

Performance Analysis of a Hybrid Parallel Linear Algebra Kernel

Sue Goudy, Lorie Liebrock, and Steve Schaffer

New Mexico Institute of Mining and Technology
Socorro, New Mexico 87801
spgoudy@nmt.edu

Abstract. The focus of this paper is the performance of a kernel from a two-dimensional iterative solver. Complexity models for hybrid parallelization of block Gauss-Seidel relaxation are derived. We examine system parameters that can affect the performance of hybrid code. Complexity estimates are tested for a variety of decomposition strategies and problem sizes. Results from the Intel Teraflops supercomputer and from the Vplant visualization cluster at Sandia National Laboratories¹ are presented. We show that the benefits of hybrid programming for this iterative solver are limited, on both the massively parallel system and the Linux cluster.

1 Introduction

This research focuses on the development of a performance model for hybrid parallelism that combines shared memory and message passing models. Can a simple model, that is, one with a few parameters that characterize the system, capture application behavior well enough to predict performance on different systems? For a particular SMP cluster, what is the most effective mapping of problem domain to processor allocation and programming paradigm? Comparing the prediction of performance given by the model to actual computation provides validation of the model.

Clusters of symmetric multiprocessors exist in many forms, from massively parallel supercomputers such as the Intel Teraflops to multiprocessor workstations linked on a network. In between are moderately parallel clusters such as the Vplant visualization cluster at Sandia National Laboratories. Performance modeling of algorithms for these computing platforms is a subject of current research.

Parallel computers can be classified on the basis of the tightness with which the processors are coupled in their ability to communicate with each other. In a symmetric multiprocessor system, the intercommunication occurs via an internal memory subsystem, such as a bus or a crossbar. In a cluster, the intercommunication occurs across a network: this can be a high speed interconnect as in Intel Teraflops or some relatively slow connection such as ethernet in a local area network.

Within a symmetric multiprocessor, the software paradigm for interprocess communication can take a variety of practical forms. For simplicity, these forms are herein categorized as shared memory; there must be a mechanism for distinguishing shared data from local data. For a distributed system of computers, all data sharing takes place via explicit interchange or message passing. In a clustered SMP system, a hybrid or mixed-mode parallel programming paradigm combines message passing with shared memory for data exchange.

As a “testbed” for evaluation of our modeling technique, we chose semicoarsening multigrid (SMG), a robust numerical technique for solution of elliptic partial differential equations in two and three dimensions. The algorithm developed by Schaffer [9] contains both coarse grained and fine grained parallel execution paths. This character is most evident in the relaxation kernel of SMG, making this kernel a good basis for experimentation in a clustered SMP environment. All experiments described here use Gauss-Seidel relaxation as implemented in two-dimensional SMG.

The rest of this paper is organized as follows. Section 2 describes the hybrid implementation of bicolor block Gauss-Seidel relaxation and a line solver based on Wang’s partition method [11]. Section 3 presents

¹ Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

complexity models and performance estimates for different mappings of the two-dimensional test problem onto a set of processors. Section 4 describes experiments and presents results from the Intel Teraflops and the Vplant cluster. The final section compares experimental results with predictions of the model and outlines future work.

2 Implementation of the Sparse Linear Algebra Kernel

Semicoarsening multigrid is a method for the solution of linear systems arising from the discretization of partial differential equations. The salient feature of multigrid methods is the recursive coarsening of the original discretization scale, a process that decreases the amount of parallel work on subsequent coarse grids. These methods can be implemented to run on parallel computers [1, 6]; however, a significant sequential component exists in the processing of the multigrid levels. Each discretization scale produces values that are needed by the next coarser scale computation. The parallelism in the algorithm occurs within a multigrid level. For detailed information on multigrid methods, refer to [5]. We examine the hybrid performance of the relaxation kernel that is the core of our multigrid solver.

Denote a partial differential equation by $Lu = f$. Discretization of the continuous equation yields a matrix equation $L^h u^h = f^h$. This linear system is solved at each point in the grid

$$\Omega^h = \{x_{i,j} : i = 1..m, j = 1..n\} .$$

Semicoarsening multigrid relates a coarse grid to Ω^h by inclusion of one half of the fine grid lines:

$$\Omega^{2h} = \{x_{i,2j-1} : i = 1..m, j = 1..n/2\} .$$

2.1 Relaxation

Experiments on a single processor have demonstrated that relaxation can account for roughly ninety percent of computation time for our two-dimensional solver. In this sense, we can regard the computational time of relaxation as representative of the entire algorithm. We confine our hybrid parallel study to the routine *RELAX2()*. Our implementation of block Gauss-Seidel relaxation proceeds in two phases: creation of a system of tridiagonal equations from the plane equation and the subsequent line solves. We investigate Wang's partition method and a block multithreaded variant of Wang's method. With respect to the line solver, the term "block" refers to the simultaneous application of the Wang algorithm to a group of lines.

The discretized operator matrix, in 9-point stencil format, together with the discretized right hand side

$$\begin{pmatrix} nw_{ij} & n_{ij} & ne_{ij} \\ w_{ij} & c_{ij} & e_{ij} \\ sw_{ij} & s_{ij} & se_{ij} \end{pmatrix} u_{ij} = f_{ij} ,$$

form a linear system that will be solved at each point of the discretized two-dimensional domain.

The blocks of the two-dimensional grid are the even numbered lines, called "red" lines, and the odd numbered, or "black", lines. The plane equation is decomposed into systems of tridiagonal equations (or line equations). Due to the compactness of the stencil, solution of the lines can proceed in parallel. Single line equations or blocks of line equations can be passed to the Wang solver.

In stencil format the right hand side of a line solve for one line of either color is

$$r_{ij} = f_{ij} - \begin{pmatrix} nw_{ij} & n_{ij} & ne_{ij} \\ 0 & 0 & 0 \\ sw_{ij} & s_{ij} & se_{ij} \end{pmatrix} u_{ij} .$$

The modified equation for a single line,

$$C u_{ij} = (w_{ij} \ c_{ij} \ e_{ij}) u_{ij} = r_{ij} ,$$

where C represents the central triad of the stencil, is then passed to the tridiagonal solver.

2.2 Line Solvers

The line solves in relaxation take a significant portion of the computation time. Therefore a tridiagonal solver that will be included in a parallel version of semicoarsening multigrid must be very efficient. The selection of an appropriate solver can be made in conjunction with the decomposition of the two-dimensional domain onto the parallel processors.

The Thomas algorithm, which is used in the serial version of our solver, is equivalent to Gaussian elimination without partial pivoting. This method is well-conditioned for diagonally dominant tridiagonal systems, such as those arising in the solution of partial differential equations. An elegant analysis of the algorithm appears in Strikwerda's text on finite difference methods [10]. Recurrences on the solution variable inhibit parallelization; however, the Thomas algorithm can be used in parallel relaxation as long as conditions permit storage and solution of a line within a single processor. To make our solver handle more general types of decompositions, we incorporated a parallel tridiagonal solver.

Cyclic reduction [8], a divide-and-conquer algorithm, has long been recognized as an efficient and readily parallelizable solver for tridiagonal systems. Other research [2] has used cyclic reduction in the setting of two-dimensional smoothing, with some measure of success. Our implementation of cyclic reduction generated intense traffic in small messages. Also, efficient addition of threads to it appeared to be nontrivial. We chose to implement a simpler algorithm to do the direct solves for the red and black lines of the Gauss-Seidel relaxation scheme.

Wang's partition method [11] consists of a data distribution phase, followed by forward and backward Gaussian elimination phases. Like the Thomas algorithm, Wang's method is well-conditioned for diagonally dominant tridiagonal systems. Wang's partition method is suitable for vector or parallel computers, if the number of processors is small compared to the order of the system of equations. The primary reason for this constraint is that the forward and backward elimination phases involve serial communication. Details of the communication pattern in the Wang solver can be found in Section 3.2.

2.3 Hybrid Parallelization Strategy

Conversion of Schaffer's algorithm to run on distributed memory computer systems proceeded in a series of steps. The Single Program Multiple Data (SPMD) model was chosen for implementation. It was decided that domain decomposition onto the parallel computer would be done prior to invocation of semicoarsening multigrid. The Message Passing Interface (MPI) library [3] was used to transfer information explicitly at subdomain boundaries. The first version of parallel relaxation was designed to work only for strip decompositions that did not partition a tridiagonal solve across processors; the Thomas method was suitable for the line solves. Next, the inherently serial Thomas algorithm was replaced by a distributed parallel implementation of Wang's partition method.

Conversion of the pure MPI block relaxation algorithm to run on clusters of symmetric multiprocessors began at the finest grain. OpenMP [7] directives for loop-level thread execution were placed in the Wang line solver. Not surprisingly, due to the small amount of parallel work within a line, this technique showed very little performance gain. Next, a blocked Wang partition algorithm using OpenMP was developed. Finally, the setup of the tridiagonal equations was added to the threaded block. Complexity analysis is presented only for the final version of the relaxation kernel.

3 Complexity Models and Performance Estimates

Our complexity model for relaxation has two parts: the setup of the tridiagonal systems that are derived from the plane equation and the solution of those line equations. The model is refined to specify the computation and communication times for each of these parts.

Implementation of relaxation follows the SPMD model. A logical $P_I \times P_J$ processor mesh is assumed to contain the data for the decomposed two-dimensional domain. Each processor in the mesh executes the same set of instructions on its portion of the data. The relaxation pseudocode in Fig. 1 does not contain explicit communication calls. Where data is indexed outside loop limits, the implementation provides data from off the local processor by using MPI library functions.

In order to simplify the computational model for relaxation, assume a square discretized domain of $N \times N$ points, where $N = 2^n$. Assume that P_I and P_J are also powers of two and that P_I and P_J need not be equal. The multigrid levels are processed sequentially; parallelization occurs within each level.

```

for P in 1..P_I, 1..P_J do                               // logical processor grid is 2d mesh
  N_I ← N/P_I ; N_J ← N/P_J                             // problem domain i, j = 1..N
  for lev ← 1..log2N do
    N_Jlev ← (2/2lev)N_J
    RELAX2(A, u, f, N_I, N_Jlev)                       // A = Lh, i = 1..N_I, j = 1..N_Jlev

procedure RELAX2(A, u, f, N_I, N_Jlev)
  red ← 2..N_Jlev, step 2
  black ← 1..N_Jlev - 1, step 2
  for j in red, black do
    for i ← 1..N_I do                                     // set up right hand side
      rij ← fij
        - (nwijui-1, j+1 + nijui, j+1 + neijui+1, j+1)
        - (swijui-1, j-1 + sijui, j-1 + seijui+1, j-1)
      SOLVE(Cjuj = rj)                               // invoke tridiagonal solver on line j

```

Fig. 1. Pseudocode for 2D relaxation.

3.1 Setup of Line Equations

We derive a complexity model for the setup phase of relaxation on all semicoarsened levels. Referring to the pseudocode in Fig. 1, the time for an invocation of *RELAX2* at multigrid level *lev* is

$$T_{RELAX2}^{lev} = T_{SETUP}^{lev} + T_{SOLVE}^{lev} . \quad (1)$$

Calculations on any black line are independent of calculations on any other black line; this is due to the compact stencil. Red line computations are similarly independent. Therefore, within a block of either color, computation can proceed in parallel. The time for setting up the equations for the blocks of red and black lines can be separated into computation time, T_{SETUP}^{calc} , and communication time, T_{SETUP}^{comm} . Communication events are present implicitly in the pseudocode as references outside the array bounds of the solution variable u . The time for an invocation of the tridiagonal solver, T_{SOLVE}^{lev} , can also be split into computation and communication times. This will be done in the Section 3.2.

Assume ϕ is the average time for a floating-point operation, then

$$T_{SETUP}^{calc} = 2(12\phi)(N_J^{lev}/2)N_I . \quad (2)$$

Assume λ is latency of communication between hosts and τ is the time for transfer of one data word, then

$$T_{SETUP}^{comm} = 2(T_{EW} + T_{NS}) = 2(2(\lambda + \tau N_J^{lev}) + 2(\lambda + \tau N_I)) . \quad (3)$$

T_{EW} is the time required for exchange of u -values on east and west boundaries of the subdomain contained within a processor. T_{NS} is the time for exchange of u -values on north and south boundaries. These boundary exchanges occur during equation setup for red and black lines. Due to semicoarsening, the amount of data transferred north/south remains the same on all multigrid levels, while the amount of data transferred east/west decreases as the grids become coarser.

Combining equations, and letting $\mu(M)$ denote $\lambda + \tau M$, the time for line equation setup on level lev is modelled as

$$T_{SETUP}^{lev} = (12\phi N_I)(2^{1-lev} N_J) + 4\mu(2^{1-lev} N_J) + 4\mu(N_I) . \quad (4)$$

Approximately half the computation occurs on the finest grid; however, only a small part of the communication occurs on that level. The full model for the setup phase of relaxation is obtained by summing Equation 4 over lev , from 1 to $n = \log_2 N$.

$$T_{SETUP} \approx 2 \cdot 12\phi(N_I N_J) + n \cdot (4\mu(2N_J/n) + 4\mu(N_I)) . \quad (5)$$

Derivation of the model for tridiagonal solves appears in the next section. Addition of parameters for thread overhead will complete the parameterization of the relaxation model.

3.2 Line Solvers

We derive time complexity of Wang's partition method. Operation counts are presented first for solution of a single line equation. Then the model for a block-structured version of Wang's algorithm is developed. Finally, the models for threaded versions of the setup and solve phases are combined into the full model for the relaxation kernel. For ease of presentation, we develop the hybrid model for Wang's method on the finest multigrid level.

Assume that P processing elements (or PE's), logically numbered 0 to $P - 1$, are available to solve a tridiagonal system of order NP . Before invocation of Wang's algorithm, distribute N equations to each of the P processors. The partition method [11] then proceeds by these steps:

1. Upper-triangularize diagonal blocks.
2. Eliminate superdiagonals of diagonal blocks.
3. Eliminate nonzero elements of superdiagonal blocks.
4. Eliminate below main diagonal.
5. Eliminate columns above main diagonal.
6. Solve the diagonal system.

Figure 2 shows pseudocode for the forward elimination phase of the Wang algorithm. Message transfers are present implicitly as references outside the array bounds. Step (4a) requires that logical PE ip receive values of $cc(N)$, $aa(N)$, and $u(N)$ from logical PE $ip - 1$ in order to calculate new values for $cc(N)$ and $u(N)$ in the local block for PE ip . This data dependency causes the serialization of messages across the logical processor array. Parallel processing is resumed in step (4b). A similar serialization of message traffic occurs for the backward elimination phase. These communication patterns limit the scalability of Wang's method. Refer to the full pseudocode in Appendix 5 for details.

```

// (4) Eliminate below main diagonal.
// (4a) Serial messages.
for ip ← 1 to P - 1
  β ← bb(N)/cc(0)
  cc(N) ← cc(N) - β * aa(0)
  u(N) ← u(N) - β * u(0)
// (4b) Parallel computation.
for ip ← 1 to P - 1
  for j ← 1 to N - 1
    β ← bb(j)/cc(0)
    aa(j) ← aa(j) - β * aa(0)
    u(j) ← u(j) - β * u(0)

```

Fig. 2. Pseudocode for a portion of Wang's partition algorithm.

The Wang partition algorithm is readily parallelized for distributed memory computers. Parallel solution of a tridiagonal system by Wang's method is modeled as

$$T_W = 21\phi N_I + (\lambda + 4\tau) + (P_I - 1)(8\phi + 2(\lambda + 4\tau)) \approx 21\phi N_I + P_I(8\phi + 2(\lambda + 4\tau)) , \quad (6)$$

where ϕ represents the average time of a floating-point operation, and λ , τ represent communication latency and bandwidth parameters. The presence of the term P_I indicates that two communication sweeps and a small amount of computation must be serialized.

There are $2 \cdot N_J^{lev} / 2$ line solves in each invocation of *RELAX2*, giving the total time for solution of the tridiagonal line equations on multigrid level *lev*,

$$T_{SOLVE}^{lev} = N_J^{lev} T_W \quad , \quad (7)$$

Substitution for $T_{SOLVE}^{lev=1}$ in Equation 5 leads to a potential optimization. We see that communication for Wang's single-line method occurs $2P_I N_J$ times in an invocation of *RELAX2* on the finest level. If a block structure with delayed communication is used, the number of communication events can be dramatically reduced. The factor N_J can be "pulled inside" the Wang method, so that

$$T_{WB} = 21\phi N_I N_J + (P_I - 1)(8\phi N_J + 2\mu(4N_J)) \quad , \quad \text{where } \mu(M) \text{ denotes } \lambda + \tau M \quad . \quad (8)$$

Now there are fewer, possibly more costly, serialized communication events. The amount of parallel computation stays the same, but the computation that occurs in the serialized portion of the algorithm increases from 8ϕ to $8\phi N_J$.

At an early stage of our investigation into hybrid parallel relaxation, we introduced OpenMP directives at the loop level in Wang's method. The results for the solver of a single tridiagonal system were not impressive. The algorithm is "communication bound", even for grids with large numbers of points. Therefore, threading within a single line solve does not effectively reduce the run time of the partition method.

3.3 Hybrid Model for Relaxation

The block-structured Wang method, as well as line equation setup, are suitable for further parallelization using OpenMP. The strategy is to assign subblocks of line solves to the threads. Each time *RELAX2* is invoked, an OpenMP parallel region is entered and threads are created. These threads are destroyed on exit from the parallel region. Communication inside a parallel region is done by only one of the threads. Logical $P_I \times P_J$ process meshes are assumed. Each process can use up to θ threads.

The model for block hybrid Wang is expanded to include terms for thread scheduling overhead Θ_S and the number of threads θ

$$T_{WBT} = 5\Theta_S + 21\phi N_I(N_J/\theta) + (P_I - 1)(8\phi N_J + 2\mu(4N_J)) \quad . \quad (9)$$

Addition of the solver term, T_{WBT} , to Equation 4 and addition of terms for thread creation and thread scheduling to relaxation setup yields the hybrid model for relaxation on the finest grid level,

$$T_{RELAX2}^{lev=1} = \Theta_C + 4\Theta_S + 12\phi N_I(N_J/\theta) + 4\mu(N_J) + 4\mu(N_I) + T_{WBT}^{lev=1} \quad . \quad (10)$$

The full relaxation model is related to this single-level model in the following ways. Total calculation on all the coarse grids is approximately equal to the calculation on the finest grid. Parallel overhead, including thread scheduling, on all grids is roughly $n = \log_2(N)$ times the parallel overhead on the finest grid. To complete the performance estimations, we specify decompositions of the problem domain into $N_I \times N_J$ subgrids in the full model,

$$\begin{aligned} T_{RELAX2} \approx & 2 \cdot 12\phi N_I(N_J/\theta) + n \cdot (4\mu(2N_J/n) + 4\mu(N_I)) + n \cdot (4\Theta_S + \Theta_C) \\ & + 2 \cdot \kappa\phi N_I(N_J/\theta) + n \cdot (P_I - 1)(8\phi N_J/n + 2\mu(4N_J)) + n \cdot 5\Theta_S \quad . \end{aligned} \quad (11)$$

The number of floating-point operations in the block Wang method depends on the number of processes P_I that hold data for a line solve. That is, $\kappa = 13$ when $P_I = 1$; otherwise, the value of κ is 21.

The model for block Gauss-Seidel relaxation, Equation 11, is deceptively simple. The parameter for communication performance depends on the layout of the data decomposition on the logical grid of processes. Average floating-point performance for the tridiagonal solver is different from that of the setup phase. In section 4.3 we discuss practical aspects of the determination of the model parameters ϕ and μ .

4 Experiments and Results

We use the same model problem for all experiments described in this paper. The domain $\Omega = [0, 1] \times [0, 1]$, with a subdomain $[0.5, 1] \times [0.5, 1]$. Different partial differential equations are defined in each region depicted in Fig. 3, yielding a system with discontinuous coefficients. In Regions 1 and 2, respectively, the equations are

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{and} \quad \frac{\partial^2 u}{\partial x^2} + 100.0 \frac{\partial^2 u}{\partial y^2} = 0 . \quad (12)$$

Dirichlet boundary conditions $u = 0$ are defined.

The square domain of the model problem can be mapped onto the processors of a cluster of symmetric multiprocessors in a number of ways.



Fig. 3. Domain of the model problem, showing regions with different partial differential equations.

4.1 Decomposition Strategies

In the timing studies described in Sections 4.4 and 4.5, we use two-dimensional cartesian domains with evenly spaced discretization grids. These fine grids are divided as evenly as possible into rectangular subdomains that are allocated to MPI tasks on a cluster of SMPs. The assignment of coarse grid points to MPI tasks is made on the basis of the owner of the fine grid points from which the coarse subgrids are defined. This decomposition leads to load imbalance, in the sense that some processors have no work to do on some coarse levels. Several techniques for dealing with “sleeping processors” have been investigated during the history of multigrid. We have chosen to ignore the sleepers, since that is an easy and acceptable solution [1].

Consider mapping a square problem domain to MPI tasks. Assume that we have a cluster of four nodes, with each node having two processors. Across the top of Fig. 4, three possible decompositions of a square domain are shown. Below each decomposition, we show a possible distribution to MPI tasks in the cluster. Data is labeled by the MPI rank of the process that owns the data. Arrows indicate the transfer of information at processor boundaries. The areas enclosed by dashed lines represent data from the domain decomposition that is assigned to each MPI task in a parallel job. The boxes that enclose MPI task data represent the computer node that is executing the individual tasks comprising the parallel job. The decomposition shown in Fig. 4(a) can be operated on by four MPI tasks running on four processors, or by four MPI tasks, each using two threads. In the former case, one CPU in each of the four nodes is idle; in the latter, all processors are busy.

For fixed numbers of grid points, performance of domain-to-processor mappings is compared with predictions from the relaxation models. For scaled speedup studies, the problem domain is sized so that each *compute node* has the same number of fine-grid points. Within a node, these points are divided evenly among the MPI tasks or the OpenMP threads. Results for pure message passing and hybrid paradigms are presented.

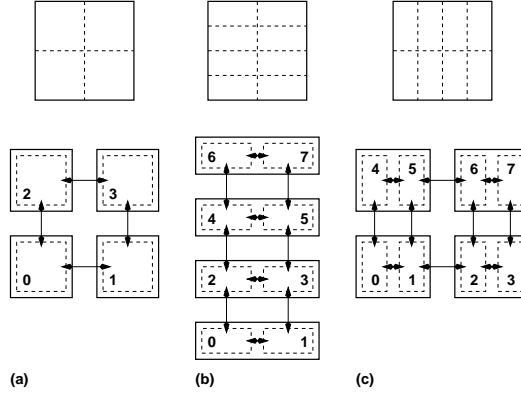


Fig. 4. Possible domain to MPI task mappings for a 4-node, 2-way SMP cluster.

4.2 Computer systems

The Intel Teraflops is a massively parallel supercomputer, composed of Intel Pentium processors connected by a proprietary communication backplane. The operating system on the compute nodes is a microkernel. Parallel jobs are loaded into memory on a portion of the compute mesh that is allocated only for that job. Nodes can be requested by a batch queuing system or by direct launch in the interactive partition.

The Vplant system is a cluster of Xeon processors connected by a Myrinet network. The compute nodes run the Linux operating system. Nodes are available only through the Portable Batch System (PBS) and are scheduled by the Maui Scheduler. There are two processor partitions available for general computation; one of these has slower processors than the other. For this set of studies we made no attempt to use heterogeneous collections of Vplant nodes. For the sake of brevity we report only the timings from the slower nodes, since these are more numerous and permit a larger parallel run.

Tests on the Intel Teraflops were run in the interactive partition as the only interactive user on the system. Tests were run on Vplant during a period of relatively low usage; frequently a sequence of timing studies was the only active job.

4.3 Measurement of Model Parameters

Latency and bandwidth were measured, using a pingpong test, on the systems where our timing studies were done. The values so obtained were not appropriate for all communication patterns in *RELAX2*, so halo exchange tests were also run. Communication times for a few representative message sizes are shown in Table 1. The data are a composite of all communication data that we obtained from our microbenchmarks. We attempted to use this average performance parameter μ in our first model. Closer examination of the microbenchmark output led us to conclude that there were directional differences in the communication parameters.

We have recently learned that our test systems use different conventions for assignment of MPI tasks; one uses a cyclic distribution and the other uses block distribution. At the time of preparation of this paper, we are incorporating the MPI task distribution into the model. We expect that the more accurate reflection of communication events that do not leave a node, as well as those that contend for a single communication link, will make the model a better predictor of performance. For example, in the model for Vplant, the communication time for north/south transfers should be larger due to the contention in the single communication link as it is accessed by two MPI tasks. Also, about half the east/west transfers in the serialized message sections of the Wang solver should be at memory-to-memory speed instead of network speed.

We derived the model parameters ϕ_{avg} and $\bar{\phi}_{avg}$, time for floating-point operations in the setup and solve phases of relaxation, respectively. Table 2 presents these parameters for Intel Teraflops and the Vplant cluster. The values in the table show the effects of memory subsystem contention when both processors on a node are used. Measurements of the basic operations (addition, multiplication, division) were made in the

Table 1. Transfer times in microseconds for a few MPI message sizes

	Intel Tflops		Vplant	
λ	29.1		13.7	
M	$\mu(M)$	τM	$\mu(M)$	τM
128	43.1	14.0	20.1	6.4
256	57.2	28.1	26.5	12.8
512	85.3	56.2	39.3	25.6
1024	142.	113.	64.9	51.2

context of two-dimensional arrays. These measured performance values were combined with operation counts to produce the averages. This approach ignored differences in performance due to cache utilization, for the domain sizes we tested.

Table 2. Average times in nanoseconds for floating-point operations

	ϕ_{add}	ϕ_{mult}	ϕ_{div}	ϕ_{avg}	ϕ_{avg}
Tflops, 1ppn	33.6	35.2	116.	34.4	50.3
Tflops, 2ppn	54.0	56.6	174.	55.3	81.5
Vplant, 1ppn	10.2	10.2	23.3	10.2	12.1
Vplant, 2ppn	19.1	18.5	24.3	18.8	20.4

We do not want to model explicitly the effects of cache on processing time. Yet we need to account for these effects. The current model uses functions for effective floating-point performance, ϕ_{eff} for tridiagonal system setup and $\tilde{\phi}_{eff}$ for the line solves. These functions are constructed by measuring single *node* performance of *RELAX2* in the context of $N \times N$ data sets, where N ranges from 16 to 1024. Extraction of timings for the setup and solve phases of relaxation are gathered from instrumentation inserted in the source code. The measurements are made for one process per node and for two processes (threads) per node, and tables of the data size are input to the model. This method gives better agreement between model and measured execution times for the different data sizes than the averaging of fundamental floating-point operations.

For measurement of overheads associated with threads, we developed a small set of benchmarks based on the methods described in [4]. Both of the systems we used have two-processor nodes, so the thread scheduling time is for two threads. The extremely low values for thread creation led us to believe that a thread pool is created only once, at process startup time. On Teraflops, this is the case: the second processor is used for a “thread” that is activated when parallel regions are entered. On Vplant, examination of output from the “ps” command seems to indicate that a second process is created to act as a thread.

Table 3. Thread creation and scheduling overhead (microseconds)

	θ_C	θ_S
Intel Tflops	0.003	8.003
Vplant	0.007	6.035

4.4 Fixed Number of Grid Points

The fixed domain scaling decompositions followed the scheme laid out in Figure 4. Experiments were run using $P_I \times P_J$ logical processor grids in powers of two, up to 128. The processor grids correspond to square arrays of two-way nodes, from 1 to 64. With this set of runs, we investigated the following utilization question. We want to determine the most effective way to map a fixed-size grid onto a fixed resource of two-way nodes, where execution time is the metric of map efficacy. We ask whether our model can provide useful estimates

of the relative merits of the different mapping strategies. Further, we ask if the model is workable for each of the systems we use.

Results graphs for the fixed scaling study are in Figures 6 - 9 , which appear at the end of the paper. Execution time of a relaxation cycle is compared with the prediction of the model, Equation 11. The keys in the graphs relate to Fig. 4 as follows: s denotes a single MPI task per node (a), t one MPI task with two threads on each node (a), v two MPI tasks per node with $P_I < P_J$ (b), and h two MPI tasks per node with $P_I > P_J$ (c).

The most effective way to map the data onto Teraflops nodes is to use more MPI tasks in the J-direction; $P_J > P_I$. The hybrid option t does not perform as well as the v option, despite the similarity in data decomposition. One possible explanation is that the write-back cache on the Teraflops compute nodes induces more memory operations than sending data through the network interface chip, so that on-node communications are more costly in t mode.

On Vplant, the measured performance shows that the t and v options are much more closely aligned. The data for the two options are divided in the same way; that is, the thread scheduling is done so that each thread gets a contiguous block of lines. It is likely that the performance of these two options is so similar because the MPI communication in-node goes through shared memory.

The model produces very good predictions of Teraflops performance. The Vplant predictions are less satisfactory, probably because the MPI task distribution has not been fully implemented in the model.

4.5 Scaled Speedup Study

For the scaled speedup studies, the problem domain was discretized so that each *node* received a square subdomain of size $N \times N$. Logical processor grids, of size $P_I \times P_J$ in powers of two up to 128, were used for these experiments. Our interest here is to determine the benefit of hybrid programming as a technique for scalability improvement. We ask if threads are effective as problem domain and number of processors are scaled up. We compare our model of a relaxation cycle with measured execution times for different subdomain sizes.

Results graphs for the scaled scaling study are in Figures 10 - 13 , which appear at the end of the paper. Execution time of a relaxation cycle is compared with the prediction of the model, Equation 11. The keys in the graphs relate to Fig. 4 as follows: s denotes a single MPI task per node (a), t one MPI task with two threads on each node (a), v two MPI tasks per node with $P_I < P_J$ (b), and h two MPI tasks per node with $P_I > P_J$ (c).

The benefits of hybrid programming appear to be very small in the system environments that are described in this paper. Both are dual-processor, bus-based systems. There is little opportunity for a hybrid approach with only two threads to demonstrate good speedup within a node. The numbers in Table 2 show that memory bus congestion has a significant influence on overall performance. The small messages that are generated in the two-dimensional solver do not drive the network hard enough to make the decrease in message count advantageous.

While hybrid programming has not produced striking gains in scalability, for Teraflops or for Vplant, the modeling exercise has some interesting points. A simple model that uses a few constant parameters seems to be a poor predictor of performance on these systems. Benchmarks that measure system parameters must mimic fairly closely the setting of the application that is being modeled. Care must be taken in the model, not only with communication patterns, but also with data layout and memory access patterns.

5 Conclusion

Our evaluation of block Gauss-Seidel relaxation as a good candidate for hybrid parallel techniques did not hold in practice. We found similar performance patterns on two computer systems with different system parameters and operating environments. Mixed-mode programming did not produce significant performance gains over pure message-passing. On a system with more processors per node, this may not be the case. We plan to apply the model and measurement techniques to SMP clusters with more processors in each node. The experiments on such systems could be more interesting; for example, fixing the number of nodes used, but varying the number of participating threads and MPI tasks independently.

The model in Equation 11 shows general trends reasonably well on the Intel Teraflops supercomputer. For example, scalability characteristics of the various decompositions are adequately captured. Our model correctly predicts, for some data sizes and node counts, that the hybrid decompositions will perform about the same as the variants that allocate two MPI tasks per node. We also predict the lack of scalability of the Wang method. The prediction capability is, at the time of this writing, not as good for Vplant. Where the model is not a good predictor, we believe that more careful representation of memory access patterns will improve the match of model with reality. We are currently enhancing the model to account for the MPI task distribution and its relation to utilization of communication links.

References

1. P.N. Brown, R.D. Falgout, and J.E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.
2. R.D. Falgout and J.E. Jones. Multigrid on massively parallel architectures. Technical Report UCRL-JC-133948, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory.
3. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.
4. J.M.Bull. Measuring synchronisation and scheduling overheads in openmp. In *First European Workshop on OpenMP*, Lund, Sweden, 1999.
5. J.E. Jones and S. McCormick. *Parallel Multigrid Methods*, volume 4 of *ICASE LARC Interdisciplinary Series in Science and Engineering*, pages 203–224. Kluwer Academic, 1997.
6. M.Prieto, R.Santiago, D.Espadas, I.M.Llorente, and F.Tirado. Parallel multigrid for anisotropic elliptic equations. *Journal of Parallel and Distributed Computing*, 61:96–114, 2001.
7. OpenMP Architecture Review Board, <http://openmp.org/specs/>. *OpenMP Fortran Application Program Interface*, fortran version 1.1 edition, 1999.
8. M.J. Quinn. *Parallel Computing, Theory and Practice*. McGraw-Hill, 1994.
9. S. Schaffer. A semicoarsening multigrid method for elliptic partial differential equations, with highly discontinuous and anisotropic coefficients. *SIAM Journal of Scientific Computing*, 20(1):228–242, 1998.
10. J.C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. Chapman and Hall, 1989.
11. H.H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software*, 7(2):170–183, June 1981.

A Pseudocode for Wang’s Partition Method

After selecting a one-dimensional partial differential equation for solution and performing a suitable discretization, the resulting tridiagonal system of equations must be decomposed onto the logical processor mesh. Assume that P processing elements, logically numbered $0..P - 1$, are available to solve a tridiagonal system of order NP . Before invocation of Wang’s algorithm, distribute N equations to each of the P processors. The partition method [11] then proceeds by these steps:

1. Upper-triangularize diagonal blocks.
2. Eliminate superdiagonals of diagonal blocks.
3. Eliminate nonzero elements of superdiagonal blocks.
4. Eliminate below main diagonal.
5. Eliminate columns above main diagonal.
6. Solve the diagonal system.

Steps (1) and (2) can be carried out in parallel. No communication between processing elements (PE’s) is required. Step (3) requires the communication of values $c(1)$, $aa(1)$, $bb(1)$, and $u(1)$ from logical PE $ip + 1$ to logical PE ip . This group of communications can be done in parallel. Step (4a) requires that logical PE ip receive values of $cc(N)$, $aa(N)$, and $u(N)$ from logical PE $ip - 1$ in order to calculate new values for $cc(N)$ and $u(N)$ in the local block for PE ip . This data dependency causes the serialization of messages

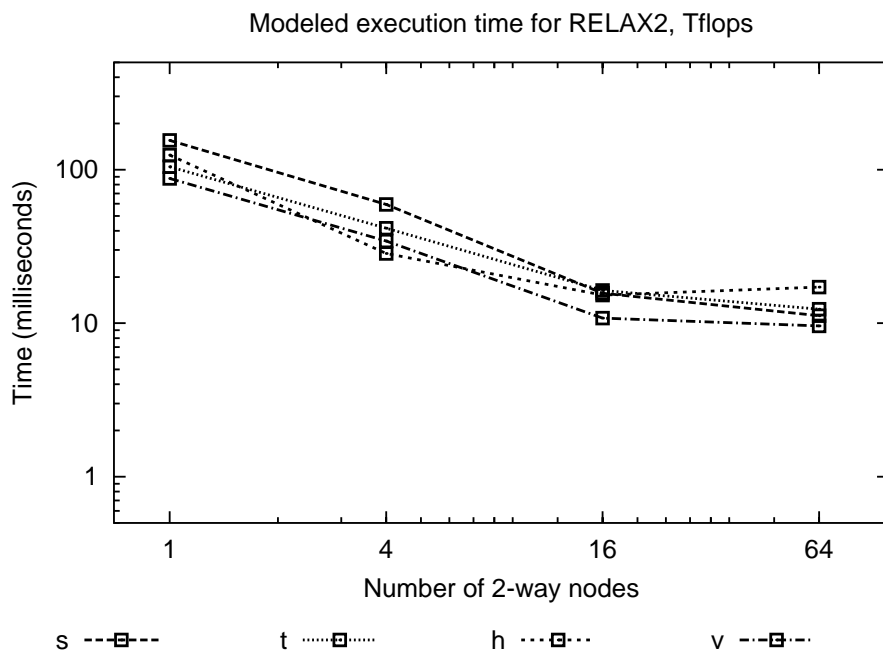
across the logical processor array. Parallel processing can be resumed in step (4b). In step (5a) the value of $u(N)$ in PE ip depends on the value of $u(N)$ in PE $ip + 1$, again causing serialization of messages between processors. Parallel processing resumes in step (5b). After these steps the system has been diagonalized so that the final solution phase can proceed in fully parallel mode in step (6). Isolation of the inherently serial communications in steps (4a) and (5a) is done to emphasize that the rest of steps (4) and (5), which contain the bulk of the computations for those steps, can be executed in parallel.

```

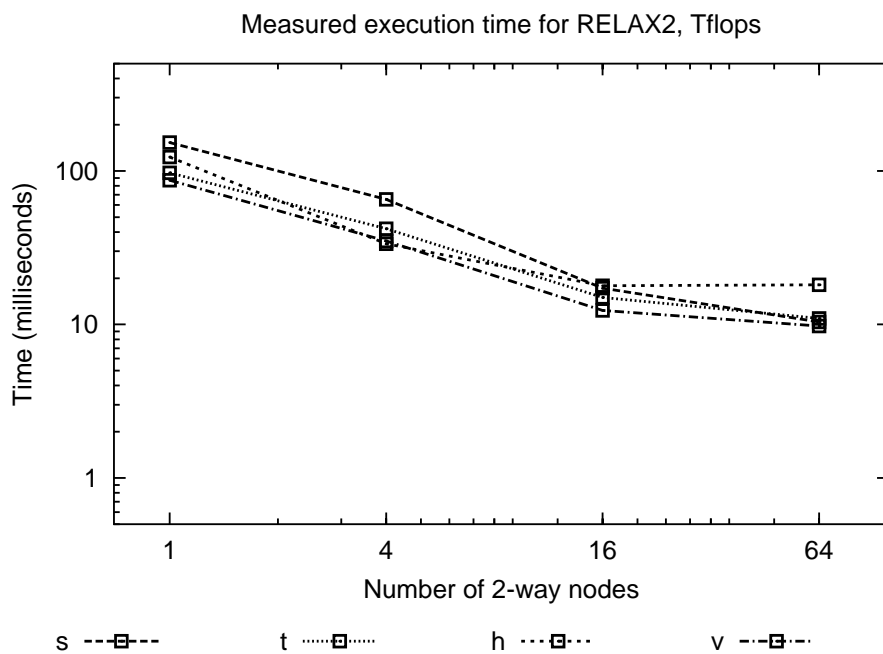
// (1) Upper - triangularize diagonal blocks.
for ip ← 0 to P - 1
  bb(1) ← b(1)
  for j ← 2 to N
    β ← b(j)/c(j - 1)
    bb(j) ← -β * bb(j - 1)
    cc(j) ← c(j) - β * a(j - 1)
    u(j) ← u(j) - β * u(j - 1)
// (2) Eliminate superdiagonals of diagonal blocks.
for ip ← 0 to P - 1
  aa(N - 1) ← a(N - 1)
  for j ← N - 1 downto 2
    α ← a(j - 1)/cc(j)
    aa(j - 1) ← -α * a(j)
    bb(j - 1) ← bb(j - 1) - α * bb(j)
    u(j - 1) ← u(j - 1) - α * u(j)
// (3) Eliminate nonzero elements of superdiagonal blocks.
for ip ← 0 to P - 2
  α ← a(N)/cc(N + 1)
  aa(N) ← -α * aa(N + 1)
  cc(N) ← cc(N) - α * bb(N + 1)
  u(N) ← u(N) - α * u(N + 1)
// (4) Eliminate below main diagonal.
// (4a) Serial messages.
for ip ← 1 to P - 1
  β ← bb(N)/cc(0)
  cc(N) ← cc(N) - β * aa(0)
  u(N) ← u(N) - β * u(0)
// (4b) Parallel computation.
for ip ← 1 to P - 1
  for j ← 1 to N - 1
    β ← bb(j)/cc(0)
    aa(j) ← aa(j) - β * aa(0)
    u(j) ← u(j) - β * u(0)
// (5) Eliminate columns above main diagonal.
// (5a) Serial messages.
for ip ← P - 2 downto 0
  α ← aa(N)/cc(N + N)
  u(N) ← u(N) - α * u(N + N)
// (5b) Parallel computation.
for ip ← P - 1 downto 0
  for j ← N - 1 downto 1
    α ← aa(j)/cc(N)
    u(j) ← u(j) - α * u(N)
// (6) Solve the diagonal system.
for ip ← 0 to P - 1
  for j ← 1 to N
    u(j) ← u(j)/cc(j)

```

Fig. 5. Pseudocode for Wang's partition algorithm.

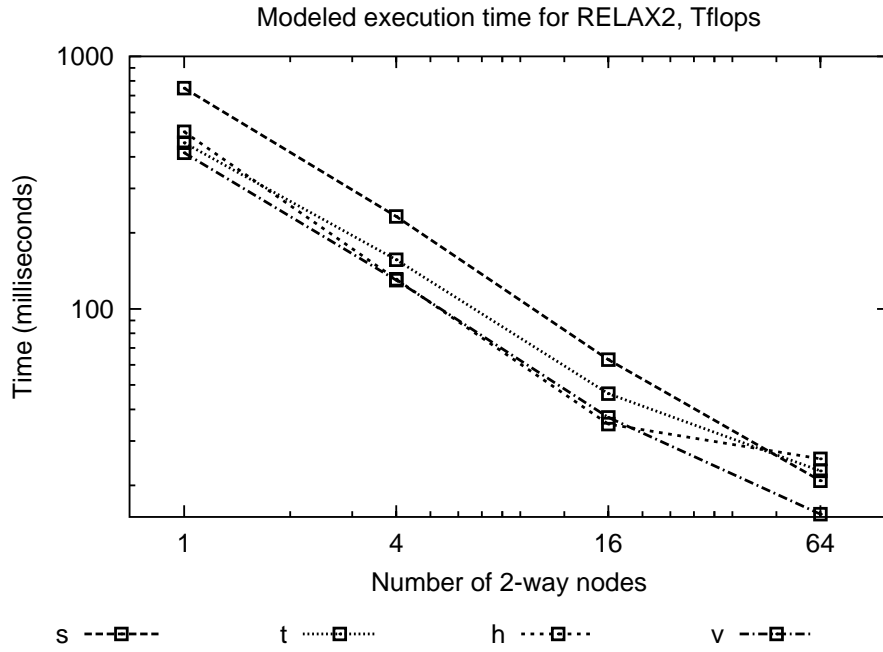


(a) Model prediction for data size 256

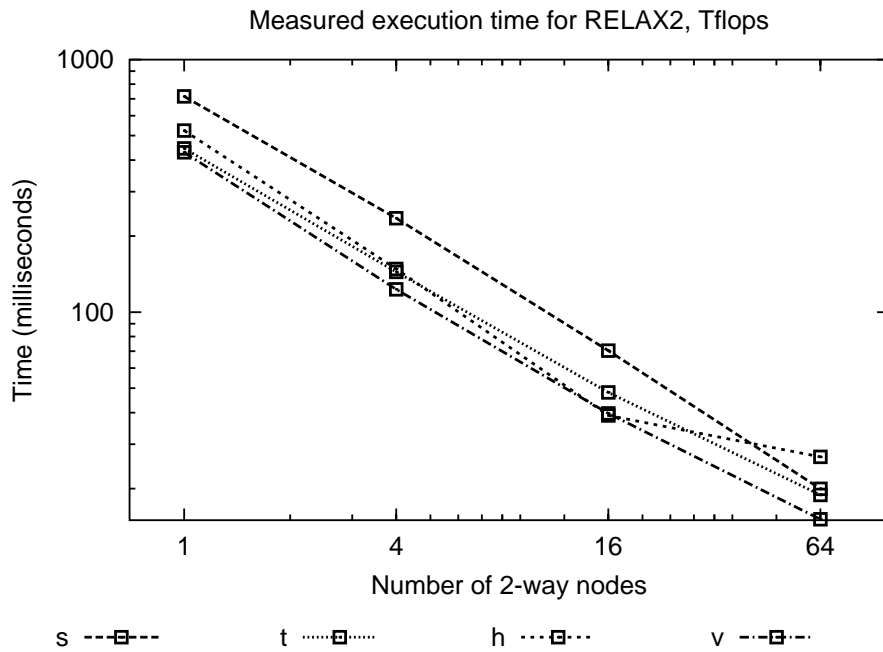


(b) Measured execution time for data size 256

Fig. 6. Comparison of model time with measured time for *RELAX2* operating on data size 256×256 . The keys represent the following task/thread distribution: *s* single MPI task per node, *t* two threads in one MPI task per node, *v* 1x2 tasks per node, *h* 2x1 tasks per node.

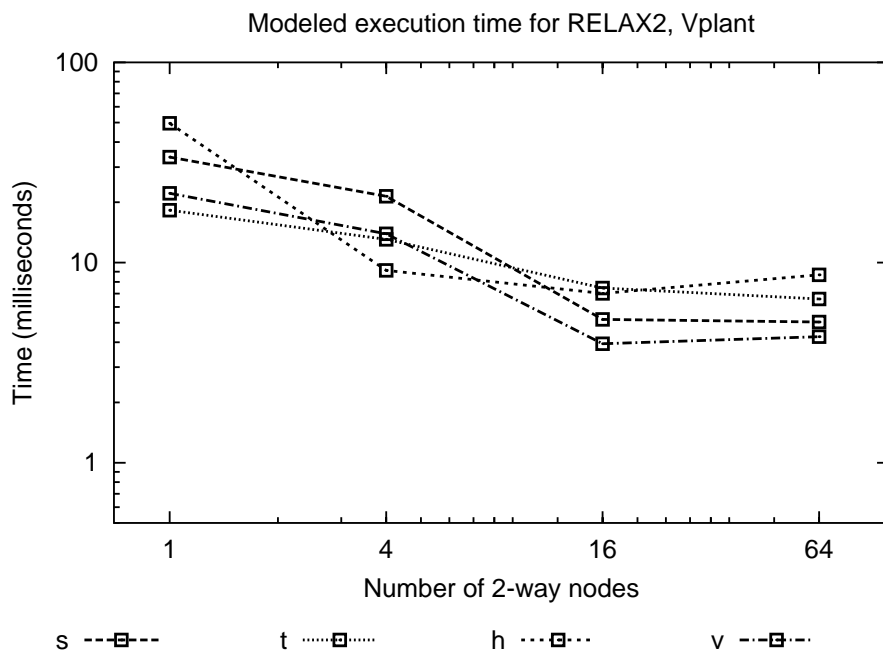


(a) Model prediction for data size 512

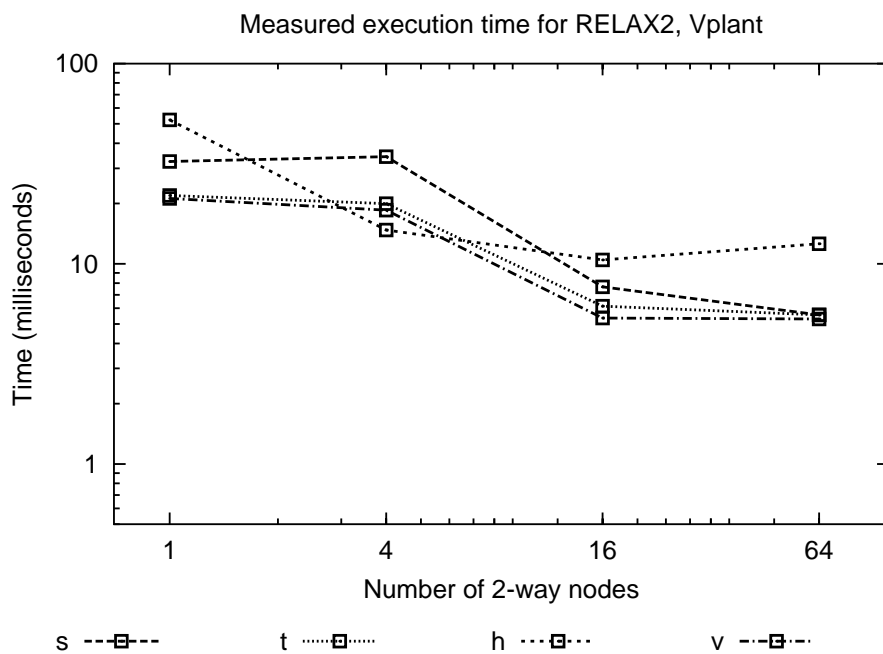


(b) Measured execution time for data size 512

Fig. 7. Comparison of model time with measured time for *RELAX2* operating on data size 512×512 . The keys represent the following task/thread distribution: *s* single MPI task per node, *t* two threads in one MPI task per node, *v* 1x2 tasks per node, *h* 2x1 tasks per node.

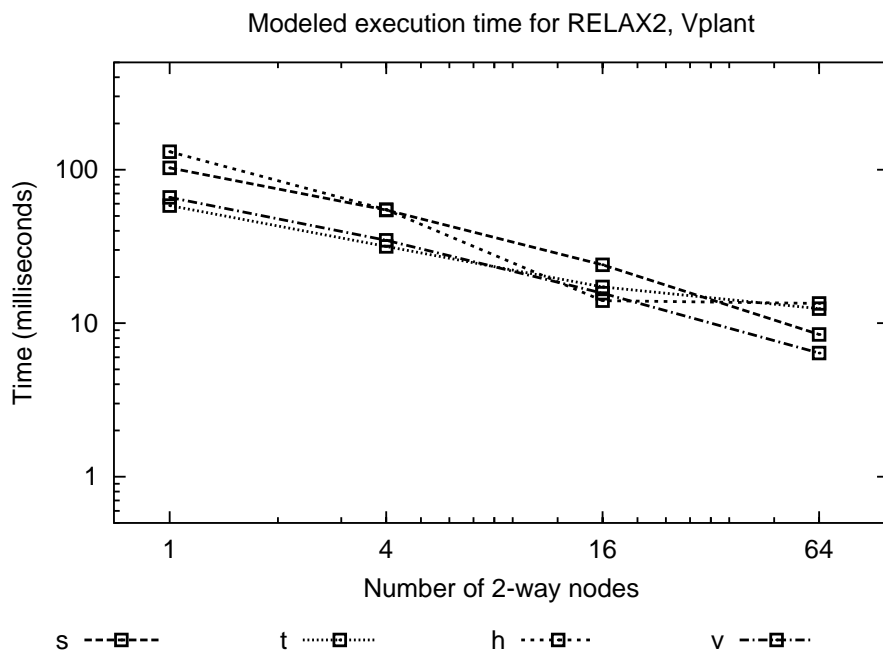


(a) Model prediction for data size 256

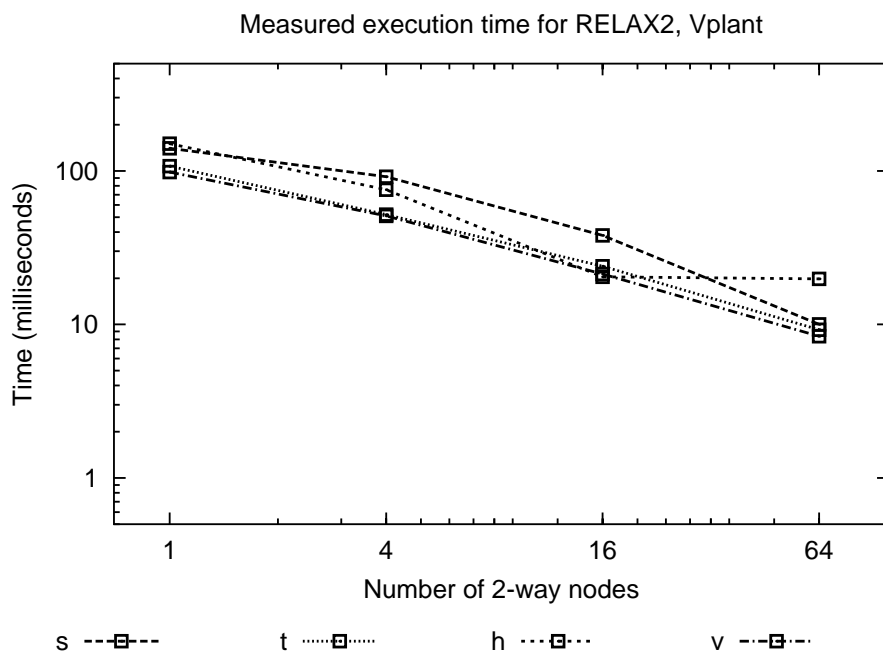


(b) Measured execution time for data size 256

Fig. 8. Comparison of model time with measured time for *RELAX2* operating on data size 256×256 . The keys represent the following task/thread distribution: *s* single MPI task per node, *t* two threads in one MPI task per node, *v* 1×2 tasks per node, *h* 2×1 tasks per node.

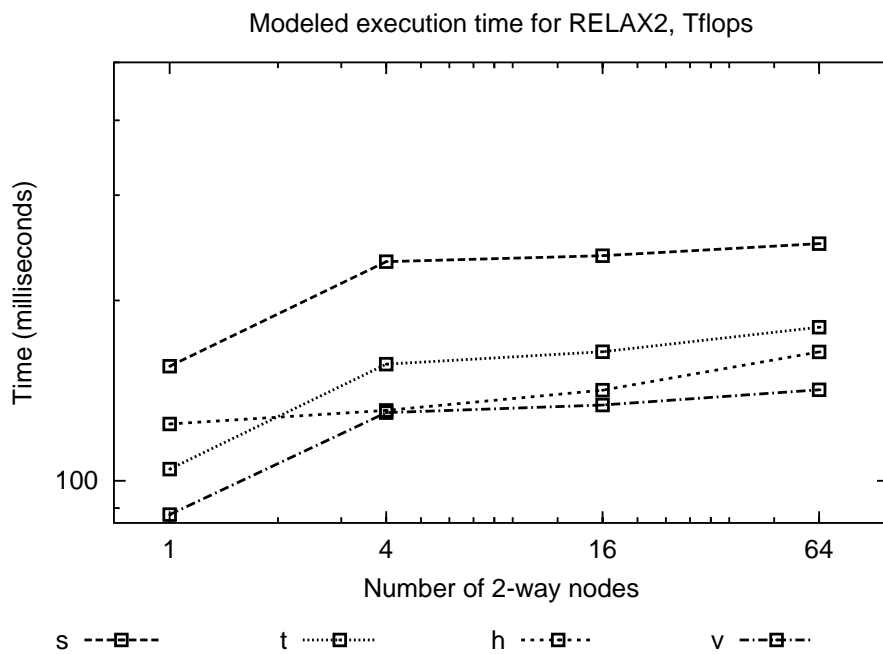


(a) Model prediction for data size 512

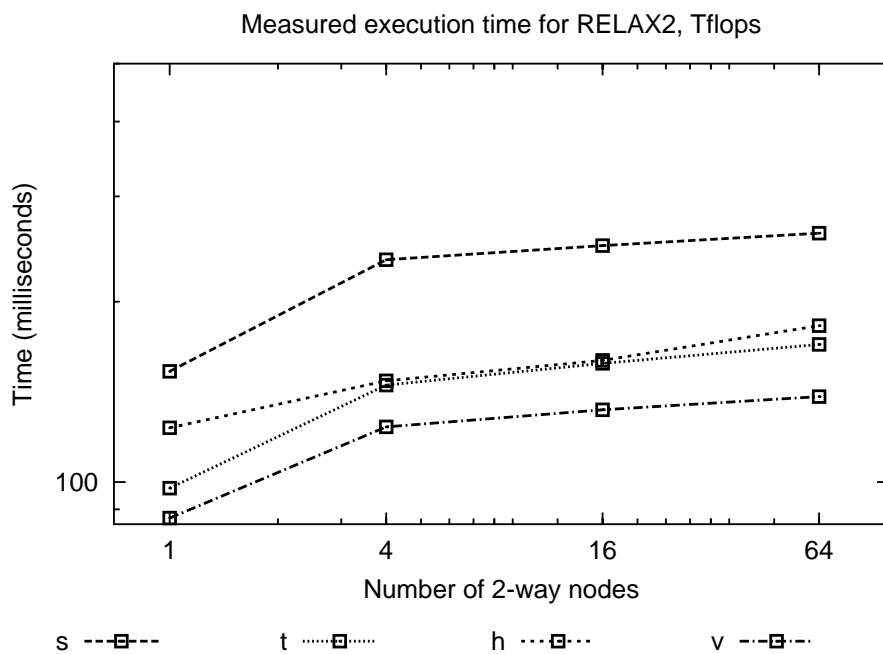


(b) Measured execution time for data size 512

Fig. 9. Comparison of model time with measured time for *RELAX2* operating on data size 512×512 . The keys represent the following task/thread distribution: *s* single MPI task per node, *t* two threads in one MPI task per node, *v* 1x2 tasks per node, *h* 2x1 tasks per node.

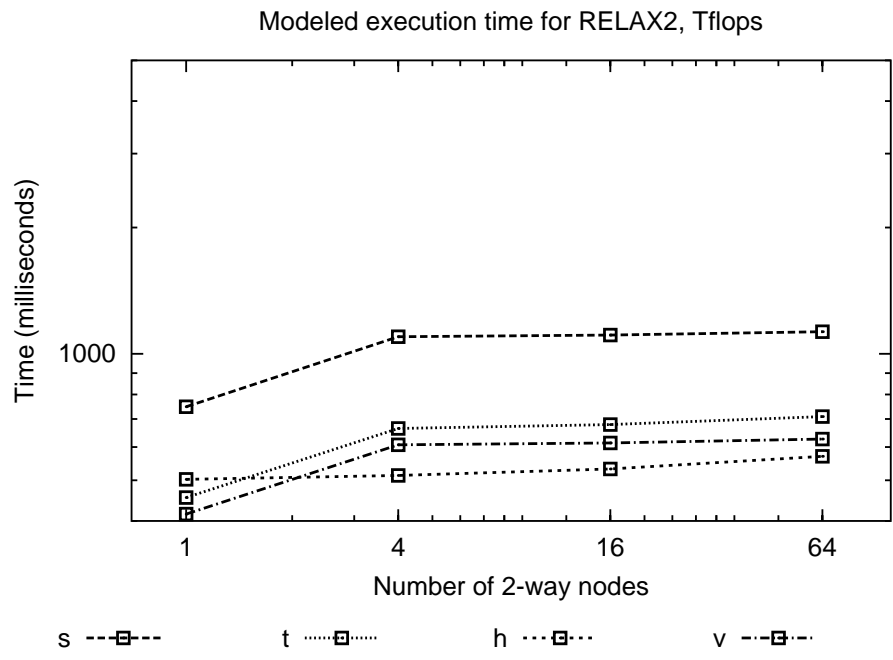


(a) Model prediction for data size 256

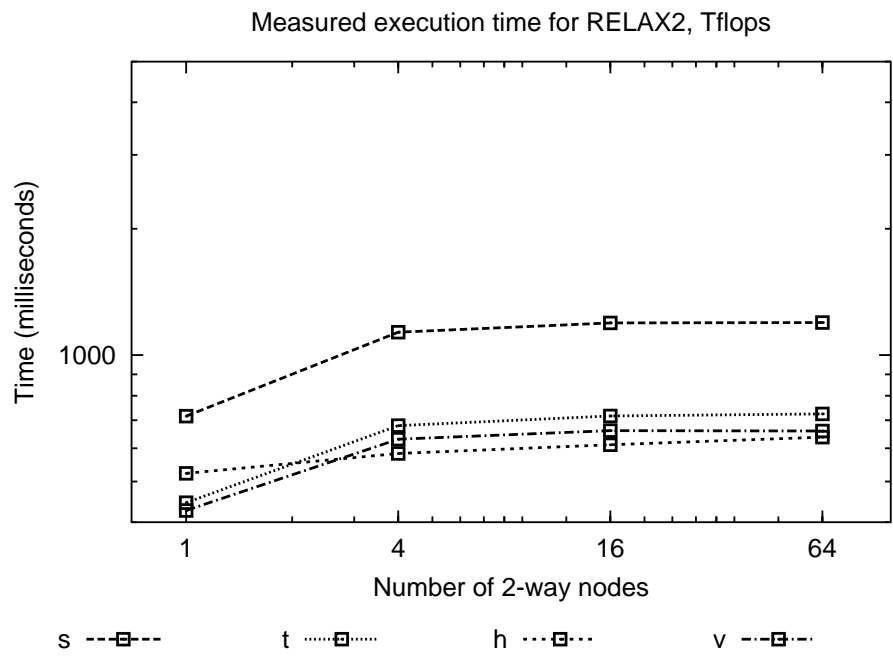


(b) Measured execution time for data size 256

Fig. 10. Comparison of model time with measured time for *RELAX2* operating on data size 256×256 . The keys represent the following task/thread distribution: *s* single MPI task per node, *t* two threads in one MPI task per node, *v* 1x2 tasks per node, *h* 2x1 tasks per node.

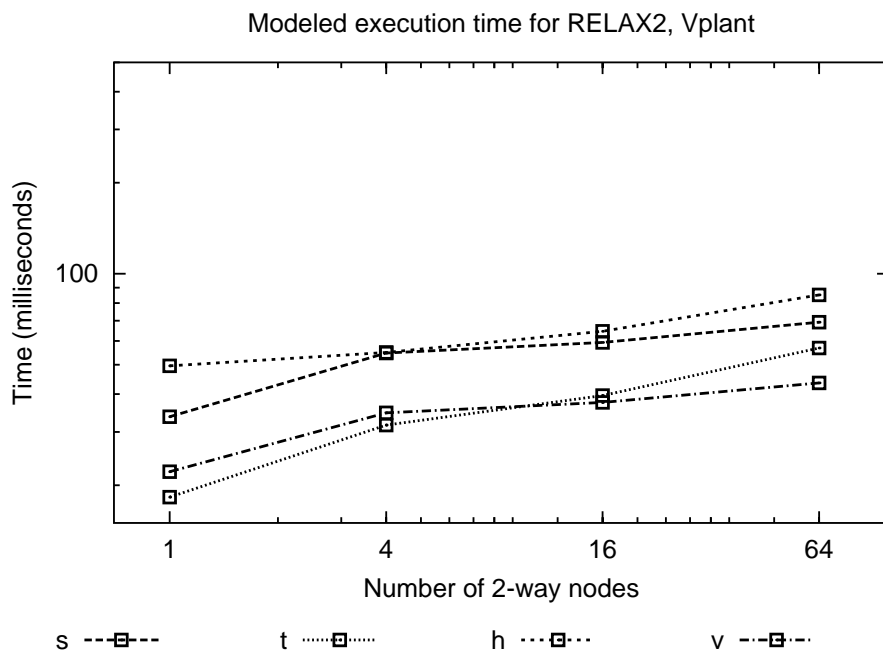


(a) Model prediction for data size 512

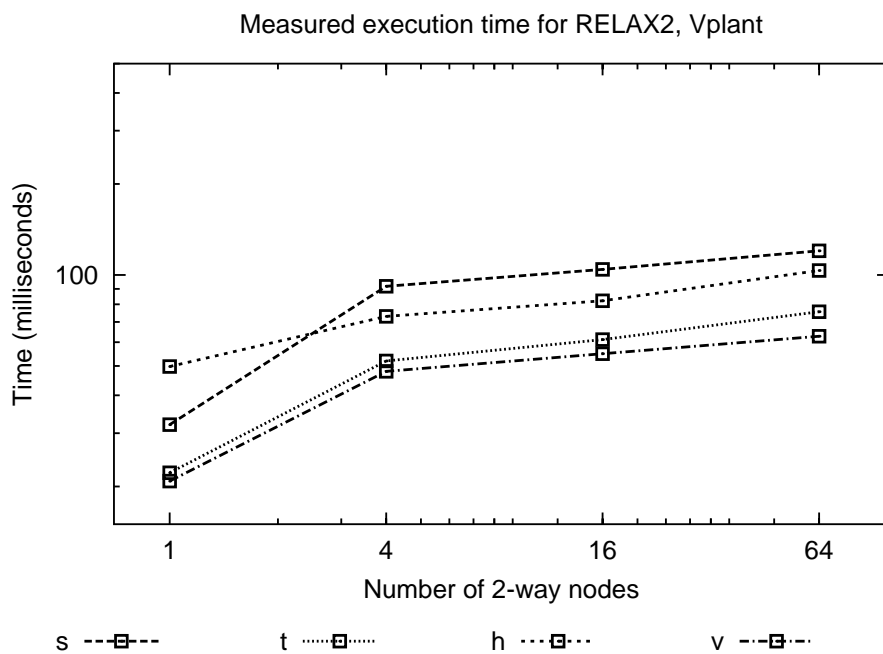


(b) Measured execution time for data size 512

Fig. 11. Comparison of model time with measured time for *RELAX2* operating on data size 512×512 . The keys represent the following task/thread distribution: *s* single MPI task per node, *t* two threads in one MPI task per node, *v* 1×2 tasks per node, *h* 2×1 tasks per node.

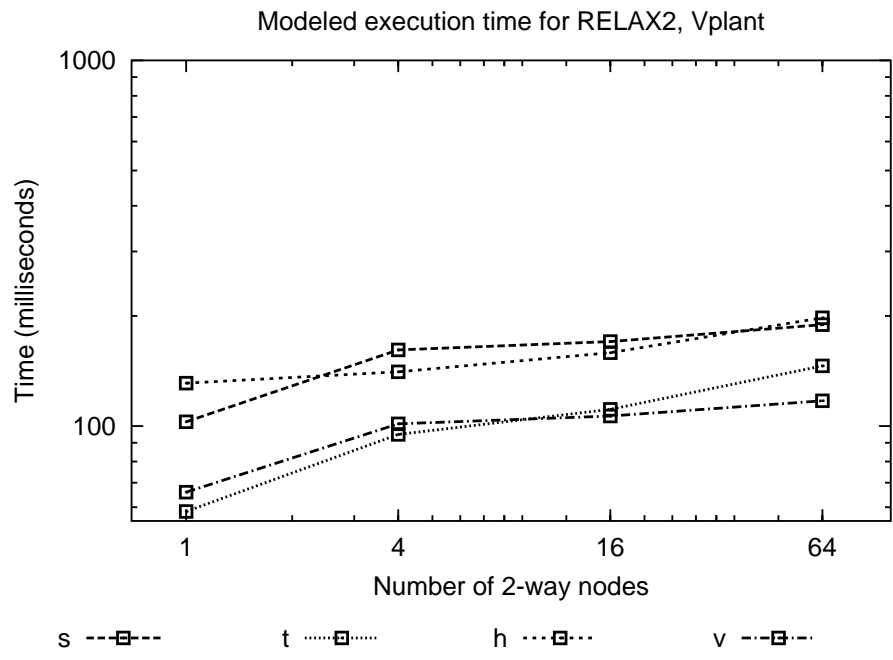


(a) Model prediction for data size 256

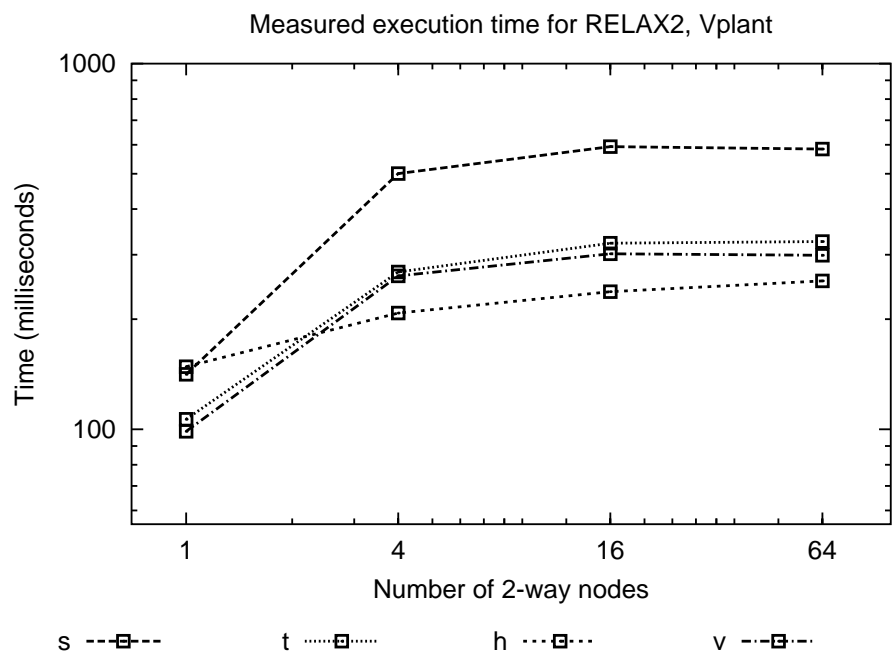


(b) Measured execution time for data size 256

Fig. 12. Comparison of model time with measured time for *RELAX2* operating on data size 256×256 . The keys represent the following task/thread distribution: *s* single MPI task per node, *t* two threads in one MPI task per node, *v* 1x2 tasks per node, *h* 2x1 tasks per node.



(a) Model prediction for data size 512



(b) Measured execution time for data size 512

Fig. 13. Comparison of model time with measured time for *RELAX2* operating on data size 512×512 . The keys represent the following task/thread distribution: *s* single MPI task per node, *t* two threads in one MPI task per node, *v* 1×2 tasks per node, *h* 2×1 tasks per node.