Chapter 1
Runtime Support and Dynamic Load Balancing Strategies for Structured Adaptive Applications *

Bongki Moon† Gopal Patnaik‡ Robert Bennett† David Fyfe§
Alan Sussman† Craig Douglas¶ Joel Saltz† K. Kailasanath§

Abstract

One class of scientific and engineering applications involves structured meshes. One example of a code in this class is a flame modelling code developed at the Naval Research Laboratory (NRL). The numerical model used in the NRL flame code is predominantly based on structured finite volume methods. The chemistry process of the reactive flow is modeled by a system of ordinary differential equations which is solved independently at each grid point. Thus, though the model uses a mesh structure, the workload at each grid point can vary considerably. It is this feature that requires the use of both structured and unstructured methods in the same code. We have applied the Multiblock PARTI and CHAOS runtime support libraries to parallelize the NRL flame code with minimal changes to the sequential code. We have also developed parallel algorithms to carry out dynamic load balancing. It has been observed that the overall performance scales reasonably up to 256 Paragon processors and that the total runtime on a 256-node Paragon is about half that of a single processor Cray C90.

1 Introduction

Detailed time-dependent, multi-dimensional numerical simulations provide an ideal vehicle to study fundamental properties of hydrocarbon flames. Physical processes in hydrocarbon flames are highly complex and interact in a strongly non-linear fashion. Numerical experimentation is an excellent way to isolate physical processes, study their interactions, or predict important properties such as flammability limits. Only highly detailed models that include complex chemistry and diffusive processes can obtain the correct flammability limits. To date, sufficiently detailed calculations for hydrocarbon flames have only been carried out for steady-state flames. Some preliminary calculations of transient methane flames with moderately detailed chemistry have been carried out at the Naval Research Laboratory (NRL) [5]. For heavier hydrocarbons, which are of more interest for Navy applications, the computational requirements are currently beyond the capabilities of current vector supercomputers. Thus, it is imperative that the current detailed flame code be ported to massively parallel computers.

The reactive flow equations of the flame modelling applications can be separated into several physical processes using the time-step splitting approach. In the chemical processes,

*This work was sponsored in part by the Office of Naval Research under contract No. N00014-93-1-0158 and contract No. SC 292-1 22193, and the NASA Microgravity Science and Applications Program.

†Dept. of Computer Science, University of Maryland, College Park, MD 20742
‡Berkeley Research Associates, P.O. Box 852, Springfield, VA 22150
§Laboratory for Computational Physics, Naval Research Laboratory, Washington DC 20375
¶Dept. of Computer Science, Yale University, New Haven, CT 06520

Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, Feb. 1995.

the ordinary differential equations (ODE) solution technique requires a different number of subcycles at each individual grid point to complete the computation within a single time step. That is, these physical processes require different amounts of computation for each grid point. On the other hand, in the fluid dynamics processes, each grid point requires the same amount of computation per time step (ignoring the effects of boundary conditions). Clearly, this distinction arises entirely from the choice of model and numerical technique used. The two types of processes require the use of both structured and unstructured methods in the same code. The Multiblock PARTI (structured) and CHAOS (unstructured) runtime support libraries have been applied to parallelize the NRL flame code with minimal changes to the existing sequential code [1, 2].

2 Computational Model

The various physical processes in the flame can be placed into one of two groups depending on their solution techniques:

1. *Structured processes*, where the computation is based on structured meshes. Certain processes, e.g. heat conduction, are represented by partial differential equations which are discretized by a finite volume technique. In these methods, the solution procedure requires a good deal of communication with neighboring points, but the amount of computation at each point is nearly the same.
2. *Pointwise processes*, where computation at each grid point can be carried out independently without requiring any communication between neighboring points, e.g. radiation, chemistry. However, the solution procedure, e.g. for chemistry, may require different amounts of computation at each grid point. This distribution of work load may change from time step to time step as the flame evolves.

A physical process can be referred to as either structured or pointwise using the criteria given above. While this distinction in structured and pointwise processes is specific to the choice of numerical techniques used in the flame code, it clearly highlights the issues of dynamic load balancing and strategies for mesh partitioning and data redistribution.

One outstanding characteristic of the computational behavior in the flame modelling code is that both structured processes and pointwise processes occur within each time step. The implication here is that there exist two conflicting demands for efficient parallel computation : (1) the necessity of regular block partitioning of the grid to take advantage of simplicity and efficiency for the structured computation, and (2) the achievement of a good load balance in the pointwise processes. Two sets of runtime support primitives, Multiblock PARTI and CHAOS, have been applied successfully to cope with both of the demands.

3 Runtime Support for Structured Computation

The Multiblock PARTI runtime support library has been developed to parallelize multiblock and multigrid codes on distributed memory machines. The library allows application programmers to lay out distributed arrays, perform data movement and distribute computation across processors in a way that load imbalance and communication is minimized. In the NRL flame code, arrays representing physical quantities at each grid point are used in the phases of structured computation, and have to be distributed across the processors. For instance, suppose the problem domain is a two-dimensional $N \times N$ grid, where N is the number of grid points in each dimension, there are P processors and the amount of computation done at each grid point is equal. Then the problem domain may be distributed in a block fashion so that each processor owns an $N/\sqrt{P} \times N/\sqrt{P}$ subgrid. Figure 1(a) illustrates how the $N \times N$ arrays, are distributed across a 2×2 grid

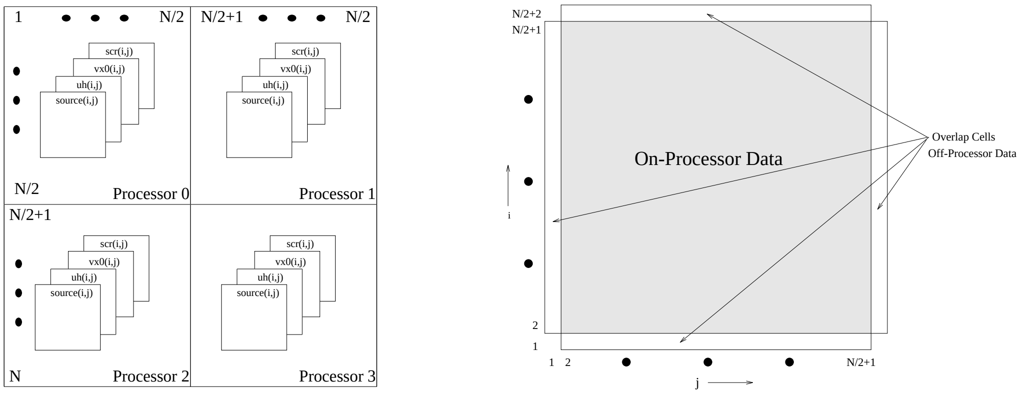

FIG. 1. (a) *Distribution of four arrays across a* 2×2 *grid of processors, and* (b) *Portion of a Distributed Array on a Processor*

```
10      do j = 1, ny
20         do i = 2, nx
30            uh(i,j) = 0.5 * (vx0(i,j) + vx0(i-1,j))
40            source(i,j) = 0.5 * (scr(i,j) + scr(i-1,j))
50         end do
60      end do
```

FIG. 2. *An example of a sequential computation loop*

of processors. Also shown are the array's global indices on each processor.

Consider the example of a typical computation loop from the sequential version of the flame code illustrated in Figure 2. This code fragment contains four arrays, `uh`, `vx0`, `scr` and `source` of size $nx \times ny$ and a few scalar values. In order to parallelize this nested loop structure, several steps must be taken. First, it must be ensured that a processor has allocated enough memory for its portion of the global array and any additional off-processor data (i.e., a layer of *overlap cells* [4] in each dimension). Second, the original loop bounds must be adjusted from global indices to local indices since the arrays no longer reside on one processor. Last, any references to data from adjacent processors (i.e., `vx0(i-1,j)`) must be resolved before entering the loop. Figure 1(b) illustrates the portion of a distributed array on a processor and the associated *overlap cells* in each dimension.

On a uniprocessor, the reference `vx0(i-1,j)` at the edge of an array returns the correct data since it is contiguous in memory. When an array is distributed across several processors, the data at the edges are no longer local to that processor; the data resides on adjacent processors. In order to preserve the correctness of the computation loop, this data must be fetched before executing the loop. Referring to Figure 1b), rows 1 and $N/2+2$ must be fetched before this array is used in the example loop. This functionality is provided by the Multiblock PARTI library.

The sequential loop in Figure 2 is parallelized by adding calls to `fill_ghost_cells()` before the loop structure. The routine `fill_ghost_cells()` calls the Multiblock PARTI data movement functions which update (fetch data from neighboring processors) the *overlap cells* for the specified arrays in the specified *dimension*. Also, the loop bounds in lines `10` and `20` are adjusted by calling `local_lower_bound()` and `local_upper_bound()` which return the local lower and upper indices, respectively, in a given *dimension*.

A processor can then perform computations on the part of the distributed array that it owns. Using the 2×2 processor mesh of Figure 1(a), when each processor executes the parallelized code fragment, the processors fill the *overlap cells* of the arrays `src` and `vx0` in the first dimension. The bounds of the parallelized loops are adjusted to $2 \leq i, j \leq \frac{N}{2} + 1$ for each processor.

In the NRL flame code, a multigrid method, based on MADPACK [3], is used as the solution technique for the elliptic equation that arises in the fluid convection model. Multigrid methods employ a number of nested meshes at different levels of resolution. The restriction and prolongation operations for shifting between different multigrid levels require moving regular array sections with non-unit strides. These operations can also be implemented using Multiblock PARTI.

4 Parallel Algorithms for Dynamic Load Balancing

To balance the work load of the pointwise processes, block-partitioned data must be redistributed across processors, and then moved back into the original locations for the next structured process. This approach inherently requires a substantial amount of communication to redistribute data every time step. This section will present new parallel algorithms which perform the redistribution of work load at low cost, namely, *Binpack* and *Binsort*. *Binpack* generates a load balancing plan for each of the participating processors, and *Binsort* is a heuristic for improving the performance of *Binpack* by reducing the communication volume for redistributing the work load.

4.1 Binpack algorithm

The computational nature of the pointwise processes allows us to formulate dynamic load balancing for such processes as a bin packing problem with a finite number of bins each of unlimited capacity. Suppose there are N objects with sizes $s_1, s_2, \ldots, s_N$ and P bins each of unlimited capacity. The goal is to distribute the N objects to P bins so that the maximal sum of object sizes which are assigned to a bin is minimized. For our purposes, the bins and the objects are processors and grid points, respectively.

The parallel *Binpack* algorithm determines which processors are heavily loaded and which processors are lightly loaded, and computes the amount of work that must be moved from heavily loaded processors to lightly loaded processors. The heavily and the lightly loaded processors are called *donor* processors and *donee* processors, respectively. Suppose there are l donor processors $\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{l-1}$ $(0 \leq l \leq P-1)$. Then, for each donor processor $\mathcal{S}_i$, the algorithm generates a list of donee processors $\mathcal{R}^i_0, \mathcal{R}^i_1, \ldots, \mathcal{R}^i_{m-1}$, and a list of work amounts $\lambda^i_0, \lambda^i_1, \ldots, \lambda^i_{m-1}$ $(1 \leq m \leq P-1)$ that should be transferred to the respective donee processors. Donor processors use this information to generate a detailed load migration plan that depicts a destination processor for each of work units.

The parallel *Binpack* algorithm generates suboptimal load balancing plans in that the maximal sum of work amounts assigned to a processor is not guaranteed to be minimal. However, it can be proved that the maximal load is bounded by $\sum_{j=1}^{N} w_j / P + l * w_{max}$ where w_j is the amount of work at grid point j and w_{max} is the maximal amount of work at any grid point.[1] This algorithm requires only one step of collective communication, followed by several steps of local computation without further communication. Since the collective communication can be efficiently implemented on most message passing parallel computers, the algorithm achieves good load balance at very low cost and scales well on varying number of processors.

[1] Detailed description and analysis of the parallel *Binpack* algorithm will appear in another paper currently in preparation.

(Number of moved grid points)	Number of processors					
	8	16	32	64	128	256
Without Binsort	33844	35732	41582	45346	45685	48092
With Binsort	16454	16877	23374	24068	30198	31817
Reduction(%)	51	53	44	47	34	34

TABLE 1

Reduction in communication volume using Binsort

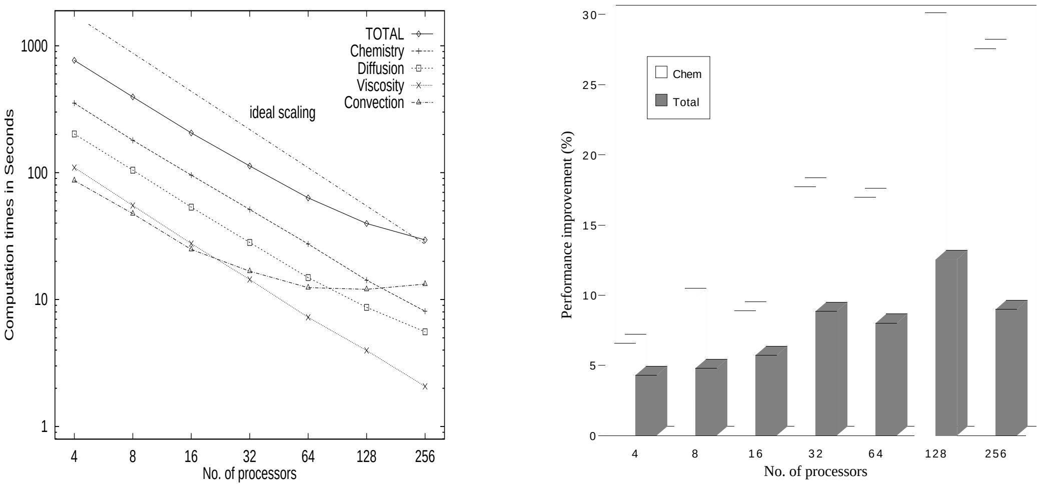

FIG. 3. (a) *Computation times, and* (b) *Performance improvement relative to block partitioning*

4.2 Binsort algorithm

As previously discussed, the structured processes and pointwise processes alternate within every time step. Since the data redistribution needs to be done every time step, reduction of communication volume is crucial. In transporting data items, using the plan for load balance generated by the *Binpack* algorithm, the *Binsort* algorithm reduces the communication volume by selecting grid points with greater amounts of work first. The *Binsort* algorithm carries out approximate sorting by classifying the grid points according to their work load into N_b bins each of unlimited capacity. For instance, if w is the amount of work required by a grid point x and $\frac{(i-1)}{N_b}\mathcal{M} \leq w < \frac{i}{N_b}\mathcal{M}$, then the grid point x is inserted into the i-th bin. Here, N_b is a constant chosen by the user and $\mathcal{M}$ is the maximal load of individual grid points.

Once all the grid points are distributed to bins, a threshold value τ is computed in such a way that $\tau = \frac{j}{N_b}\mathcal{M}$ for some $j (0 \leq j \leq N_b - 1)$. Then, only the grid points in the bins j, $j+1$, $\ldots N_b$ are considered for movement to other processors. This algorithm carries out approximate sorting of the grid points in $\mathcal{O}(N)$ time, and does not require communication between processors, provided that the excessive amount of work E_p is known to each processor p. As shown in Table 1, it has been observed that the communication volume can be reduced by up to 53 percent by applying the *Binsort* algorithm. It should be noted that the effect of the *Binsort* algorithm on the number of messages transmitted is negligible, because it does not change the relationship between donor and donee processors.

5 Experimental Results and Concluding Remarks

The *Binpack* and *Binsort* algorithms have recently been developed and incorporated into the CHAOS runtime support library, which aims at parallelizing highly adaptive problems on distributed memory parallel computers. In this paper, the experiment was restricted to applying the CHAOS runtime library to the structure and dynamics of 2-dimensional flames in lean methane-air mixtures. The effects of radiative loss from water and carbon-dioxide have been included. A moderately detailed reaction mechanism involving 16 species and 35 reactions has also been included for methane combustion. Although experiments have been conducted for problems of several different sizes on the Caltech CCSF Paragon distributed memory parallel computer, we present here only the results from a 256×256 grid due to space limitation.

Figure 3(a) shows the computation times to process 10 time steps. The curves `Chemistry`, `Diffusion`, `Viscosity` and `Convection` represent the performance scaling of separate physical processes on varying numbers of processors. While even the pointwise `Chemistry` process scales reasonably well, it clearly shows that the poor scaling of the overall performance, especially with the large number of processors, arises mainly from the `Convection` process. This is due to the large amount of communication required both in the explicit portion and in the elliptic solution of the pressure equation using multigrid. The current multigrid solution uses an iterative method for the coarse grid solution, which accounts for the majority of the communication. We are currently examining several alternatives to the coarse grid solution, including a direct solution on a single processor.

As shown in Figure 3(b), we have observed up to about 30 percent performance improvement, in terms of elapsed time, in the `Chemistry` process by using the CHAOS runtime library and an irregular partitioning, as compared with using a simple block partitioning. With 128 Intel Paragon processors, the performance improvement in the `Chemistry` process alone leads to a 12 percent overall performance improvement. Although the overall performance improvement is less with 256 processors due to increased overhead, it has also been observed that the total runtime on the 256-node Paragon is about half that of a single processor Cray C90 (where on the Cray 10 time steps take about 58 seconds).

Acknowledgments

Access to the Caltech CCSF Paragon was provided by the Center for Research on Parallel Computation.

References

[1] G. Agrawal, A. Sussman, and J. Saltz, *Compiler and runtime support for structured and block structured applications*, in Proceedings Supercomputing '93, IEEE Computer Society Press, Nov. 1993, pp. 578–587. An extended version available as University of Maryland Technical Report CS-TR-3052 and UMIACS-TR-93-29.

[2] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang, *Communication optimizations for irregular scientific computations on distributed memory architectures*, Tech. Rep. CS-TR-3163 and UMIACS-TR-93-109, University of Maryland, Department of Computer Science and UMIACS, Oct. 1993. To appear in Journal of Parallel and Distributed Computing.

[3] C. C. Douglas, *Implementing abstract multigrid or multilevel methods*, in Sixth Copper Mountain Conference on Multigrid Methods, N. D. Melson, T. A. Manteuffel, and S. F. McCormick, eds., vol. CP 3224, Hampton, VA, 1993, NASA, pp. 127–141.

[4] M. Gerndt, *Updating distributed variables in local computations*, Concurrency: Practice and Experience, 2 (1990), pp. 171–193.

[5] G. Patnaik, K. J. Laskey, K. Kailasanath, E. S. Oran, and T. A. Brun, *FLIC — a detailed, two-dimensional flame model*, Memorandum Report 6555, Naval Research Laboratory, Sept. 1989.