Experience with Graph Scheduling for Mapping Irregular Scientific Computation*

Apostolos Gerasoulis, Jia Jiao
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
{gerasoulis, jjiao}@cs.rutgers.edu

Tao Yang
Department of Computer Science
University of California
Santa Barbara, CA 93106
tyang@cs.ucsb.edu

Abstract

Parallelizing irregular scientific computation efficiently is a challenging open research area. In this paper we investigate the applicability of graph scheduling techniques in solving irregular problems on distributed memory machines. Our approach is to express irregular computation in terms of a macro-dataflow task model and use an automatic scheduling system to map task graphs and also generate parallel code based on the scheduling result. Our preliminary experiments in nCUBE-2 have produced promising results with regards to the practicality of scheduling.

1 Introduction

The need to solve irregular problems, where the patterns of computation and communication are unstructured and/or changing dynamically, arises in many scientific computation applications. Automatic scheduling techniques are useful in exploring parallelism of unstructured computation. Self scheduling and randomized load balancing techniques have been proposed for mapping dynamic independent and tree-structured tasks [1, 18]. For iterative irregular computation, the CHAOS system [21] uses the inspector and executor approach and recently there has been an attempt to extend HPF compilers to handle irregular computation [19]. In this paper, we discuss how static graph scheduling techniques can guide the mapping of irregular computations on multiprocessors and effectively utilize parallelism hidden in complex data dependence structures.

Task scheduling theory and algorithms have been of long-standing research interest because of the promise that such approach could better utilize the power of multicomputers. Finding optimal schedules is difficult, and many scheduling heuristic algorithms have been developed, [8, 11, 15, 22, 23, 2, 17]. Furthermore, scheduling systems that predict the performance of task graph computation have been built [8, 23, 24]. An application of scheduling is in the sparse matrix computation area [4, 3, 20].

In section 2, we will first briefly discuss task graph scheduling techniques and the execution of a schedule. Static scheduling relies on task and communication edge weights, but there is always a discrepancy between static weight estimation and actual run-time performance. Thus it is of interest to know that small estimation errors will not dramatically alter the run-time performance behavior. In section 3, we will provide an analysis on the discrepancy between the static performance predicted by the scheduler at compile-time and the actual run-time performance. In section 4, we present our experience with graph scheduling for irregular N-body simulations.

*The work presented here was in part supported by ARPA contract DABT-63-93-C-0064 under "Hypercomputing and Design" project, by the Office of Naval Research under grant N00014-93-1-0944, by NSF RIA CCR-9409695. The content of the information herein does not necessarily reflect the position of the government and official endorsement should not be inferred.

2 Task computation model and scheduling

A task graph is a directed acyclic task graph (DAG) defined by a tuple $G = (V, E, \mathcal{C}, \mathcal{T})$ where V is the set of task nodes and $v = |V|$ is the number of nodes, E is the set of communication edges and $e = |E|$ is the number of edges, $\mathcal{C}$ is the set of edge communication costs and $\mathcal{T}$ is the set of node computation costs. The value $c_{i,j} \in \mathcal{C}$ is the communication cost incurred along the edge $e_{i,j} = (n_i, n_j) \in E$, which is zero if both nodes are mapped in the same processor. The value $\tau_i \in \mathcal{T}$ is the execution time of node $n_i \in V$. $PRED(n_x)$ is the set of immediate predecessors of n_x and $SUCC(n_x)$ is the set of immediate successors of n_x. We use the *macro-dataflow task model* of execution. A task receives all input in parallel before starting execution, executes to completion without interruption, and immediately sends the output to all successor tasks in parallel.

Scheduling parallel tasks with precedence relations over distributed memory multiprocessors has been found to be much more difficult than the classical scheduling problem [5]. This is because data transferring between processors requires substantial transmission delays. Without imposing a limitation on the number of processors, the scheduling problem that ignores communication overhead is solvable in polynomial time. When communication overhead is present, however, the problem becomes NP-complete [22]. Several scheduling algorithms have been proposed in the literature, e.g. [2, 8, 16, 15, 22, 23]. The algorithm needs to take into account the following tradeoff: Placing two tasks in different processors *parallelizes* the computation process but also could induce higher communication cost. On the other hand, placing tasks in one processor eliminates communication cost but also *sequentializes* the computation process.

We have developed efficient scheduling algorithms and a system called PYRROS [24] that integrates scheduling with code generation techniques. The input to PYRROS is a weighted task graph and the associated sequential C or Fortran code. The output is a static schedule and parallel code for a given architecture. PYRROS performs scheduling and code generation as follows:

1) **Clustering.** The goal of clustering is to identify "useful" parallelism and eliminate unnecessary communication. PYRROS uses the Dominant Sequence Algorithm (DSC) and we have shown that DSC performs well for general DAGs and also produces the optimal solutions for fork, join and coarse grain tree DAGs, [26].

2) **Mapping clusters to processors.** After finding the clusters, PYRROS uses a variation of *work profiling method* [9] for cluster merging in order to balance the load of computation. Then PYRROS uses a heuristic algorithm to map p virtual clusters onto p physical processors of a given interconnection architecture so that the overall communication volume is minimized, [12].

3) **Task ordering.** Once the physical mapping is fixed, interprocessor communication delay between tasks can be determined accurately. We need to order the execution of tasks within each processor Finding a task ordering that minimizes the parallel time is NP-hard. We have proposed a heuristic algorithm RCP and have shown that RCP is optimal for fork and join DAGs, [25].

The total complexity of our scheduling stages is $O((v + e)\log v + p^3)$. In [12], we have compared this multi-stage method with the one-stage ETF method [14] using randomly generated graphs, and found that our algorithm has lower complexity and produces competitive or even better solutions.

3 Executing static schedules

Once a schedule is determined, this schedule must be executed on a real architecture. For message-passing architectures, the basic model of PYRROS in executing a static schedule at processor i is described in Fig. 1.

Parallel machines are complex systems and accurate estimation of communication and computation weight at compile time is very difficult. Therefore, it is important to know the sensitivity of the schedule executor on inaccurate weight estimation and also other run time weight variations.

We now address the sensitivity of the scheduling

Tasklist = ordered tasks assigned to processor i
based on the static schedule.
While Tasklist is not empty **Do**
Remove the first task T_x in the list,

Examine the incoming edges of this task T_x.
If data items it needs are not in the local memory,
issue a message-receiving to fetch desired
data items from the communication buffer.

Perform the task computation.

Send out the produced data items to
the processors that need these data items.
ENDWHILE

Figure 1: The run-time execution of a static schedule.

performance on the variation of task communication and computation weights. Let G be the graph for which the scheduling algorithm produces a schedule S. At run-time, the weights of this graph change, we call the new graph as G^r. We examine the performance of schedule S when used to execute G^r.

In [10], we define the grain of a task T_x as

$$g_x = min(\frac{\min_{T_k \in PRED(T_x)} \tau_k}{\max_{T_k \in PRED(T_x)} c_{k,x}}, \frac{\min_{T_k \in SUCC(T_x)} \tau_k}{\max_{T_k \in SUCC(T_x)} c_{x,k}})$$

and the *granularity* of a DAG G as

$$g(G) = \min_{x=1:v} \{g_x\}.$$

We assume that at run-time, each communication weight $c_{i,j}$ is changed to $c^r_{i,j}$ and the computation weight to τ^r_i. They satisfy

$$(1-f^c_1)c_{i,j} \le c^r_{i,j} \le (1+f^c_2)c_{i,j}, \qquad (1-f^\tau_1)\tau_i \le \tau^r_i \le (1+f^\tau_2)\tau_i$$

where $f_1 = max(f^c_1, f^\tau_1)$ and $f_2 = max(f^c_2, f^\tau_2)$. The lower and upper bounds in the inequalities above are chosen to be sharp.

Let the length of the schedule S for G be $PT(G)$. The run-time method in Fig. 1 uses the mapping and task ordering of S to execute graph G^r, producing a parallel time $PT(G^r)$.

Let us assume that $PT(G) \le B \times PT_{opt}(G)$ where $PT_{opt}(G)$ is the optimal schedule length for G where the minimum is taken over all possible schedules S. Similarly, let $PT_{opt}(G^r)$ be the optimal schedule length for G^r. If B is close to 1 then the scheduler, say for example the PYRROS static scheduler, produces close to the optimum schedules. What happens to the actual run-time execution of this schedule is described by the following property:

Property 1 Assume that $PT(G,S) \le B \times PT_{opt}(G)$, then

$$\frac{PT(G^r)}{PT_{opt}(G^r)} \le min\ (\ F_1,\ F_2(1+\frac{1}{g(G^r)})(1+\frac{1}{g(G)})\ )B$$

where $F_1 = \frac{1+f_2}{1-f_1}$, $F_2 = \frac{1+f^r_2}{1-f^r_1}$.

Proof: Given a schedule S of G which defines the mapping and ordering of task to processors. By adding an edge from T_x to T_y if T_x is executed immediately before T_y in the same processor and T_x and T_y are independent, then we obtain the execution graph G_s for G under schedule S. The graph structure of G_s is identical to G^r_s, the graphs differing only in the values for communication and computation weights. We set $PT(G^r)$, the length of the run-time schedule for the method in Figure 1 when using schedule S. We have that

$$PT(G^r) = \sum_{CP(G^r_s)} (\tau^r_i + c^r_{i,i+1}) \le (1+f_2) \sum_{P(G_s)} (\tau_i + c_{i,i+1})$$

$$\le (1+f_2)PT(G).$$

where $CP(G^r_s)$ is the critical path in G^r_s and $P(G_s)$ is the same path in G_s which might not be a critical path in G_s because of the change in the weights. The summations are taken over all nodes in the corresponding paths in the graph.

Now, let $PT_{opt}(G^r)$ be the length of the optimal schedule $S = ropt$ for G^r. Let $PT^*(G)$, be the parallel time of this optimum schedule applied to G. We have that

$$(1-f_1)PT_{opt}(G) \le (1-f_1)PT^*(G) =$$

$$\sum_{CP(G_{ropt})} ((1-f_1)\tau_i + (1-f_1)c_{i,i+1}) \le$$

$$\le \sum_{P(G^r_{ropt})} (\tau^r_i + c^r_{i,i+1}) \le \sum_{CP(G^r_{ropt})} (\tau^r_i + c^r_{i,i+1}) = PT_{opt}(G^r_{ropt}).$$

The proof of the first bound is obvious.

To prove the second bound we use the definition of granularity to derive another upper bound for run-time communication with respect to computation: $c^r_{i,i+1} \le \frac{\tau^r_i}{g(G^r)}$. We have that

$$PT(G^r) = \sum_{CP(G^r_s)} (\tau^r_i + c^r_{i,i+1}) \le (1+\frac{1}{g(G^r)}) \sum_{CP(G^r_s)} \tau^r_i \le$$

$$\leq (1+f_2^r)(1+\frac{1}{g(G^r)}) \sum_{P(G_s)} (\tau_i + c_{i,i+1}) \leq$$

$$\leq (1+f_2^r)(1+\frac{1}{g(G^r)}) \sum_{CP(G_s)} (\tau_i + c_{i,i+1}) =$$

$$(1+f_2^r)(1+\frac{1}{g(G^r)}) PT(G).$$

For the lower bound assume again that the schedule $S = ropt$ is optimum for G^r and $PT^*(G)$ is the corresponding parallel time when $ropt$ is applied to G. Then we use the facts that $PT_{opt}(G) \leq PT^*(G)$ and $c_{i,i+1} \leq \frac{\tau_i}{g(G)}$ to show that

$$(1-f_1^r)\frac{PT_{opt}(G)}{1+\frac{1}{g(G)}} \leq (1-f_1^r) \sum_{CP(G_{ropt})} \frac{\tau_i + c_{i,i+1}}{1+\frac{1}{g(G)}}$$

$$\leq \sum_{P(G^r_{ropt})} \tau_i^r \leq \sum_{P(G^r_{ropt})} \tau_i^r + c^r_{i,i+1} \leq \sum_{CP(G^r_{ropt})} \tau_i^r + c^r_{i,i+1}$$

$$= PT_{opt}(G^r).$$

Combining the lower and upper bounds we prove the property. □

This property implies that the run-time performance depends on how good the compile time schedule is and how small the variations of the weights at run-time are. It should be mentioned that for coarse grain task graphs it is easier to produce good schedules. As a matter of fact for a completely connected architecture with unbounded number of processors we can always produce a schedule, see [10], such that

$$\frac{PT(G)}{PT_{opt}(G)} \leq 1 + \frac{1}{g(G)}.$$

This implies that $B = 1 + 1/g(G)$ tends to 1 as the grain of graph becomes coarser. Also for coarse grain task graphs we can estimate the computation and communication weights much more accurately as long as there is no resource contention.

We have an experimental verification of the above analysis. In section 4, we consider the execution of static schedules for irregular N-body graphs. For a test case with 10000 particles, we schedule and execute the computation, and record the actual parallel time in nCUBE-2. After that we perturb the task and computation weights randomly so that the schedule is produced using inaccurate weights. We then compare the run-time performance of the schedule using accurate weights with the one using inaccurate weights, recording the slow-down percentage. The ratio of weight

processors	slow-down(%)
p=8	0.4%
p=16	5.1%
p=32	7.4%
p=64	6.3%

Table 1: The increase in parallel time after random weight perturbation of the N-body simulation on nCUBE-2.

perturbation is between -20% and +20% and the performance degradation in terms of parallel time due to perturbation is listed in Table 1. We observe that performance variation is virtually none for small number of processors and very modest for larger number of processors. We also have verified other cases and the experiments give a strong evidence that a static scheduling is insensitive to weight perturbation as long as the perturbation is not excessive.

4 Scheduling and executing N-body simulations

N-body simulations are required in many scientific computation applications. Given a distribution of n particles in space, and a rule governing their interactions, e.g. gravitational force, the problem is to compute the movement of the particles. This movement can be discretized in time steps, during each of which we compute the force applied on each particle, and update the particle positions. For example, in 2-D space, the force applied to a particle located at position z_j, represented by a complex number, is:

$$F_j = \sum_{i=1, i\neq j}^{n} \frac{A_i}{z_i - z_j}$$

where A_i is the charge of particle i located at z_i.

We use PYRROS to map the 2-D Fast Multipole Method (FMM). The basic idea of the algorithm is to subdivide the space into boxes. The evaluation of particle interactions between far-away boxes can be approximated by series of expansions. The coefficients of the series are computed once and used by all "far-away" boxes. Depending on the input distribution, the algorithm comes in two flavors:

(1) Non-adaptive: all boxes are of the same size. This is applied to uniform distributions.

(2) Adaptive: boxes are subdivided further in the regions of higher particle density. This is used for highly-non-uniform distributions. For details see Greengard's thesis, [13].

We consider both non-adaptive and adaptive FMM algorithms. To introduce the method we consider the non-adaptive algorithm first.

4.1 Scheduling for the non-adaptive FMM algorithm

Graph Structure: Figure 2 is a 2-level regular subdivision and its corresponding quadtree. The structure of this tree is regular but the task weight distribution is irregular.

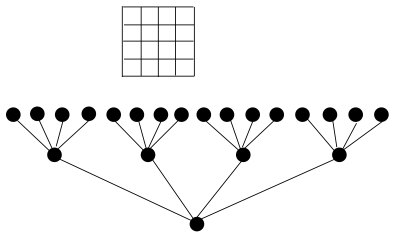

Figure 2: A 2-level non-adaptive subdivision

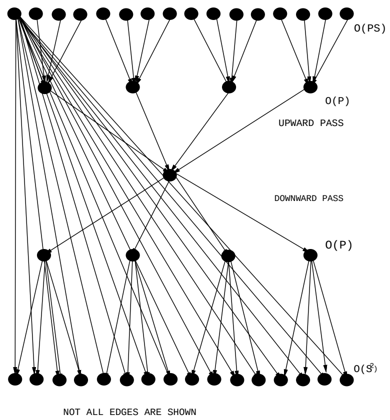

Figure 3: Task graph for a 2-level subdivision. The order of task weights are also shown. Only communication from one leave of the upward pass tree to the leaves of the downward pass tree is shown. The others are deleted for simplicity of presentation.

The basic structure of the task graph is a concatenation of this in-tree with an out-tree which is its reverse. Because of the regular subdivisions, the tree is a complete quad-tree. The leaves of the tree correspond to the boxes at the finest level, and the internal nodes correspond to the boxes at higher levels. Following the thesis of Greengard, we call the in-tree "upward pass", and the out-tree "downward pass." There are additional edges between the two parts of the graph, which represent interactions between boxes. Figure 3 is an example of the task graph for a 2-level subdivision. For clarity we only draw the communication edges for one leaf node.

Note that although in this graph, every leaf communicates with all the other leaves, this is not true for divisions of higher levels. A leaf communicates with its neighbors and "interaction list", which is essentially the list of all second near neighbors, see figure 4 . The number of boxes in each of the lists is bounded by a constant, so the graph has $O(4^l)$ edges, l being the subdivision level. It's not easy to draw these graphs clearly, but from the following description it should be easy to understand their structure.

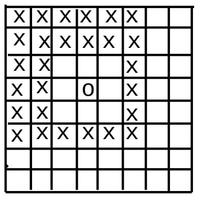

Figure 4: A 3-level non-adaptive subdivision. X are the boxes in the interaction list of box O. Any box has most 27 boxes in its interaction list, and at most 8 neighbors.

There are four kinds of nodes in the graph: leaf nodes in the upward pass, non-leaf nodes in the upward pass, non-leaf nodes in the downward pass, and leaf nodes in the downward pass. Each group of nodes has similar functionality. We are not giving the details of the functions they perform here. Roughly speaking, the multipole expansions are generated in the upward leaf nodes, sent up toward the tree root, and "translated" to the non-leaf nodes in the downward pass which lie in the interaction list. The expansions are then converted into "local expansions" and sent down the downward tree, and the leaf nodes in the downward tree perform the major computation, see [13] for more details.

proc	Time(ms)	PYRROS-run	PYRROS-compile
p=4	320017	3.96	3.98
p=8	160320	7.90	7.95
p=16	81258	15.6	15.7
p=32	41638	30.4	31.0
p=64	21463	59.0	61.0

Table 2: The speedup of the non-adaptive FMM on nCUBE-2s. PYRROS-run is the real speed up, while PYRROS compile is the prediction of the scheduler.

The weights of the tasks depend on the input and are irregular, so we only describe the order of the weights in terms of number of particles in a leaf box s and number of terms in the expansion p. For simplicity we use s as upper bound to the number of particles in each leaf box. number would vary from box to box but not too much. The weights are $O(p * s)$ for leaf nodes in the upward pass, $O(s^2)$ for the leaf nodes in downward pass, and $O(p)$ for all non-leaf nodes.

Granularity: The graph is fine grain near the root but coarse grain at the leaf level where high degree of parallelism exists. So the speedup is expected to be very high.

Performance results: We use an example of 80000 randomly distribution particles, with a 5-level (32*32) subdivision. This graph has about 2700 nodes and 30000 edges. PYRROS scheduler can handle graph with such size quite rapidly. The speedup for the initial iteration is almost linear, shown in Table 2. We will discuss the issue of the iterative execution later.

The largest problem we were able to run on a 16MB memory per processor, 128-processor nCUBE-2s machine is a problem with $8 * 10^5$ particles, demonstrating that our system is capable of handling large real world problems.

4.2 Scheduling for the adaptive FMM algorithm

Although the non-adaptive algorithm has very good performance for uniform distributions of particles, it cannot handle highly non-uniform distributions encountered in some applications, such as galaxy simulations. Adaptive subdivision must be used for such distributions. Figure 5 illustrates an example of an adaptive subdivision and the corresponding unbalanced tree for this partition. The real task graph consists of the concatenation of this in-tree with the symmetric out-tree and edges between the two parts.

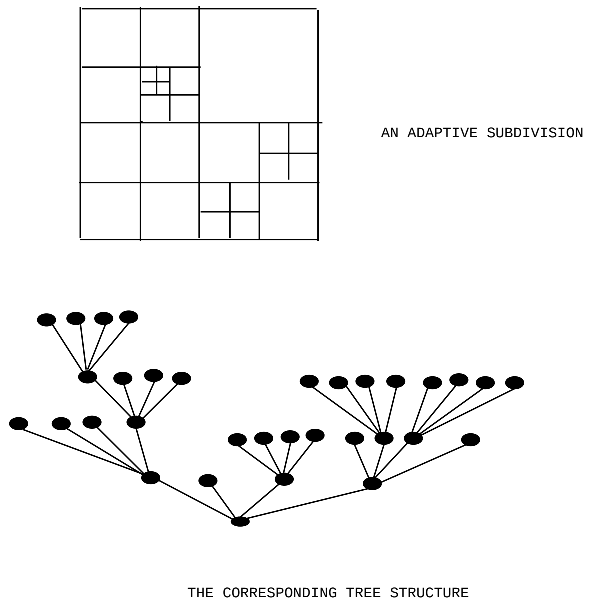

Figure 5: An adaptive subdivision

Graph structure: The task graph structure is similar to that of the non-adaptive case in the sense that it still has an upward pass (in-tree) and a downward pass (out-tree). But the tree is unbalanced, and the height is larger for the places with higher particle density. So the tree structure varies, depending on the input distribution. For galaxy distributions, the tree is highly unstructured.

The function of the nodes are basically the same as in the non-adaptive algorithm, with the following complications (see pp.28-38 in Greengard's thesis for detailed explanations.) :

(a.) Neighboring leaf boxes could be of different sizes because they may not be in the same level of the tree.
(b.) Since multipole expansions cannot be translated between boxes of different sizes, we need to deal with "far-away" interactions between boxes in different levels of the tree in some special way.

With these extra cases in mind, a task graph can be

#proc	Time(ms)	PYRROS-run	PYRROS-compile
p=4	105542	3.94	3.97
p=8	53663	7.76	7.83
p=16	28270	14.7	15.1
p=32	16648	25.0	28.0
p=64	10458	40.0	45.4

Table 3: The speedup of the adaptive FMM on nCUBE-2s.

generated by first creating different lists for each box. These lists are classified as neighbors, far-away boxes of the same size, smaller size, and bigger size. We then generate tasks and communication edges using these lists.

Performance results: Due to the irregular structure, it is necessary to use an automatic scheduling tool to schedule such task graphs. Our example is a non-uniform galaxy distribution with 40000 particles. This distribution results in a highly unbalanced tree, with the maximum depth of the leaves being as large as 9, and the minimum depth of the leaves being only 1. Such kind of distribution is very difficult to parallelize. Nevertheless, PYRROS is able to achieve good performance with speedup of 40 using 64 processors, as shown in Table 3.

4.3 Static scheduling performance for iterative execution

The N-body problem is a dynamic problem, in the sense that particles are moving, and their positions are updated during each time step. So they could move from one box to another, resulting in changes in the task graph. Notice that in the non-adaptive algorithm, only task weights are changing due to particle movement, However in the adaptive case, the task graph structure could also be changing if the space is repartitioned .

Here we investigate the effect of iterative execution on the performance if we reuse the initial schedule. We believe that a static scheduling tool could be useful even for such dynamic applications. This is because the movement of particles is small during each time step. Even if the particles have high velocity, the time step δt has to be chosen small enough to ensure stability. Since PYRROS scheduler for coarse grain graphs does not need precise weight information, we expect that small perturbations should not lead to significant deterioration in performance, as we saw in section 3. Thus a schedule can be reused for many steps, until its performance deteriorates to the point where rescheduling becomes beneficial. The interesting question is how frequent we need to reschedule, since this will determine the overall performance of PYRROS for the N-body problem. We investigate this next.

processors	slow-down(%)
p=8	4.0
p=16	4.9
p=32	6.1
p=64	7.5

Table 4: The increase in parallel time after 100 iterations.

Using a uniform distribution of 10000 particles, we choose $\delta t = 0.001$ which for this case is large enough to get sizable movement of particles in each step, but small enough to ensure stability. Our experiment runs iteratively on 8-64 processors. The same initial schedule is used for all iterations. We calculate the deterioration of the performance: i.e. $\frac{PT_{100}-PT_1}{PT_1}$, where PT_1 is the parallel time for the first iteration and PT_{100} is the parallel time for the 100th iteration. We call this value the "slow-down". The results are shown in Table 4.

Note that the slow-down in parallel time is only 7.5% for 64 processors. Of course, these data are only for this particular uniform distribution. But in general, if the change of the task graph happens slowly and gradually, we could say that an initial schedule can be reused for a substantial number of steps with reasonable performance. Combined with runtime reschedule to correct the imbalance when it gets too large, this approach could be useful in many real world applications, not just N-body. We plan to investigate this approach further.

5 Conclusions

We have presented our approach to N-body simulations using automatic mapping techniques. We have provided an analysis on the run-time execution performance of static schedule. Our experiments show that the automatically scheduled task computation could attain a good performance for irregular problems provided that sufficient task parallelism is available. We have also experimented with sparse matrix factorization. However the speedups achieved were moderate mainly due to insufficient parallelism in the partitioning. On the other hand, Chong et al. [4] and Chong and Schreiber [3] have reported very good speedups for DSC scheduling algorithm of PYRROS, when used to solve sparse triangular systems. They have developed special communication libraries on CM-5 that resulted in a reduction of communication cost and an increase in granularity.

References

[1] S. Chakrabarti, A. Ranade, and K. Yelick, Randomized load balancing for tree-structured computation. *Proc. of 1994 Scalable High-Performance Computing Conference*, 666-673.

[2] Y-C Chung and S. Ranka, Applications and performance analysis of a compile-time optimization application for list scheduling algorithms on distributed memory multiprocessors, *Supercomputing 92*, pp. 512-521.

[3] F. T. Chong and Robert Schreiber, Parallel sparse triangular solution with partitioned inverses and prescheduled DAGs, Tech Report, MIT, 1994.

[4] F. T. Chong, Shamik D. Sharma, Eric A. Brewer and Joel Saltz, Multiprocessor Runtime Support for Fine-Grained Irregular DAGs, To appear in *Toward Teraflop Computing and New Grand Challenge Applications.* Eds. Rajiv K. Kalia and Priya Vashishta. Nova Science Publishers, Inc. New York. 1995.

[5] E.G. Coffman and R.L. Graham, Optimal scheduling for two-processors systems, *Acta Informatica*, 3 (1972), 200-213.

[6] J.J. Dongarra and D. C. Sorensen, SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs, in *The Characteristics of Parallel Algorithms,* D.B. Gannon, L.H. Jamieson and R.J. Douglass (Eds), MIT Press, 1987, pp363-394.

[7] T.H. Dunigan, Performance of the INTEL iPSC/860 and nCUBE 6400 hypercube, ORNL/TM-11790, Oak Ridge National Lab., TN, 1991.

[8] H. El-Rewini and T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, *J. of Parallel and Distributed Computing*, 9(1990), 138-153.

[9] A. George, , M.T. Heath, and J. Liu, Parallel Cholesky factorization on a shared memory processor, *Lin. Algebra Appl.,* 77(1986), pp. 165-187.

[10] A. Gerasoulis and T. Yang, On the granularity and clustering of directed acyclic task graphs, *IEEE Trans. on Parallel and Distributed Systems.*, Vol 4, No 6, June 1993 pp. 686-701.

[11] A. Gerasoulis and T. Yang, A comparison of clustering heuristics for scheduling DAGs on multiprocessors, *J. of Distributed and Parallel Computing,* Vol. 16, No. 4, pp. 276-291 (Dec. 1992).

[12] A. Gerasoulis, J. Jiao, and T. Yang, A multistage approach to scheduling task graphs. To appear in DIMACS Book Series on Parallel Processing of Discrete Optimization Problems. AMS publisher. Edited by P.M. Pardalos, K.G. Ramakrishnan, and M.G.C. Resende.

[13] Leslie Greengard, The Rapid Evaluation of Potential Fields in Particle Systems Ph.D thesis, Yale University, 1987.

[14] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times, *SIAM J. Comput.,* pp. 244-257, 1989.

[15] S.J. Kim and J.C Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, *Proc. of Int'l Conf. on Parallel Processing*, vol 3, pp. 1-8, 1988.

[16] C.L. McCreary and D.H. Gill, Automatic determination of grain size for efficient parallel processing, *Communications of ACM,* 1073-1078, Sept., 1989.

[17] S. Pande, D. Agrawal and Jon Mauney Compiling functional parallelism on distributed-memory systems. *IEEE Parallel and Distributed Technology: Systems & Applications,* Spring 1994, pp.64-76.

[18] C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, 1988.

[19] R. Ponnusamy, Y.-S. Hwang, J. Saltz, A. Choudhary, G. Fox. Supporting Irregular Distributions in FORTRAN 90D/HPF Compilers, Univ. of Maryland, CS-TR-3268, 1994.

[20] R. Pozo, Performance modeling of sparse matrix methods for distributed memory architectures, in *Lecture Notes in Computer Science, No. 634,* , Springer-Varlag, 1992, 677-688.

[21] Saltz, J., Crowley, K., Mirchandaney, R. and Berryman,H., Run-time scheduling and execution of loops on message passing machines, *J. of Parallel and Distributed Computing,* Vol. 8, 1990, pp. 303-312.

[22] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, The MIT Press, 1989.

[23] M. Y. Wu and D. Gajski, Hypertool: A programming aid for message-passing systems, *IEEE Trans. on Parallel and Distributed Systems,* vol. 1, no. 3, pp.330-343, 1990.

[24] T. Yang and A. Gerasoulis, PYRROS: Static task scheduling and code generation for message-passing multiprocessors, *Proc. of 6th ACM Inter. Conf. on Supercomputing,* Washington D.C., July, 1992, pp. 428-437.

[25] T. Yang and A. Gerasoulis, List scheduling with and without communication delay, *Parallel Computing*, Vol 19, 1993, pp. 1321-1344.

[26] T. Yang and A. Gerasoulis, DSC: Scheduling parallel tasks on an unbounded number of processors, *IEEE Trans. on Parallel and Distributed Systems.*, Vol. 5, No. 9, 951-967, 1994.