and introduces the notation used throughout the article. Section 3 refers to a pure message passing parallelization of tiled loop algorithms, while Section 4 presents the hybrid programming alternatives. Section 5 focuses on the proposed load balancing schemes, whereas Section 6 experimentally evaluates the efficiency of these schemes against ADI and DE kernels. Last, Section 7 concludes the paper and summarizes the performance results.

2 ALGORITHMIC MODEL - NOTATION

Our algorithmic model concerns tiled algorithms, that can be formally described as in Alg. 1. Such algorithms can be typically obtained by applying tiling transformation to fully permutable loops. Tiling is a popular loop transformation and can be applied in order to implement coarse granularity in parallel programs. Tiling partitions the original iteration space of an algorithm into atomic units of execution (tiles). This partitioning facilitates the parallelization of the algorithm, as a tile-to-process allocation scheme can be selected to implement domain decomposition in a straightforward manner. In our case, each process assumes the execution of a sequence of tiles, successive along the longest dimension of the original iteration space.

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`$(\overrightarrow{tile})$;
            `Send`$(\overrightarrow{tile})$;

Formally, we assume an $N+1$-dimensional algorithm with an iteration space of $X_1 \times \cdots \times X_N \times Z$, which has been partitioned using a tiling transformation defined by function `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 (Wolf and Lam (1991)). In the above code, tiles are identified by an $N+1$-dimensional vector $\overrightarrow{tile} = (tile_1, \ldots, tile_{N+1})$. **foracross** implies parallel execution, as opposed to sequential execution (**for**). In each tile denoted by a specific instance of vector $\overrightarrow{tile}$, a process first receives data required for the computations associated with the current tile (`Receive`), then performs these computations (`Compute`) and finally transmits computed data required by its neighboring processes (`Send`).

Generally, tiled code is associated with a particular tile-to-process 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 (Calland et al. (1997)) has proven the optimality of the columnwise allocation of tiles to processes, as long as sequential pipelined execution along the longest dimension is assumed. Therefore, all parallel algorithms considered in this paper implement computation distribution across the N outermost dimensions, while each process computes a sequence of tiles along the innermost $N+1$-th dimension.

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, which are common for the distributed memory architectures addressed here. 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 Tang and Zigman (1994). 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 iteration $\vec{j}$ from $\vec{p}$ need to be sent to $\vec{p'}$, since they will be required for the computation of data at iteration $\vec{j'}$.

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. Usually, if P_{mpi} the total number of processes for the message passing programming model, whereas P_{hybrid} the total number of processes and T_{hybrid} the respective number of threads for the hybrid model, both quantities P_{mpi} and $P_{hybrid} \times T_{hybrid}$ equal the total number of available processors, in order to fully exploit the computational infrastructure for the particular class of algorithms we address in this article. Nevertheless, for the sake of generality, we will only be considering processes and threads, as opposed to processors, where 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 MESSAGE PASSING PARALLELIZATION

The proposed pure message passing parallelization for the algorithms described above is based on the tiling transformation and 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 $tile_{N+1}$ 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 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 paper. 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. More on the effect of granularity on the parallel performance can be found in Kumar et al. (2003), Hodzic and Shang (1998), Andonov et al. (2003).

Furthermore, an advanced scheduling that allows for computation-communication overlapping 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 $(p_1, \ldots, p_N, tile_{N+1} - 1)$. $\mathbb{S}_{\vec{p}}$ denotes the set of valid data transmission directions of process $\vec{p}$, that is, if $\overrightarrow{dir} \in \mathbb{S}_{\vec{p}}$ for a non-boundary process $\vec{p}$, then $\vec{p}$ needs to send data to process $\vec{p} + \overrightarrow{dir}$. Similarly, $\mathbb{R}_{\vec{p}}$ corresponds to the valid data reception directions of process $\vec{p}$, implying that process $\vec{p}$ should receive data from process $\vec{p} - \overrightarrow{dir}$ if $\overrightarrow{dir} \in \mathbb{R}_{\vec{p}}$. $\mathbb{S}_{\vec{p}}$ and $\mathbb{R}_{\vec{p}}$ are 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 in the above scheme by combining non-blocking communication 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, prohibits such advanced non-blocking communication, but nevertheless the same limitations hold for our hybrid model, 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, an assumption that to some extent misrepresents the efficiency of the message passing communication primitives.

Algorithm 2: pure message passing model

/*determine tile sequence from pid $\vec{p}$ */
for $i \leftarrow 1$ **to** N **do**
    $tile_i = p_i$;
/*main loop: traverse all tiles */
for $tile_{N+1} \leftarrow 1$ **to** $\left\lceil \frac{Z}{z} \right\rceil$ **do**
    /*for each active transmission direction $\overrightarrow{dir}$... */
    foreach $\overrightarrow{dir} \in \mathbb{S}_{\vec{p}}$ **do**
        /*...pack previously computed data... */
        Pack($\overrightarrow{dir}$,$tile_{N+1} - 1$,$\vec{p}$);
        /*...and send to process $\vec{p} + \overrightarrow{dir}$ */
        MPI_Isend($\vec{p} + \overrightarrow{dir}$);
    /*for each active reception direction $\overrightarrow{dir}$... */
    foreach $\overrightarrow{dir} \in \mathbb{R}_{\vec{p}}$ **do**
        /*...receive from process $\vec{p} - \overrightarrow{dir}$ for next tile */
        MPI_Irecv($\vec{p} - \overrightarrow{dir}$);
    /*compute current tile */
    Compute($\overrightarrow{tile}$);
    /*complete pending communication */
    MPI_Waitall ;
    /*for each active reception direction $\overrightarrow{dir}$... */
    foreach $\overrightarrow{dir} \in \mathbb{R}_{\vec{p}}$ **do**
        /*...appropriately unpack received data */
        Unpack($\overrightarrow{dir}$,$tile_{N+1} + 1$,$\vec{p}$);

4 HYBRID PARALLELIZATION

The potential for hybrid parallelization is directly associated with the multi-threading support provided by the message passing library. From that perspective, there are mainly five levels of multi-threading support addressed in relevant 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 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 funneled or serialized thread support level, thus effectively restraining the message passing communication capabilities within multi-threaded regions, while only few proprietary libraries allow for full multi-threading support (multiple thread support level). 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.

Two major hybrid parallel programming variations are discussed in related literature, namely fine-grain and coarse-grain. The fine-grain model applies incremental multi-threading parallelization solely to specific computational code parts, and therefore requires minimum multi-threading support on behalf of the message passing library, as even the masteronly thread support level is sufficient. On the other hand, the coarse-grain hybrid programming model enforces an SPMD-like programming style by spawning threads only once close to the beginning of the program, thus calling for at least a funneled thread support level, as message passing will have to be conducted within multi-threaded parallel regions. The particular thread support level provided by the message passing library leads to two further flavors of the coarse-grain hybrid model, depending on whether only the master thread is allowed to perform message passing communication, or if all threads are enabled to call message passing primitives. The application of the three hybrid models (fine-grain, coarse-grain funneled and coarse-grain multiple) in the parallelization process of tiled loop algorithms will be the subject of this Section.

4.1 Fine-grain Hybrid Parallelization

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 an application to performance profiling and further parallelizing performance critical parts with the aid of multi-threading processing. Also, fine-grain parallelization is the only feasible hybrid approach for those message passing libraries, that support only masteronly multi-threading.

However, the efficiency of the fine-grain hybrid model 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 poor CPU utilization and inefficient 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. The thread management overhead can be substantial, especially in 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.

Algorithm 3: fine-grain hybrid model

/*determine group sequence from pid $\vec{p}$ */
for $i \leftarrow 1$ **to** N **do**
  $group_i = p_i$;
/*main loop: traverse all group instances */
foreach $group_{N+1} \in \mathbb{G}_{\vec{p}}$ **do**
  /*for each active transmission direction $\overrightarrow{dir}$... */
  foreach $\overrightarrow{dir} \in \mathbb{S}_{\vec{p}}$ **do**
    /*...pack data computed at previous group... */
    `Pack`($\overrightarrow{dir}$,$group_{N+1} - 1$,$\vec{p}$);
    /*...and send data to process $\vec{p} + \overrightarrow{dir}$ */
    `MPI_Isend`($\vec{p} + \overrightarrow{dir}$);
  /*for each active reception direction $\overrightarrow{dir}$... */
  foreach $\overrightarrow{dir} \in \mathbb{R}_{\vec{p}}$ **do**
    /*...receive from $\vec{p} - \overrightarrow{dir}$ for next group */
    `MPI_Irecv`($\vec{p} - \overrightarrow{dir}$);
  /*multi-threaded parallel construct */
  #pragma omp parallel
    /*calculate candidate tile to execute... */
    for $i \leftarrow 1$ **to** N **do**
      $tile_i = p_i T_i + t_i$;
    $tile_{N+1} = group_{N+1}$ - $\sum_{i=1}^{N} tile_i$;
    /*...and compute if valid tile */
    if $1 \leq tile_{N+1} \leq \lceil \frac{Z}{z} \rceil$ **then**
      `Compute`($\overrightarrow{tile}$);
  /*complete pending communication */
  `MPI_Waitall` ;
  /*for each active reception direction $\overrightarrow{dir}$... */
  foreach $\overrightarrow{dir} \in \mathbb{R}_{\vec{p}}$ **do**
    /*...appropriately unpack received data */
    `Unpack`($\overrightarrow{dir}$,$group_{N+1} + 1$,$\vec{p}$);

The proposed fine-grain hybrid implementation for iterative algorithms is depicted in Alg. 3. Hyperplane scheduling categorizes the tiles assigned to all threads of a specific process into *groups*, which can be concurrently executed. Each group contains all tiles, that can be safely executed in parallel by the specified number of threads T, without violating the data dependencies of the initial algorithm. Each group 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 execution time steps of process $\vec{p}$, and depends both on the process and thread topology. Formally, vector $\overrightarrow{group}$ and set $\mathbb{G}_{\vec{p}}$ are defined as follows:

$$\begin{aligned}
\overrightarrow{group} &= (group_1, \ldots, group_N, group_{N+1}) \\
group_i &= \begin{cases} p_i, & 1 \leq i \leq N \\ g \in \mathbb{G}_{\vec{p}}, & i = N+1 \end{cases} \\
\mathbb{G}_{\vec{p}} &= \Bigg\{ g \in \mathbb{N} | \sum_{i=1}^{N} p_i T_i + 1 \leq g \leq \sum_{i=1}^{N} p_i T_i + \\
&\quad + \sum_{i=1}^{N} \{T_i - 1\} + \left\lceil \frac{Z}{z} \right\rceil \Bigg\}
\end{aligned}$$

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.

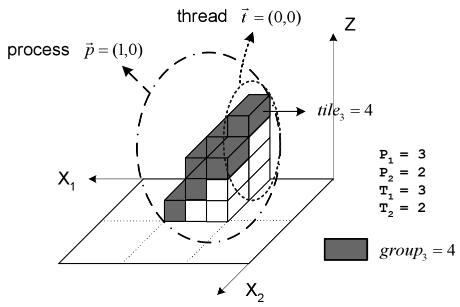


Figure 1: Hybrid parallel program for 3D algorithm and 6 processes × 6 threads

All message passing communication is performed outside of the parallel region (lines 4-8 and 15-17), while the multi-threading parallel computation occurs in lines 9-14. Note that no explicit barrier is required for thread synchronization, as this effect is implicitly achieved by exiting the multi-threaded parallel region. Note also that only the code fraction in lines 9-14 fully exploits the underlying processing infrastructure, thus effectively limiting the parallel efficiency of the algorithm. Fig. 1 clarifies some of the notation used in the hybrid algorithms.

4.2 Coarse-grain Funneled Hybrid Parallelization

According to the coarse-grain model, threads are only spawned once and their ids are used to determine their flow of execution in the SPMD-like code. Obviously, message passing communication will be performed within the multi-threaded parallel region, hence the coarse-grain programming model requires at least a funneled thread support level. Assuming funneled multi-threading support, the master thread will have to undertake the entire message passing communication required for the inter-node data transfer, though other threads will be allowed to perform computation at the same time. We shall briefly refer to this case as coarse-grain funneled model, or simply coarse-grain model in short.

The additional promising feature of the coarse-grain approach is the potential for overlapping 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 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, and 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 tiled algorithms is depicted in Alg. 4. Note that the inter-node communication (lines 8-13 and 17-20) is conducted by the master thread, per communication direction and per owner thread, incurring additional complexity compared both to the pure message passing and the fine-grain model. Also, note the `bal(`$\vec{p}$,$\vec{t}$`)` parameter in the computation, that corresponds to a balancing factor for thread $\vec{t}$ of process $\vec{p}$ and optionally implements load balancing between threads, as will be described in Section 5.

4.3 Coarse-grain Multiple Hybrid Parallelization

Should the message passing library provide full multi-threading support, another variation of the coarse-grain hybrid parallelization scheme is feasible. In fact, in that case of a thread-safe message passing library, relatively less programming effort is required compared to the funneled paradigm, as each thread can in theory satisfy its own communication needs. Moreover, the multiple thread support level allows for a more balanced distribution of the communication load to the available threads, as opposed to the funneled model, where the master thread undertakes the entire task of inter-process communication.

However, as message passing libraries consider only processes as peer communicating entities, additional care must be taken to implement point-to-point communication between threads residing on different processes with the aid of the message passing primitives. For instance, according to the MPI standard, there is no direct way for thread $\vec{t_i}$ residing on process $\vec{p}$ to address thread $\vec{t_j}$ of a different process $\vec{p'}$. Message passing communication can be established only between the two processes $\vec{p}$ and $\vec{p'}$, as opposed to a finer level between threads. As a workaround to this problem, it is possible to implicitly integrate the thread id information into the MPI message tag, so that a message coming from a remote thread is indirectly matched only by the appropriate local thread.

Alg. 5 outlines the coarse-grain multiple implementation. The algorithm is similar to Alg. 4, except that the **master** directives have been removed, since all threads contribute to the inter-process communication. $\mathbb{S}_{\vec{p},\vec{t}}$ cor-

Algorithm 4: coarse-grain funneled hybrid model

/*multi-threaded parallel construct */
#pragma omp parallel
  /*determine group and tile sequences */
  for $i \leftarrow 1$ **to** N **do**
    $group_i = p_i$;
    $tile_i = p_i T_i + t_i$;
  /*main loop: traverse all group instances */
  foreach $group_{N+1} \in \mathbb{G}_{\vec{p}}$ **do**
    /*calculate candidate tile to execute... */
    $tile_{N+1} = group_{N+1}$ - $\sum_{i=1}^{N} tile_i$;
    /*only master thread communicates */
    #pragma omp master
      /*for each active process transmission direction... */
      foreach $\overrightarrow{dir} \in \mathbb{S}_{\vec{p}}$ **do**
        /*...pack communication data for all threads... */
        for $th \leftarrow 1$ **to** T **do**
          `Pack(`$\vec{dir}$,$group_{N+1} - 1$,$\vec{p}$,th`);`
        /*...and send to neighbor process */
        `MPI_Isend(`$\vec{p} + \overrightarrow{dir}$`);`
      /*for each active process reception direction... */
      foreach $\overrightarrow{dir} \in \mathbb{R}_{\vec{p}}$ **do**
        /*...receive data from neighbor process */
        `MPI_Irecv(`$\vec{p} - \overrightarrow{dir}$`);`
    /*compute (balanced?) if valid tile */
    if $1 \leq tile_{N+1} \leq \left\lceil \frac{Z}{z} \right\rceil$ **then**
      `Compute(`$\overrightarrow{tile}$`,bal(`$\vec{p}$,$\vec{t}$`));`
    /*only master thread communicates */
    #pragma omp master
      /*complete pending communication */
      `MPI_Waitall ;`
      /*for each active process reception direction... */
      foreach $\overrightarrow{dir} \in \mathbb{R}_{\vec{p}}$ **do**
        /*...unpack communication data for all threads */
        for $th \leftarrow 1$ **to** T **do**
          `Unpack(`$\overrightarrow{dir}$,$group_{N+1} + 1$,$\vec{p}$,th`);`
    /*synchronize threads for next time step */
    #pragma omp barrier

Algorithm 5: coarse-grain multiple hybrid model

/*multi-threaded parallel construct */
#pragma omp parallel
  /*determine group and tile sequences */
  for $i \leftarrow 1$ **to** N **do**
    $group_i = p_i$;
    $tile_i = p_i T_i + t_i$;
  /*main loop: traverse all group instances */
  foreach $group_{N+1} \in \mathbb{G}_{\vec{p}}$ **do**
    /*calculate candidate tile to execute... */
    $tile_{N+1} = group_{N+1}$ - $\sum_{i=1}^{N} tile_i$;
    /*for each active thread transmission direction... */
    foreach $\overrightarrow{dir} \in \mathbb{S}_{\vec{p},\vec{t}}$ **do**
      /*...pack communication data... */
      `Pack(`$\vec{dir}$,$group_{N+1} - 1$,$\vec{p}$,$\vec{t}$`);`
      /*...and send to neighbor process */
      `MPI_Isend(`$\vec{p} + \overrightarrow{dir}$`,tag(`$\vec{t}$`));`
    /*for each active thread reception direction... */
    foreach $\overrightarrow{dir} \in \mathbb{R}_{\vec{p},\vec{t}}$ **do**
      /*...receive data from neighbor process */
      `MPI_Irecv(`$\vec{p} - \overrightarrow{dir}$`,tag(`$\vec{t}$`));`
    /*compute if valid tile */
    if $1 \leq tile_{N+1} \leq \left\lceil \frac{Z}{z} \right\rceil$ **then**
      `Compute(`$\overrightarrow{tile}$`);`
    /*complete pending communication */
    `MPI_Waitall ;`
    /*for each active thread reception direction... */
    foreach $\overrightarrow{dir} \in \mathbb{R}_{\vec{p},\vec{t}}$ **do**
      /*...unpack communication data */
      `Unpack(`$\overrightarrow{dir}$,$group_{N+1} + 1$,$\vec{p}$,$\vec{t}$`);`
    /*synchronize threads for next time step */
    #pragma omp barrier

responds to the set of valid transmission directions for thread $\vec{t}$ of the owner process $\vec{p}$. In particular, if $\overrightarrow{dir} \in \mathbb{S}_{\vec{p},\vec{t}}$, then data computed by thread $\vec{t}$ need to be sent to process $\vec{p} + \overrightarrow{dir}$, assuming $\vec{p}$ denotes a non-boundary process. Similarly, if $\overrightarrow{dir} \in \mathbb{R}_{\vec{p},\vec{t}}$, then process $\vec{p}$ needs to receive from its neighbor process $\vec{p} - \overrightarrow{dir}$, in order to satisfy dependencies related to data computed by thread $\vec{t}$ of $\vec{p}$. As all threads call message passing primitives based on the sets $\mathbb{S}_{\vec{p},\vec{t}}$, $\mathbb{R}_{\vec{p},\vec{t}}$, the parameter `tag()` emphasizes the role of the message tag in matching communicating thread pairs. Note also that we will not be considering load balancing for the coarse-grain multiple case, as this hybrid implementation primarily aims to be as simple as possible, by benefiting from the full multi-threading support provided by the message passing library. For the sake of simplicity, we shall refer to this programming model as multiple hybrid for the rest of this article.

5 LOAD BALANCING FOR THE HYBRID MODEL

Since most existing message passing libraries allow only the master thread to perform inter-process communication, the coarse-grain hybrid models suffer from intrinsic load imbalance and require an appropriate computation distribution scheme in order to achieve good performance. In the opposite case, the master thread will inevitably have to perform a larger fraction of work compared to the other threads, that is, if equal computational loads are assigned indiscriminately to all threads.

The hyperplane scheduling scheme enables 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, the former can be arbitrarily distributed among threads. Thus, 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.

In order to achieve this, we propose both a static and a dynamic (adaptive) approach for the calculation of the balancing factor(s). According to the static approach, load balancing is applied at compile time based upon a theoretical estimation of the relative communication vs computation cost of a specific algorithm on the particular underlying infrastructure. Only fundamental system characteristics (average iteration execution time, network bandwidth and latency) are included in this analysis, so as to keep the static approach applicable in practice. On the other hand, the dynamic alternative makes for a more obtrusive approach, where the relative communication vs computation cost of the algorithm is sampled at run-time, and no prior assumptions are made regarding the theoretical behaviour of the underlying system.

In this Section, we shall refer to the three proposed load balancing schemes, namely two variations of the static approach (constant and variable load balancing), as well as the dynamic, adaptive load balancing methodology.

5.1 Static Balancing

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, irrespective of the selected process topology. For this purpose, we consider a non-boundary process, that performs communication across all N process topology dimensions, and theoretically determine the computational fraction of the master thread, that equalizes tile execution times on a per thread basis. We then apply this constant balancing factor to all processes and quantitatively specify how much less computational load should be assigned to the master thread, so as to achieve the equalization of the total tile execution times of all threads.

The second scheme (*variable balancing*) requires further knowledge of the process topology, and ignores communication directions cutting the iteration space boundaries, since these do not result to actual message passing. According to the variable balancing scheme, we compute a different balancing factor for boundary and non-boundary processes, since the former are from the communication perspective less heavily burdened than the latter. Depending on the position of its owner process in the topology, each master thread is assigned a different balancing factor, as opposed to the constant balancing scheme, where the same balancing factor was applied to all master threads.

For both cases (constant and variable balancing), the balancing factor(s) can be obtained by 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 $P = P_1 \times \cdots \times P_N$ 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*

$$bal = 1 - \frac{T-1}{t_{comp}\left(\frac{Xz}{P}\right)} \sum_{\substack{i=1 \\ i \in \mathbb{S}_{\vec{p}}}}^{N} t_{comm}\left(\frac{d_i P_i X z}{X_i P}\right) \qquad (1)$$

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

$t_{comm}(x)$ *The transmission time of an x-sized message*

z *The tile height for each execution step*

$\mathbb{S}_{\vec{p}}$ *Valid data transmission directions of process $\vec{p}$*

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

The proof of the lemma is presented in the Appendix. We assume the computation time t_{comp} to be a linear function of the number of iterations, that is, we assume $t_{comp}(ax) = at_{comp}(x)$. Note that if condition $i \in \mathbb{S}_{\vec{p}}$ is evaluated independently for each process, variable balancing is enforced, as each communication term is only included in the sum of (1) as long as it contributes to message passing along the particular dimension for the specific process. If the above check is omitted, (1) delivers the constant balancing factor.

The constant balancing scheme only 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. 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, so as to preserve the desirable thread load balancing.

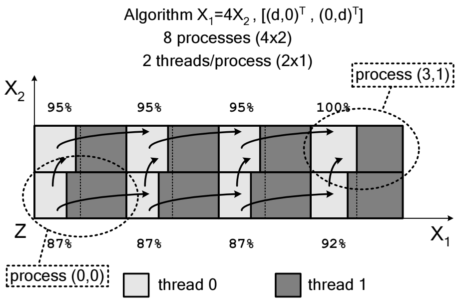


Figure 2: Variable balancing for 8 processes × 2 threads and 3D algorithm with iteration space $X_1 \times X_2 \times Z, X_1 = 4X_2$

Fig. 2 demonstrates the variable load balancing scheme for a 3D tiled algorithm, that has been mapped onto a 2D virtual process topology on a dual SMP cluster. The different balancing factors have been computed by applying (1), where functions $t_{comp}()$ and $t_{comm}()$ have been approximated as in equations (4) and (5) of Section 6, whereas $N = 2, P = 8, P_1 = 4, P_2 = 2, T = 2, X_1 = 4X_2, d_1 = d_2 = d$. Generally, a factor bal, $0 \le bal \le 1$, for load balancing T threads implies that the master thread assumes $\frac{bal}{T}$ of the process's computational share, while each other thread is assigned a fraction of $\frac{T-bal}{T(T-1)}$ of that share. Consequently, larger balancing factors tantamount to the master thread assuming larger fractions of the process's computational load, with a balancing factor of 100% meaning equal distribution of that load among all existing threads (no balancing). For instance, since process $(0,1)$ does not have to perform communication along dimension X_2, its master thread assumes a greater balancing factor (95% vs 87%) compared to the master thread of process $(0,0)$, which has to perform message passing communication along both X_1 and X_2 directions. Moreover, process $(3,1)$ will not be transmitting any communication data, hence no balancing is applied to its master thread.

Clearly, the most important aspect for the effectiveness of both static load balancing schemes is the accurate architectural modeling of both the underlying infrastructure, as well as its basic performance parameters, such as the sustained network bandwidth and latency, 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 structure and behavior. A dynamic scheme adopting an adaptive, run-time balancing approach, could potentially overcome some of the drawbacks associated with static balancing and its approximations, and is hence considered next.

5.2 Dynamic Balancing

Instead of applying thread load balancing a priori, according to a particular static modeling of application-system performance, it is possible to adaptively calculate appropriate balancing factors, by slightly restructuring the hybrid coarse-grain paradigm. Under a more obtrusive approach, each thread can time its computation and communication performance at run-time, so that load balancing is applied mainly based upon that information, and by further relying on a minimum set of assumptions. Obviously, as with most dynamic, obtrusive programming approaches, adaptive balancing incurs an additional profiling and processing overhead, mainly due to the insertion of timers, though only for a relatively short part of the code.

Suppose we apply an initial factor bal for the balancing of the T threads of a specific process. bal can be computed by using either of the static load balancing techniques mentioned above. Assuming P processes, our intention is to sample program execution for more than PT time steps, so that the execution wavefront reaches the most distant process, as we would like to time communication performance in a full parallel pipeline state. Based on the information gathered at the sampling period, a more appropriate load balancing factor bal' can be applied for the rest of the algorithm, according to the following lemma:

Lemma 2. *Assuming a balancing factor bal and T threads, let t^m_{comp}, t^m_{comm} be the average tile computation and communication times of the master thread, profiled at run-time. A more efficient balancing factor bal' can be computed using the following expression:*

$$bal' = 1 - bal\frac{T-1}{T}\frac{t^m_{comm}}{t^m_{comp}} \tag{2}$$

The proof of Lemma 2 is presented in the Appendix. It should be noted that t^m_{comm} refers to the non-overlapped communication time of the master thread, that is, excluding any DMA-driven communication overlapped with computation, hence it can easily be profiled. In practice, the coarse-grain model depicted in Alg. 4 is unfolded into two parts, the sampling period and the main execution period. During the sampling period, the initial balancing factor bal

is applied, and the master thread profiles its partial tile computation and communication times (t^m_{comp} and t^m_{comm}, respectively), that approximately constitute the total tile execution time. Based on this measurements, the new balancing factor bal' is computed, and further applied for the main execution period of the program.

The adaptive balancing approach is expected to incur some additional performance penalty, mainly due to the timing requirements and re-balancing process. More importantly, it adopts two basic assumptions: first, that the communication time is invariant to the computation distribution between threads, and second that the computation time is proportional to the number of iterations. The first assumption misrepresents the potential of overlapping computation and communication, while the second fails to consider cache effects. However, these simplifications ensure the generality and applicability of the proposed dynamic balancing scheme. In fact, the dynamic scheme requires even less input to be supplied by the user concerning application and platform characteristics, as it basically extracts that information by monitoring and profiling program execution performance at run-time.

6 EXPERIMENTAL RESULTS

In order to test the efficiency of the proposed load balancing schemes, we compared the performance of all proposed programming models (message passing, fine-grain hybrid, coarse-grain hybrid) against two kernel benchmarks, namely Alternating Direction Implicit integration (ADI), as well as a second order backward discretization of the Diffusion Equation (DE). ADI is a stencil computation used for solving partial differential equations (Karniadakis and Kirby (2002)). Essentially, ADI is a simple three-dimensional perfectly nested loop algorithm, that imposes unitary data dependencies across all three space directions. On the other hand, DE is a similar three-dimensional algorithm, that can be derived from the following unsteady diffusion PDE with the aid of backward discretization:

$$\begin{aligned} \frac{\partial \Theta}{\partial t} &= \nabla^2 \Theta \\ \Theta(x, y, 0) &= f(x, y) \\ \Theta(x, y, t) &= g(x, y, t) \text{ on } \partial\Omega \end{aligned} \tag{3}$$

Both kernels have an iteration space of $X_1 \times X_2 \times Z$, where Z is considered to be the longest algorithm dimension. Furthermore, both applications are suitable for the performance evaluation of all parallel programming models and techniques discussed here, as they are typical representatives of nested loop algorithms and comply with our target algorithmic model. More importantly, they impose communication in all three iteration space dimensions, and thus facilitate the comparison of the various programming models in terms of communication performance. In fact, in the DE kernel, a process transmits three times the communication volume along each iteration space dimension compared to ADI, therefore the relative performance of the two kernels enables scalability evaluation of the different programming models in respect to the inherent communication needs of a specific application.

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 optimization flags: `-O3 -mcpu=pentiumpro -openmp -static`. We also used two MPI implementations, namely the popular open-source MPICH library (v.1.2.6), as well as the proprietary ChaMPIon/Pro MPI library (v.1.1.1). Both MPI libraries have been appropriately configured, in order to perform SMP intra-node communication efficiently (e.g. through SYS V shared memory). MPICH provides at most a funneled thread support level, while ChaMPIon/Pro is fully thread-safe, and will be mainly used for the performance comparison of the multiple hybrid implementation with the other hybrid alternatives. Some fine-tuning of the MPICH communication performance for our experimental platform indicated using a maximum socket buffer size of 104KB, so the respective environment variable was appropriately set to that value for all cluster nodes.

Under the MPICH library, we are able to fully exploit our cluster architecture, as we will be using 16 processes for the pure message passing experiments, and 8 processes with 2 threads per process for all hybrid programs. With the ChaMPIon/Pro library, the number of available licenses compelled us to use a maximum of 8 processes for the message passing case, and only 4 processes for the hybrid implementations, with each process spawning 2 threads. Furthermore, all experimental results are averaged over three independent executions for each case.

Regarding static thread load balancing, 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 (1). 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

$$t_{comp}(x) = x t_{comp}(1) \tag{4}$$

$$t_{comm}(x) = t_{startup} + \frac{x}{B_{sustained}} \tag{5}$$

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. For instance, we ignored cache effects in the estimation of the computation cost, as we intended to avoid performing a memory access pattern analysis of the tiled application in respect to the memory hierarchy configuration of the underlying architecture. Also, a major difficulty we encountered was modeling the TCP/IP communication performance and incorporating that analysis in our static load balancing scheme. Assuming distinct, non-overlapping computation and communication phases and relatively high sustained network bandwidth allowed us to bypass this restriction. However, this hypothesis underestimates the communication cost for short messages, since these are mostly latency-bound and sustain relatively low throughput. On the other hand, it overestimates the respective cost in the case of longer messages, where DMA transfers alleviate the CPU significantly. For our analysis, we considered $t_{comp}(1) = 288nsec$, $t_{startup} = 107usec$ and $B_{sustained} = 100Mbps$.

Finally, as regards adaptive balancing, we considered a duration of $2PT$ time steps for the sampling period. For instance, given $P = 8$ and $T = 2$, this corresponds to 32 time steps, while each process will be executing 82 to 8k tiles, depending on the tile height (z ranges from 2 to 200, while dimension Z is always equal to 16k, for all iteration spaces). Note that for some cases, the duration of the sampling period is not negligible, compared to the main execution period.

6.1 ADI Integration

Initially, we performed an overall performance comparison of all proposed programming models against the ADI kernel, by using both the MPICH and the ChaMPIon/Pro MPI implementations. It should be noted that although this comparison may be useful towards the efficient usage of SMP clusters, it cannot be generalized beyond the chosen hardware-software combination, as it largely depends upon the comparative performance of the specific programming APIs (MPICH, ChaMPIon/Pro, OpenMP support on Intel compiler), as well as how efficiently the MPI library supports multi-threaded programming.

We tried five different iteration spaces for ADI, in order to investigate the performance variation of each model for different process topologies. For each iteration space and programming model, we selected among all feasible cartesian process topologies the one that minimizes the total execution time. The selected topologies for each case are depicted in Table 1. The variety of iteration spaces and the potential for multiple alternative process topologies eloquently demonstrates that the comparison of the pure message passing model with the hybrid approach is a non-trivial issue, even from the communication perspective. Generally, the total communication volume is reduced when assuming the hybrid approach, however this reduction is not necessarily proportional to the number of threads T, but also depends on the particular process topology. For instance, considering the iteration space $32 \times 256 \times 16k$, we expect the process topology assumed with the hybrid implementations to be quite beneficial, as it eliminates the need for communication across the long $X_2 = 256$ iteration space dimension. On the other hand, for the iteration space $256 \times 256 \times 16k$, the communication data is reduced by approximately 33%, while there are only half as many entities to perform message passing under funneled multi-threading support (16 processes in the pure message passing model vs 8 master threads in the hybrid alternatives). Thus, even theoretically, the advantages of the hybrid approach may be diminished without proper thread load balancing.

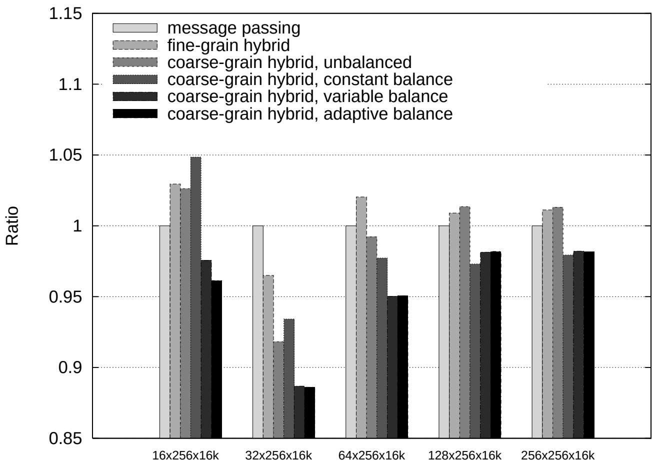


Figure 3: Comparison of hybrid models (ADI integration, 8 dual SMP nodes, MPICH, various iteration spaces)

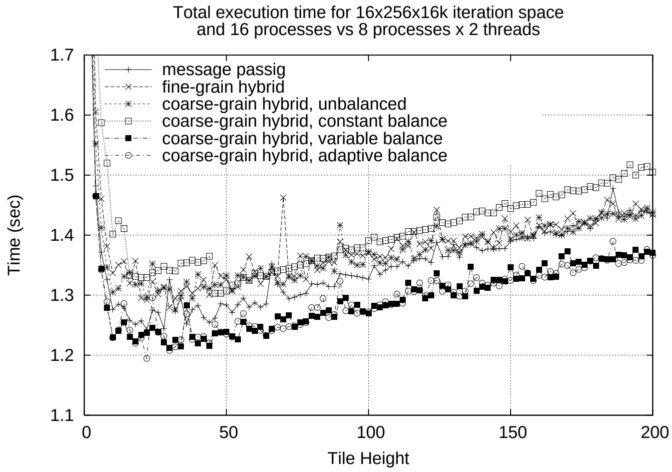
iteration space	message passing	hybrid
$16 \times 256 \times 16k$	1×16	1×8
$32 \times 256 \times 16k$	2×8	1×8
$64 \times 256 \times 16k$	2×8	1×8
$128 \times 256 \times 16k$	2×8	2×4
$256 \times 256 \times 16k$	4×4	2×4

Table 1: Selected process topologies for the message passing and for the hybrid models

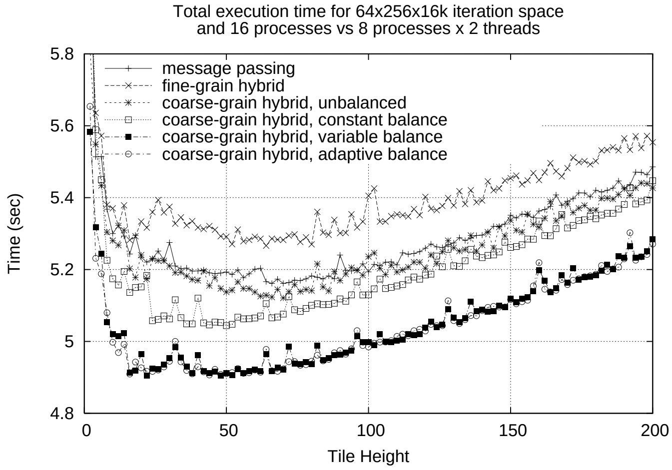
The overall experimental results for ADI integration and the above iteration spaces are depicted in Fig. 3 (MPICH) and Fig. 5 (ChaMPIon/Pro). These results are normalized in respect to the message passing execution times, so as to allow for straightforward quantitative comparison. Furthermore, granularity measurements for various iteration spaces are depicted in Fig. 4 for the MPICH library.

Following conclusions can be drawn from the thorough investigation of the obtained performance measurements (MPICH library):

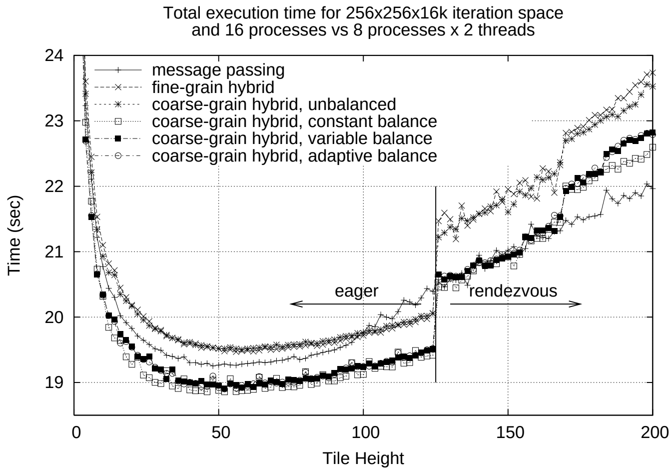
- Unoptimized hybrid parallelization (fine-grain, coarse-grain unbalanced) is often worse than the pure message passing approach. If no load balancing is applied to the hybrid model, its relative performance compared to the pure message passing model exclusively depends on the appropriateness of the selected pro-



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



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



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

Figure 4: Granularity results for ADI integration (MPICH, 8 dual SMP nodes)

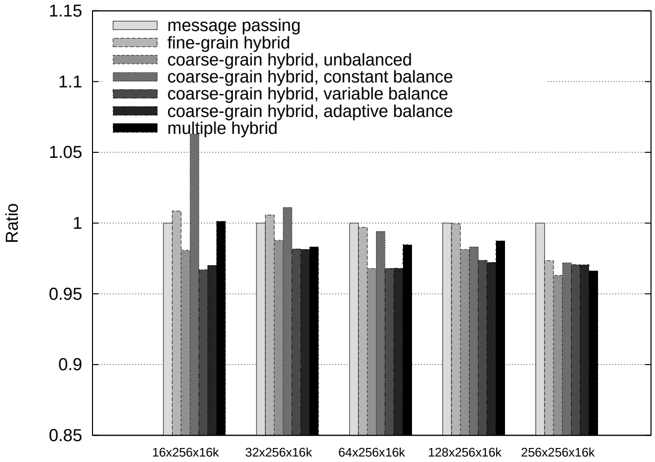


Figure 5: Comparison of hybrid models (ADI integration, 4 dual SMP nodes, ChaMPIon/Pro, various iteration spaces)

cess topology, given a specific algorithm and iteration space. As anticipated, the hybrid implementation performs better for the $32 \times 256 \times 16k$ iteration space, even without load balancing, as the assumed process topology of the hybrid implementation eliminates the largest fraction of the message passing communication required. In almost all other cases, the pure message passing model performed better than the unbalanced hybrid implementations.

- The coarse-grain hybrid model generally performs better than the fine-grain alternative for most iteration spaces. However, the coarse-grain model is not always more efficient than its fine-grain counterpart (e.g. $128 \times 256 \times 16k$, $256 \times 256 \times 16k$). This observation reflects the fact that the poor load balancing of the simple coarse-grain model diminishes its advantages compared to the fine-grain alternative.

- When applying constant static balancing, in some cases the balanced coarse-grain implementation is less effective than the unbalanced alternatives (e.g., $16 \times 256 \times 16k$, $32 \times 256 \times 16k$). This can be attributed both to inaccurate theoretical modeling of the system parameters for the calculation of the balancing factors, as well as to the inappropriateness of the constant balancing scheme for boundary processes. Generally, the constant balancing approach performs well for relatively symmetric process topologies, with few of the processes lying at the topology boundary.

- When applying variable static balancing, the coarse-grain hybrid model was able to deliver superior performance to the message passing alternative in all cases. The performance improvement lies in the range of 2-12%, also depending upon the selected process topology in each case.

- The same performance improvement is also observed for the adaptive dynamic balancing approach, thus confirming that the required information for the application of efficient load balancing can be obtained at run-time, with minimal overhead. Overall, the variable static and the adaptive dynamic balancing techniques have delivered the best parallel performance in almost all cases, and have proven to be the most reliable and efficient parallelization approaches.

The granularity results in Fig. 4 reveal that these two balancing techniques (variable, adaptive) perform better than any other implementation for almost all granularities. Some performance degradations observed at certain threshold values of the tile height z can be ascribed to the transition from the eager to the rendezvous MPICH message protocol, occurring at 128000 bytes (for instance, see Fig. 4(c) at $z = 125$). Depending on the message size, MPICH resorts to three different message passing communication protocols, namely short, eager and rendezvous, each assuming a different approach in terms of intermediate buffering and/or receiver notification. Generally, there are certain trade-offs when transitioning between these protocols, that mainly correlate sender-receiver synchronization with the overhead of intermediate buffering, therefore performance implications should not be surprising.

The performance evaluation of all models with the aid of the ChaMPIon/Pro implementation mainly aimed at providing additional insight for the multiple hybrid implementation, since we simply repeated the same experiments on a smaller scale. Fig. 5 indicates that an appropriately balanced hybrid funneled implementation is usually even slightly better than the multiple hybrid implementation, confirming our intuition that appropriate load balancing could mitigate the limited multi-threading support and also avoid the overhead associated with ensuring thread safety of the message passing library. As many popular message passing libraries currently provide only limited multi-threading support, appropriately load balancing the funneled hybrid model arises as a feasible and efficient parallel programming alternative. In addition, even if the thread safety of the message passing library allows all threads to perform inter-node communication, the underlying network infrastructure might be too restrictive for the efficient utilization of this feature (e.g., one NIC to be used by multiple threads).

6.2 Diffusion Equation

We conducted similar experimental evaluation for the DE kernel, by using both the MPICH and the ChaMPIon/Pro libraries. The results are summarized in Fig. 6 for the MPICH library, and in Fig. 7 for the ChaMPIon/Pro implementation. In all cases, the execution times have been normalized to the message passing model.

We have observed similar behavior as in the ADI benchmark. Variable and adaptive balancing deliver the best results, by providing a performance improvement in the range 3-22%. The relative performance improvement in the DE kernel is significantly higher in all iteration spaces

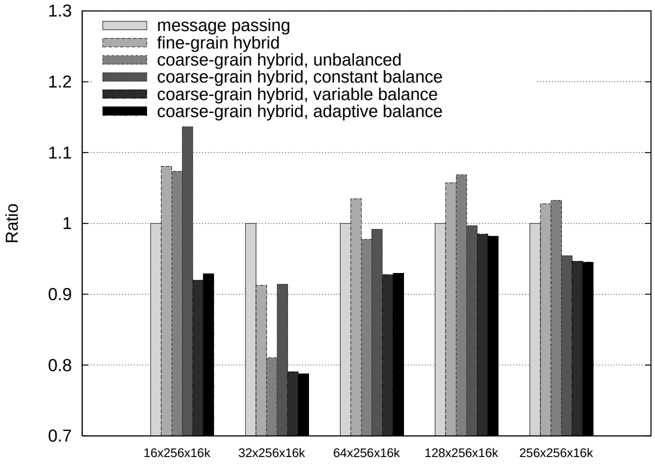


Figure 6: Comparison of hybrid models (Diffusion equation, 8 dual SMP nodes, MPICH, various iteration spaces)

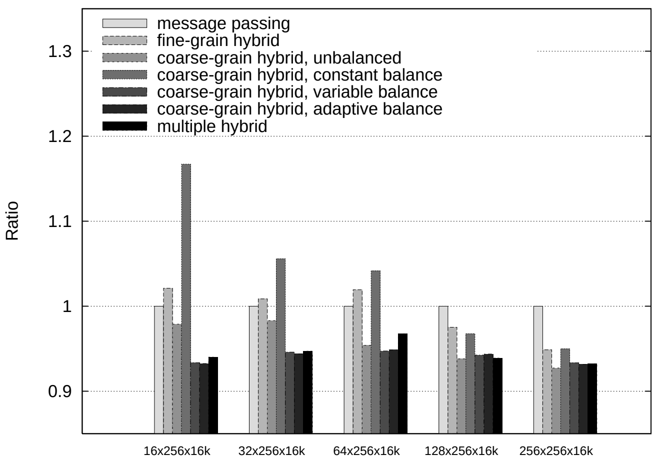


Figure 7: Comparison of hybrid models (Diffusion equation, 4 dual SMP nodes, ChaMPIon/Pro, various iteration spaces)

compared to ADI, reflecting the fact that hybrid programming can be particularly beneficial in algorithms with high communication to computation demands. The reader should also note that although DE imposes more communication overhead than ADI, it is still heavily computation-bound. Since the advantage of hybrid programming lies in the efficient utilization of the underlying architecture for communication purposes, it could be even more suitable for the parallelization of communication-bound applications, as long as these do not saturate the available memory bandwidth.

Fig. 7 further confirms that multiple hybrid parallelization is not more efficient than either the variably or the adaptively balanced coarse-grain funneled hybrid implementation, and even leads to slight performance degradation in the $64 \times 256 \times 16k$ iteration space. Also, the relative performance improvement is significantly higher for the DE kernel (more than 5% in all cases) in comparison to ADI (around 3% for the various iteration spaces), thus the same qualitative behavior observed at the MPICH experiments is exhibited also for the ChaMPIon/Pro library, though on a smaller scale.

7 CONCLUSIONS

This paper discusses load balancing issues regarding hybrid parallelization of tiled loop algorithms. We propose three load balancing schemes, namely two static techniques (constant, variable), as well as a dynamic (adaptive) one. Static balancing is based on application-system modeling, so as to determine a suitable task distribution between threads. On the other hand, dynamic balancing extracts similar information at run-time, by profiling the actual application performance and re-evaluating the initial load balancing strategy. We have compared both message passing and hybrid implementations, and demonstrated the performance improvements that can be attained by resorting to an appropriately balanced coarse-grain hybrid implementation.

The experimental evaluation has confirmed that for the algorithms considered here unoptimized hybrid parallelization may perform poorly, even worse than the monolithic message passing paradigm. Moreover, coarse-grain SPMD-like hybrid parallelization does not always outperform the fine-grain incremental alternative, as the poor load balancing of the former diminishes its relative advantages in respect to the latter (e.g. higher parallelization efficiency, overlapping computation with communication, no thread re-initialization overhead). However, when applying variable or adaptive load balancing, the coarse-grain funneled hybrid model was able to deliver superior performance in respect to message passing parallelization, and even exhibit similar performance to the coarse-grain multiple alternative, but without requiring additional thread-safe support from the message passing library.

Generally, the performance improvements are proportional to the intrinsic communication demands of the parallelized algorithm, and also depend on the efficiency of the process topology assumed in each case. Conclusively, efficient load balancing can effectively mitigate limited multi-threading support, and delivers superior performance compared to standard message passing implementations. All models that have been proposed can easily be adopted to more generic parallelization techniques, as we have emphasized on the elements of applicability and simplicity.

A Static Load Balancing

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, under a balancing factor bal, the master thread will assume the execution of a fraction $\frac{bal}{T}$ of the process's computational load, while each of the other $T-1$ threads will be assigned a fraction of $\frac{T-bal}{T(T-1)}$ of the remaining 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

$$t_{tile} = max\{t^m_{tile}, t^o_{tile}\} \tag{6}$$

where

$$t^m_{tile} = t_{comp}\left(\frac{bal}{T}\frac{Xz}{P}\right) + \sum_{\substack{i=1 \\ i \in \mathbb{S}_{\vec{p}}}}^{N} t_{comm}\left(\frac{d_i P_i Xz}{X_i P}\right) \tag{7}$$

the tile execution time of the master thread, while

$$t^o_{tile} = t_{comp}\left(\frac{T-bal}{T(T-1)}\frac{Xz}{P}\right) \tag{8}$$

the tile execution time of a non-master thread.

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^m_{tile} must be equal to t^o_{tile}. If this is not the case, that is if $t^m_{tile} \neq t^o_{tile}$, there can always be a more efficient load balancing strategy, by assigning more work to the more lightly burdened thread(s). Consequently, for minimal completion time under the assumed mapping, it holds

$$t^m_{tile} = t^o_{tile} \tag{9}$$

Assuming that the computation time t_{comp} is a linear function of the number of iterations, that is, $t_{comp}(ax) = at_{comp}(x)$, (7), (8) and (9) can be easily combined to deliver (1).

□

B Dynamic Load Balancing

Proof. We assume that the initial thread load balancing is in general non-optimal in practice, that is, if t^m_{comp}, t^m_{comm} the average tile computation and communication times of the master thread and t^o_{comp} the average tile computation time of a non-master thread, it holds

$$\begin{aligned} t^m_{tile} &\neq t^o_{tile} \Rightarrow \\ t^m_{comp} + t^m_{comm} &\neq t^o_{comp} \end{aligned}$$

Such suboptimal load balancing between the threads can be ascribed to inaccurate theoretical system modeling, as well as to the various approximations in performance estimation. Our goal is to determine a new balancing factor bal', such that the average tile execution times become equalized for all threads, that is

$$t'^m_{comp} + t'^m_{comm} = t'^o_{comp} \quad (10)$$

We assume that the time required for the computation of a tile is proportional to the number of iterations associated with that tile. Formally, we assume

$$t^m_{comp} \sim \frac{bal}{T} \frac{\prod_{i=1}^{N} X_i z}{P} \quad (11)$$

$$t^o_{comp} \sim \frac{T - bal}{T(T-1)} \frac{\prod_{i=1}^{N} X_i z}{P} \quad (12)$$

Because of (11), it holds

$$t'^m_{comp} = \frac{bal'}{bal} t^m_{comp} \quad (13)$$

and due to (12), it also holds

$$t'^o_{comp} = \frac{T - bal'}{T - bal} t^o_{comp} \quad (14)$$

Moreover, assuming communication time invariance, we approximate

$$t'^m_{comm} \simeq t^m_{comm} \quad (15)$$

By combining (13), (14) and (15), (10) delivers

$$\frac{bal'}{bal} t^m_{comp} + t^m_{comm} = \frac{T - bal'}{T - bal} t^o_{comp} \Rightarrow$$

$$bal' = \frac{balTt^o_{comp} - bal(T - bal)t^m_{comm}}{(T - bal)t^m_{comp} + balt^o_{comp}} \quad (16)$$

By further approximating t^o_{comp} by $\frac{T-bal}{bal(T-1)} t^m_{comp}$ because of (11) and (12), we can easily deduce (2)

□

ACKNOWLEDGMENT

The Project is co-funded by the European Social Fund (75%) and National Resources (25%).

REFERENCES

Andonov, R., Balev, S., Rajopadhye, S., and Yanev, N. (2003). Optimal Semi-oblique Tiling. *IEEE Trans. on Parallel and Distributed Systems*, 14(9):944–960.

Ayguadé, E., González, M., Martorell, X., and Jost, G. (2004). Employing Nested OpenMP for the Parallelization of Multi-zone Computational Fluid Dynamics Applications. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium 2004 (IPDPS 2004)*, Santa Fe, New Mexico, USA.

Calland, P., Dongarra, J., and Robert, Y. (1997). Tiling with Limited Resources. In Thiele, L., Fortes, J., Vissers, K., Taylor, V., Noll, T., and Teich, J., editors, *Application Specific Systems, Architectures and Processors*, pages 229–238. IEEE Computer Society Press.

Cappello, F. and Etiemble, D. (2000). 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. IEEE Computer Society.

Chavarría-Miranda, D. G. and Mellor-Crummey, J. M. (2003). An Evaluation of Data-parallel Compiler Support for Line-sweep Applications. *Journal of Instruction Level Parallelism*, 5:1–29.

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

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

Drosinos, N. and Koziris, N. (2004). 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 (IPDPS 2004)*, page 15, Santa Fe, New Mexico.

El-Ghazawi, T., Carlson, W., and Draper, J. (2002). UPC Language Specification (V 1.1.1). In *2nd UPC workshop*, Washington DC, USA.

Henty, D. S. (2000). 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. IEEE Computer Society.

Hodzic, E. and Shang, W. (1998). On Supernode Transformation with Minimized Total Running Time. *IEEE Trans. on Parallel and Distributed Systems*, 9(5):417–428.

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

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

Kumar, V., Grama, A., Gupta, A., and Karypis, G. (2003). *Introduction to Parallel Computing*, pages 205–208. Addison Wesley.

Legrand, A., Renard, H., Robert, Y., and Vivien, F. (2004). Mapping and Load-balancing Iterative Computations. *IEEE Trans. on Parallel and Distributed Systems*, 15(6):546–558.

Loft, R. D., Thomas, S. J., and Dennis, J. M. (2001). 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. ACM Press.

Majumdar, A. (2000). Parallel Performance Study of Monte Carlo Photon Transport Code on Shared-, Distributed-, and Distributed-Shared-Memory Architectures. In *Proceedings of the 14th International Parallel Processing Symposium (IPPS 2000)*, page 93, Cancun, Mexico.

Merlin, J. and Hey, A. (1995). An Introduction to High Performance Fortran. *Scientific Programming*, 4(2):87–113.

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

Nakajima, K. (2003). OpenMP / MPI Hybrid vs. Flat MPI on the Earth Simulator: Parallel Iterative Solvers for Finite Element Method. In *Proceedings of the 5th International Symposium on High Performance Computing (ISHPC 2003)*, pages 486–499, Tokyo-Odaiba, Japan.

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

Su, M., El-Kady, I., Bader, D., and Lin, S. (2004). A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization. In *Proceedings of the International Conference on Parallel Processing (ICPP'04)*, pages 373–379, Montreal, Canada.

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

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