```
            *((int  * *) arglist)++ = (& *sum_right) ;
            arg_list_ptrs[my_index] = arglist; /* Setup for sum(a,midpoint+1,r,&sum_right) */
            poolStack_base[my_index] = teamStack_top[my_index]
             = poolStack_top[my_index] = taskPointer; /* Create task description */
            factor(taskPointerT); /* Setup for team-dividing */
            my_pending_no++;
            issueTask(taskPointerT); /* Issue tasks and join. Team-Cooperation may take place */
            my_pending_no--;
     }
     my_endpoint = input_endpoint;
     (* result) = (*sum_left + *sum_right);
     my_local_heap = fpoint;
     }
my_local_heap = fpoint;
}
```

```
        *((int  * *) arglist)++ = (& *sum_right) ;
        arg_list_ptrs[my_index] = arglist; /* Setup for sum(a,midpoint+1,r,&sum_right) */
        if ( my_task < my_task_top ) {
                UsLock(taskStack_lock[my_index], 0);
                if (((poolStack_top[my_index] - poolStack_base[my_index])
                    < my_pending_no)
                  || (poolStack_base[my_index]
                    == teamStack_top[my_index])) { /* Enough Task Pool space */
                   *my_current_pending++;
                   my_task->func = PP_sum ;
                   my_task->stat = WAIT;
                   my_task->startProc = my_index;
                   my_task->PARA = para1;
                   my_task->next = NULL; /* Create the waiting task description */
                   if ( poolStack_base[my_index] == teamStack_top[my_index] ) {
                        poolStack_base[my_index] = poolStack_top[my_index]
                        = my_task++;
                   }else { poolStack_top[my_index] = my_task++;
                   }
                   UsUnlock(taskStack_lock[my_index]); /* Setup one task waiting on the pool */
                   PP_sum(para); /* Execute the another task on current processor */
                   if ( POOL_CHECK() == 1 ) { /* Check task pool */
                        PP_sum(para1); arg_list_ptrs[my_index] = para1;
                            /* Take the unstolen task back to current processor */
                   } else {
                        POOL_WAITING(); /* Wait for joining. Team-Cooperation may take place */
                   }
                } else { /* Task-Pool is full. Execute both tasks on current processor */
                   UsUnlock(taskStack_lock[my_index]);
                   PP_sum(para);
                   arg_list_ptrs[my_index] = para;
                   PP_sum(para1);
                   arg_list_ptrs[my_index] = para1;
                }
        } else { fprintf(&_iob[2], "stack overflow \n");
                 exit(3);
        }
} else { /* More than one member in a team */
        taskPtr taskPointer, taskPointerT;
        int *arglist; int *temp;
        taskPointerT = taskPointer =  my_task++;
        taskPointer->func = PP_sum ;
        taskPointer->weight = 1 ;
        arglist = taskPointer->PARA = arg_list_ptrs[my_index];
        taskPointer->stat = WAIT;
        taskPointer->next = my_task;
        temp = arglist;
        *((int  * *) arglist)++ = a ;
        *((int *) arglist)++ = l ;
        *((int *) arglist)++ = midpoint ;
        *((int  * *) arglist)++ = (& *sum_left) ;
        arg_list_ptrs[my_index] = arglist; /* Setup for sum(a,l,midpoint, &sum_left) */
        taskPointer =  my_task++;
        taskPointer->func = PP_sum ;
        taskPointer->weight = 1 ;
        arglist = taskPointer->PARA = arg_list_ptrs[my_index];
        taskPointer->stat = WAIT;
        taskPointer->next = NULL;
        *((int  * *) arglist)++ = a ;
        *((int *) arglist)++ = (midpoint + 1) ;
        *((int *) arglist)++ = r ;
```

A A Brief Description of the SELSYN Implementation

SELSYN-C has been implemented with two integrated parts, the source-to-source compiler and the run-time environment. The source-to-source compiler takes the input source code which is written in SELSYN-C to construct an internal AST (Abstract Syntax Tree). Various transformations that include the addition of synchronization are carried out on this AST. Based on the transformed AST, different output source codes can be generated for a variety of target machines. The output source code can be processed by the compiler of the target machine. Linked with the run-time environment implemented on the target machine, and the resulting object code can run directly on that machine.

We present here one sample output source code. The input source code is the example which we presented in section 2.2 in Figure 3. The source-to-source compiler generated the following output source code for the target machine BBN Butterfly GP-1000. The output source code includes the calls to the run-time enironment. Via these links, shared variable declarations, task issuing, task scheduling and processor assignment is supported and carried out. The code presented here shows that by using the simple SELSYN-C, programmer is spared all of the complexity that is involved with a complicated and efficent scheduling mechanism.

```
int PP_sum (PARA)
int *PARA; /* Modified function name, so that it can accept shared variables passing. */
  {int *fpoint = my_local_heap;
   int  * a  = *((int  * *) PARA)++;
   int l  = *((int *) PARA)++;
   int r  = *((int *) PARA)++;
   int  * result  = *((int  * *) PARA)++; /* Obtain parameters via shared space */
   int *sum_left ; int *sum_right ;
   sum_left = ((int * ) my_local_heap)++;
   sum_right = ((int * ) my_local_heap)++; /* Allocate the memory space for shared variables */
   if ((l == r))
      (* result) = a[l];
   else {
        int *fpoint = my_local_heap; int input_endpoint; int midpoint ;
        midpoint = ((l + r) / 2);
        input_endpoint = my_endpoint;
        /**** Here goes the transformation for parallel function call ****/
        /* sum(a,l,midpoint, &sum_left) // sum(a,midpoint+1,r,&sum_right) */
        if ( my_index == my_endpoint ) { /* Only one member in team. */
                int *arglist; int *para, *para1;
                para1 = arglist = arg_list_ptrs[my_index];
                *((int  * *) arglist)++ = a ;
                *((int *) arglist)++ = l ;
                *((int *) arglist)++ = midpoint ;
                *((int * *) arglist)++ = (& *sum_left) ;
                para = arglist ; /* Setup for sum(a,l,midpoint, &sum_left) */
                *((int  * *) arglist)++ = a ;
                *((int *) arglist)++ = (midpoint + 1) ;
                *((int *) arglist)++ = r ;
```

[MvR87] P. Mehrotra and J. van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, Vol. 5, 1987.

[VR88] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, Vol. 17(No. 4), 1988.

References

[BAD87] E.D. Brooks III, T.S. Axelrod, and G.A. Darmohray. The cerberus multiprocessor simulator. In *Proc. Third SIAM Conference on Parallel Processing*, 1987.

[BBN89] BBN Advanced Computers Inc. *Private Communication*. BBN Hotline, 1989.

[Bro89] Eugene D. Brooks III. PCP: A parallel extension of C that is 99% fat free. In Tom MacDonald, editor, *Workshop on Scientific And Numerical Programming in C, ACM Supercomputing 89*, 1989.

[CGR89] R. F. Cmelik, N. H. Gahani, and W. D. Roome. Experience with multiple processor versions of concurrent C. *IEEE Transaction on Software Engineering*, Vol. 15, March 1989.

[CT90] K. M. Chandy and S. Taylor. Composing parallel programs. In *Beauty is our Business*. New York: Springer-Verlag, 1990.

[DGNP88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, 1988.

[Din92] Weiren Ding. SELSYN-C: A self-synchronizing parallel programming language. Master's thesis, School of Computer Science, McGill University, (expected completion February) 1992.

[FT89] I. Foster and S. Taylor. Strand: New concepts in parallel programming. In Englewood Cliffs, editor, *Strand: New concepts in Parallel Programming*. NJ: Prentice-Hall, 1989.

[GPHL88] Mark D. Guzzi, David A. Padua, Jay P. Hoeflinger, and Duncan H. Lawrie. Cedar FORTRAN and other vector and parallel FORTRAN dialects. Technical report, University of Illinois–Center for Supercomputing Research and Development, January 1988. CSRD Rpt. No. 731, Submitted to 1988 International Conference on Parallel Processing, St. Charles, IL.

[Hal85] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[Hen90] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, January 1990.

[JBAJ89] Harry F. Jordan, Muhammad S. Benten, Gita Alaghband, and Ruediger Jakdo. The force: A highly portable parallel programming language. In *Proceeding of the 1989 International Conference on Parallel Processing*, 1989.

[Jor87] Harry F. Jordan. The force. In Leah H. Jamieson, Dennis Gannon, and Robert J. Douglass, editors, *The Characteristics of Parallel Algorithms*. The MIT Press, 1987.

[KHM89] D. A. Kranz, R. H. Halstead Jr., and E. Mohr. Mul-T: A high-performance parallel lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, pages 81–90, 1989.

[KS85] James T. Kuehn and Howard Jay Siegel. Extensions to the C programming lnaguage for SIMD/MIMD parallelism. *IEEE*, 1985.

[LR91] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Proceeding of the Third ACM/SIGPLAN Symposium on Pronciples and Practice of Parallel Programming*, April 1991.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language reference manual version 1.2. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-146, Rev. 1.

Backing up our language extensions, we developed a self-scheduling mechanism that can achieve high efficiency and good utilization of processors. Our work on scheduling is influenced by the work of WorkCrew [VR88] and PCP [Bro89]. The WorkCrew approach [VR88] introduces a mechanism that handles task divisions properly. Under this model, task subdivisions are queued for execution by any processor which becomes available while the requesting processor is busy. The WorkCrew approach won't divide a task unless there is a processor becomes idle and it will issue subtask on its own processor. Its mechanism is based on fork mechanism and therefore it faces the cost of processor acquisition. Eugene Brooks introduced a split-join model rather than a fork-join model in [Bro89]. In the split-join model the job starts out fully parallel with all of the processors it will ever have, and this team of processors is disassociated into independent subteams as nested concurrency is encountered. The weak points of this system is that it's lack of the ability to balance the work load between independent processors. Our mechanism has the advantage of both these two mechanisms. We reduce the cost by adopting the split-join model (which we call team-division), while at the same time we introduced team-cooperation to rebalance the work distribution. Rather than having a centralized task-pool, we create individual controller for each team so that the cost could be reduced further.

Other related research on task scheduling has been introduced for Strand [FT89] and for Mul-T [KHM89]. Mul-T uses lazy task creation to control the parallelism and try to balance the work distribution. Both approaches adopted central scheduler which high expense can be expected. Our mechanism uses some similar ideas, but implemented in the domain of the popular language C.

6 Conclusions

This paper has reported on the SELSYN-C compiler and associated team-cooperation scheduling mechanisms. The SELSYN-C compiler has been implemented and is currently producing parallel code for a BBN GP-1000 parallel processor.

As discussed in this paper, the design our the SELSYN-C language was driven by the need for an "easy-to-use" language that was accessible to a wide variety of programmers. Our SELSYN-C language consists of simple extensions to C that free the programmer from dealing with architectural details like processor management. Rather, the programmer can concentrate on the application and leave the details of scheduling and resource management to be handled completely by the the SELSYN-C compiler and associated run-time library.

In order to provide an effective and efficient scheduling strategy we have incorporated the notion of weights into the SELSYN-C language, and we have developed a new team-cooperation scheduling mechanism that can make use of these weights. Finally, we have demonstrated the effectiveness of our approach on several divide-and-conquer applications.

Although we have not yet fully explored this possibility, we would also like to explore the performance when the search heuristic is incorporated as weights. The heuristic could assign more processors to the more hopeful search branches and a well chosen heuristic should further improve the performance of the Cooperating-Team approach.

4.3 Summary of Experiments

From the two test cases we presented in this section, we can see the Cooperating-Team approach achieves the the best performance for both the case when weights are unequal and known (as illustrated with quicksort), and the case when weights are unequal and unknown (as illustrated with puzzle). Other experiments which indicate similar results have been performed and are reported in [Din92]. It should be noted that for some cases in which the workload is evenly split, the Even-Team mechanism is superior. In these cases neither the weighted-division nor the team-cooperation improves the performance of the scheduling, since there is an overhead associated with both the Weighted-Team and Cooperating-Team mechanisms.

5 Related Work

The development of parallel programming languages has attracted much attention from researchers. Some have worked on developing entirely new languages for parallelizing programs, such as Strand [FT89], PCN [CT90], SISAL [MSA+85] and BLAZE [MvR87]. Although these are interesting and often elegant languages, they do not satisfy our goal of being familiar to programmers.

There are also many activities which involve developing extensions to existing languages. In this area, one of the major thrusts has been the development of extensions to the language FORTRAN. There are several proposed extensions to this language, which include EPEX/Fortran [DGNP88], Cedar Fortran [GPHL88] and Force [Jor87, JBAJ89]. Several other researchers have focused on developing extensions to language C. Efforts in this direction include Concurrent C [CGR89], Jade [LR91], PCP [Bro89], and parallel-C [KS85]. In the domain of symbolic languages like Lisp there has been effort in designing Mul-T [KHM89] and Multilisp [Hal85].

Rather than developing a new language, we have selected the programming language C as the basis of our work because it is familiar to a wide range of programmers, and thus it provides a large user base for our new language. SELSYN-C is an extension set to C. It provides a very simple way to express the parallelism in imperative language C explicitly. This extension includes the definition of shared variables and parallel function calls. Compared to the other development on the C extensions, our language has the advantage of being easy to use and understand. We have concentrated on minimizing the number of new keywords and operators, and we have introduced application specific extensions that are useful for the compiler (weights).

```
search(root)
{
        if (root->state == goal ) {
                output the solution path;
                return;
        } else {
                if ( satisfy the search bound ) {
                        generate all possible children nodes
                        if ( there is node to be searched ) {
                                search(root->firstchild)
                                // search(root->sibling);
                        }
                }
        }
}
```

Figure 8: Test case 2: Puzzle Search

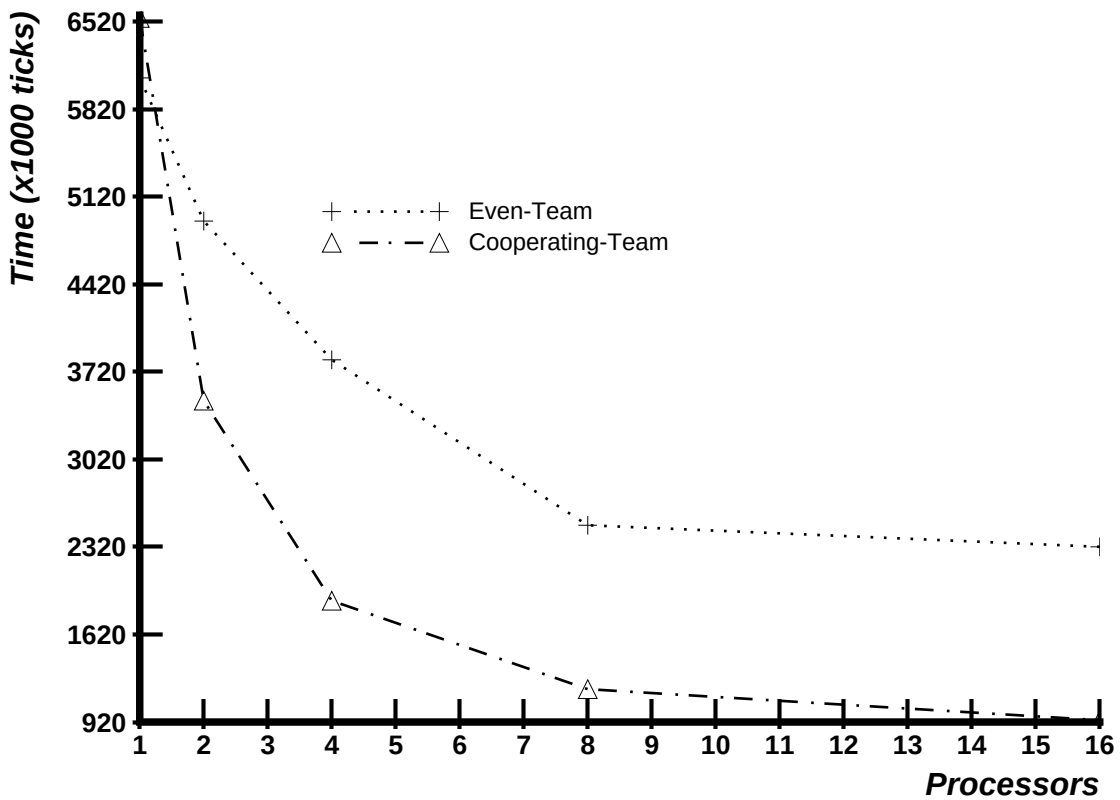

Figure 9: Puzzle Search

	Time (x10^3ticks)	Speedup (compare to Even-Team)
Even-Team	18	1.00
Weighted-Team	15	1.20
Cooperating-Team	11	1.64

Figure 7: Relative performance on six processors

Running the Even-Team on one processor is equivalent to running the sequential program. That is because the Even-Team mechanism immediately resorts to the sequential computation when only one processor is available. However, the Weighted-Team and the Cooperating-Team mechanisms have some amount of overhead, even if only one processor is available. By studying the performance of each mechanism for the one processor case, we can determine the overhead of the Weighted-Team and the Cooperating-Team mechanisms by comparing them with the Even-Team case. Note that even though the overhead is significant, we can see the benefits of the more expensive mechanisms for all cases with more than one processor.[4]

4.2 Example 2: Parallel Search

Searching is a very important technique in the area of artificial intelligence. Since searching involves the creation and manipulation of trees which nicely subdivide into parallel tasks, it can frequently take advantage of multiple processors.

For our test, we selected a 4 by 4 puzzle search application. It finds all possible solutions within some cost bound. This application manipulates a search tree in which each node has at most three children. For our test we represented the tree using an approach ("symmetric lists") that generalizes easily to trees in which each node can have any number of children. Thus, each node in our tree includes pointers to: (1) its parent node, (2) its first child, and (3) its right sibling. Using this representation, the parallel search program can be described as in Figure 8. Note that although the searches can proceed in parallel, we do not have any trivial way of indicating the expected weights, and therefore this is a good test to determine how well our Cooperating-Team mechanism can handle load balancing.

Figure 9 shows the test result by applying the Even-Team mechanism and the Cooperating-Team mechanism to this program.[5] Although both approaches get reasonable performance improvements on up to 8 processors, the Cooperating-Team approach performs significantly better than the Even-Team approach.

[4]Note that we are concerned only with the relative performance of the various scheduling mechanisms. To fully analyze the shape of the performance curves we would need to consider other factors like memory contention and data locality.

[5]Note that since there are no weights, we do not have a Weighted-Team experiment.

Now we can see that part 1 and part 3 have 2 elements and 11 elements that need to be sorted respectively. This unevenness of the partitioning step tests the ability of our mechanisms to deal with an unbalanced work load distribution which is unpredictable at compile-time.

In Figure 6, we present the result of sorting 4096 elements. The x-axis represents the number of processors used for the test, while the y-axis represents execution time in machine ticks (one tick equals to 62.5 microseconds [BBN89].). The three lines give the performance for the Even-Team, Weighted-Team, and Cooperating-Team scheduling mechanisms. For this experiment both the Weighted-Team and Cooperating-Team tests used a parallel function call with weights. In particular, the weight for each recursive call to `quick-sort()` was given as the number of the elements in the input list.

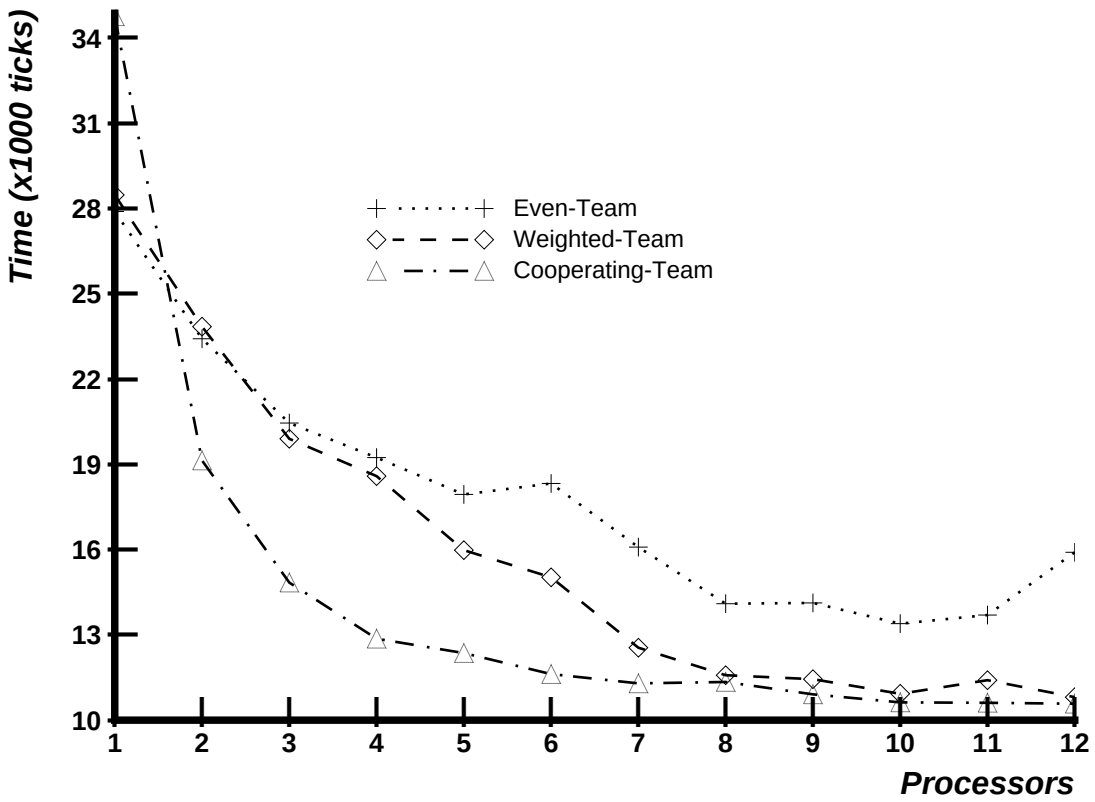

Figure 6: Quicksort: 4096 elements

Comparing the three performance lines, we can see that if the number of processors is greater than 2, the Cooperating-Team provides the best performance, the Weighted-Team is in second position and the Even-Team gives the worst performance. Note that the gap between the Even-Team and Weighted-Team lines indicates the improvement due to weights, and the gap between the Weighted-Team and Cooperating-Team lines indicates the further improvement due to the team-cooperation mechanism. Figure 7 summarizes the relative performance among three mechanisms running on six processors.

Even-Team: Divide each team into two equal sized sub-teams. When there is only 1 processor in a team, execute parallel function calls sequentially.

Weighted-Team: Divide each team according to the ratio of weights as indicated by the programmer. When there is only 1 processor in a team, execute parallel function calls sequentially.

Cooperating-Team: Divide each team according to weights, assume the weights are equal if no weights are provided. When there is only 1 processor in a team, use the team cooperation mechanism outlined in section 3.4.

4.1 Example 1: Parallel Quicksort

The first experiment that we present is a parallel version of the familiar quicksort algorithm as applied to sorting a linked-list. As indicated by the pseudo-code given in Figure 5, quicksort first divides the input list into three parts. The first part contains elements less than the pivot value (called P), the second part contains the elements equal to P, while the elements greater than P fall into the third part. Note that the first part and the third part of this list can be quicksorted independently.

```
quick-sort(head)
{
  if (length(head) > 1) {
     partition(head, &part1, &part2, &part3);
           /* comment: using first element as pivot */

     quick-sort(part1)@(length(part1))
        // quick-sort(part3)@(length(part2));
           /* comment: sort part 1 and part 3 in parallel */

     combine-lists(part1, part2, part3);
  }
}
```

Figure 5: Test case 1: Parallel Quicksort

For our purposes, the important characteristic of this algorithm is that the execution time for the two parallel calls to quicksort may vary widely. For example, consider the following input list:

3 2 1 100 23 13 35 8 9 10 3 12 7 9 30

After `partition()`, it will be divided into following three parts,

Part 1: 2 1

Part 2: 3 3

Part 3: 100 23 13 35 8 9 10 12 7 9 30

Exiting a parallel function call:

When a leading processor has finished a task, it must synchronize with its sibling team before control can be given to the statement following the parallel function call. For example, if the leading processor that was given the task for `f1()` finishes, then it must wait for the team with `f2()` to finish before control can move to the statement following the call `f1()@w1 // f2()@w2`. However, rather than just waiting idly, it checks the task pool of the leading processor of its sibling team for a task that can be stolen. If such a task exists, it executes the task.

Although the full implementation details are outside of the scope of this paper, we do provide an example of the code generated by our compiler in appendix A. This example illustrates the sort of program transformation that is required to support both the weighted team sub-division and team-cooperation that has been discussed in this section.

3.4.1 Summary of the SELSYN Task Scheduling Mechanism

In this section we have addressed our solution to the problem of scheduling tasks. In developing our approach we considered the following desirable requirements:

1. processor allocation should be inexpensive,
2. parallelism should be controlled so that scheduling overhead is reduced,
3. processors should work relatively independently in order to reduce contention for locks and minimize synchronization, and
4. a limited form of task redistribution should be incorporated to allow load balancing.

To achieve the first three requirements we introduced the notion of teams of processors and team-dividing strategies that can make use of user-supplied weights. To achieve the fourth requirement we introduced the idea of team cooperation.

4 Experimental Results

In this section, we present the performance figures for the three scheduling strategies which have been described in the previous section. In particular, we give the experimental results for a list-based quicksort, and a recursive parallel search algorithm.

For the purposes of this section we consider the following scheduling strategies:

Leading processor: The processor which has the lowest processor number in a team. This is the only processor which can issue the parallel tasks. In Figure 4, the numbers in the nodes represent the leading processors. If the team has only one processor, the single processor is the leading processor. As shown at the bottom of the Figure 4, such 1-processor teams will always be at the leaves of the team-tree.

Task Stack: A stack which keeps the information of the team-dividing and task status. Each processor has one task stack.

Task Pool: A pool may contain waiting tasks. It keeps the status of the tasks. Each processor has one task pool.

As described previously, the basic program execution model is that of team-dividing. This can be summarized as follows. A procedure starts to run sequentially on the leading processor of its allocated team. When the execution of the procedure encounters a parallel function call, the team is divided into two sub-teams with the leading processors of these two sub-teams continuing on with the individual tasks as specified in the parallel function call. When both tasks have completed the two sub-teams are joined and execution continues.

By introducing the team-cooperation, we change this basic execution model for the case in which a team has only one processor. In the new model, team-cooperation is carried out between the leading processors of the sibling teams. This cooperation is done for the situation in which a team has only one processor, but has encountered a parallel function call that provides a new parallel task. In the previous simple model, a 1-processor team running on leading processor P simply executes the two parallel tasks sequentially. However, in the team-cooperation model, processor P takes one task to perform itself, and puts a concise description of the other task on its own task pool. This task is now available to processor P's sibling team. If the leading processor of P's sibling team, call it P_{sib}, finds itself idle waiting for P to complete, then it will look in P's task pool to see if there is a task to steal. If such a task exists then P_{sib} steals it from P's pool and executes the task. Note that the stolen task may execute new parallel function calls, and if the team associated with P_{sib} has more than 1 processor, then the stolen task will be further sub-divided and executed on sub-teams associated with P_{sib}.

When P finishes its first task, it checks to see if the other task has been "stolen" by its sibling team's leading processor. If the task remains (the sibling team was always busy with its own tasks), P will take back the task and execute it. Otherwise, P can deduce that P_{sib} has stolen the task, and perhaps has generated more subtasks. Thus, P checks the task pool of P_{sib} to see if it can steal a sub-task back.

Thus, we can summarize the team-cooperation mechanism as follows:

Entering a parallel function call `f1()@w1 // f2()@w2`:

CASE 1 - The current team has more than one processor: Use the ratio of the weights to sub-divide the current team into two sub-teams, T_1 and T_2. The leading processor for T_1 executes `f1()` and the leading processor for T_2 executes `f2()`.

CASE 2 - The current team has only one processor, call it P: Put a concise descriptor of the call `f1()` on P's task pool, and execute `f2()`. After `f2()` has completed, check to see if the task for `f1()` has been stolen. If it has not been stolen, then execute `f1()`. If the task has been stolen, then its sibling team may still be working on the task, and P will look in P_{sib}'s task pool to try and steal back a sub-task.

to weight2.[3] As illustrated by the example above, these weights may be dynamically evaluated at run-time and are provided by the programmer. The weights can be used to express any sort of application specific information. In our example we illustrate one sort of information - the expected time complexity with respect to the size of the input data. Other sorts of weights may be constants, or even complex heuristics.

Comparing the simple evenly-divided team strategy with the dynamic weight-divided team strategy, we can see that the evenly-divided strategy has less overhead (no dynamic computation of weights), while the dynamic strategy may get higher processor utilization and efficiency. We would also like to emphasize that this is an example of how very simple language features can encode application specific information that is very useful to the compiler. In sequential programs there is no processor allocation to be done, and therefore no need to encode information about relative weights of function calls. However, in parallel programs we can make good use of this information which is often easily expressed by the programmer.

3.4 Team-Cooperation

Although the dynamic strategy leads to improved processor utilization, we still need to deal with the problem of unbalanced work load distribution. This situation may occur when the programmer has given no weight information, or when the weight information does not always accurately predict the execution time. Thus, at run-time we may find the situation whereby one team has only one processor and has parallel tasks to be executed, while its sibling team may have one or more processors that are idle with nothing to work on. This situation clearly causes wasted processor resources, and we would like an inexpensive mechanism to help rebalance the workload when such a situation exists.

In this subsection, we introduce a new concept of team-cooperation as a mechanism that further improves the processor-team model. This team-cooperation is used in the situation when one sub-team has processors idle and at the same time the other sub-team has run out of processors and has more parallelism that can be exploited.

In order to explain our team-cooperation mechanism, we introduce the following concepts:

Team: A collection of processors. If there is more than one processor, a team can be divided into two sub-teams that work on independent parallel tasks. For example, in Figure 4, at the beginning there is only one team, team [0 - 7], and then this team is subdivided into two sub-teams, team [0 - 3] and team [4 - 7].

Sibling Team: When a team is divided into two sub-teams, each sub-team is the sibling team of the other. For example, in Figure 4, the team [0 - 3] and the team [4 - 7] are the sibling teams of each other. Our notion of sibling team is more general in the sense that a team may have several sibling teams, it has its direct sibling as well as the siblings of all of its ancestors in the team-tree. For instance, in Figure 4 we have illustrated the situation where a number of parallel function calls have been executed, resulting in the original team, team [0 - 7], being divided into 8 1-processor teams, from team[0] to team[7]. In this case team [0] has the following sibling teams: team [1], team [2 - 3], and team [4 - 7].

[3] If no weights are given, then the division defaults to the simple strategy of dividing the team equally.

3.3 A Weighted Team-Division Strategy

As shown in Figure 4, a simple team-division strategy is to divide a team into two equal size subteams. Although this strategy can be efficiently implemented and is applicable to some types of programs, it has an obvious drawback. That is, if the work load of two tasks is uneven, processor resources will be wasted. Thus, as outlined in this subsection, we have developed the dynamic weighted team-division strategy.

Let us illustrate the problem of even team-division and our weighted team-division solution with an example. Consider the following recursive divide-and-conquer function, `foo()`.

```
foo(argn)
{  if  (base_case(argn))
     /* process base case */
     ...
   else
     { /* divide input into 2 components arg1 and arg2 */
       ...
       /* solve the two sub-problems in parallel */
       foo(arg1) // foo(arg2);
     }
}
```

If the procedure `foo()` always splits the input problem into equal sized pieces such that the time to solve `foo(arg1)` and `foo(arg2)` is approximately the same, then the even-team dividing mechanism is a good choice. However, if the size of `arg1` and `arg2` may vary widely, then the even-team mechanism will result in wasted processor resources. Now recall that our parallel function call mechanism provides the programmer with a way of specifying a weight associated with each function call. Let us assume that for this particular application, we know that the size of `arg1` and the size of `arg2` may be quite different, and we know that the time complexity for `foo` can be approximated as $O(n^2)$ where n is the size of the input to `foo`.

Clearly, we would like to be able to express this application-specific knowledge in our parallel program, and we would like our team-dividing mechanism to make use of this information. We can encode our knowledge using weights in the parallel function call. For example, we could change the line "`foo(arg1) // foo(arg2)`" in our initial program to the following:

```
{ int s1 = arg_size(arg1),  s2 = arg_size(arg2);
  int weight1 = s1 * s1,    weight2 = s2 * s2;
  foo(arg1)@weight1 // foo(arg2)@weight2;
}
```

Using this weight information available from the programmer, our weighted team-division mechanism calculates the sizes of the sub-teams based on: (1) the number of processors in the original team, and (2) the ratio of weight1

execution of parallel function calls. When a parallel function call such as `f1() // f2()` is encountered, the team is divided into two independent sub-teams, one sub-team will execute `f1()` and the other sub-team will execute `f2()`. This dividing procedure can be applied repeatedly as the problem is recursively decomposed until there is only one processor in each team.

A very simple strategy for team-dividing is illustrated in Figure 4.[2] In this figure, the numbers enclosed in the symbols "[ ]" represent the team members, and the numbers inside the node represent the processor in charge of the team. At the root node (corresponding to initiating the main program), there is only one team consisting of all processors which will take part in this computation (in this case processors 0 through 7). The main program begins executing on processor 0 and when a parallel function call is encountered, the team is divided into two equal size subteams, one subteam for each parallel function call. This dividing procedure continues for each parallel function call encountered until there is only one processor per team. When there is only one processor in a team there can be no further subdividing and subsequent parallel function calls are executed sequentially. In this manner, the overhead required to start up a new parallel task is only incurred when there are processors available for the task.

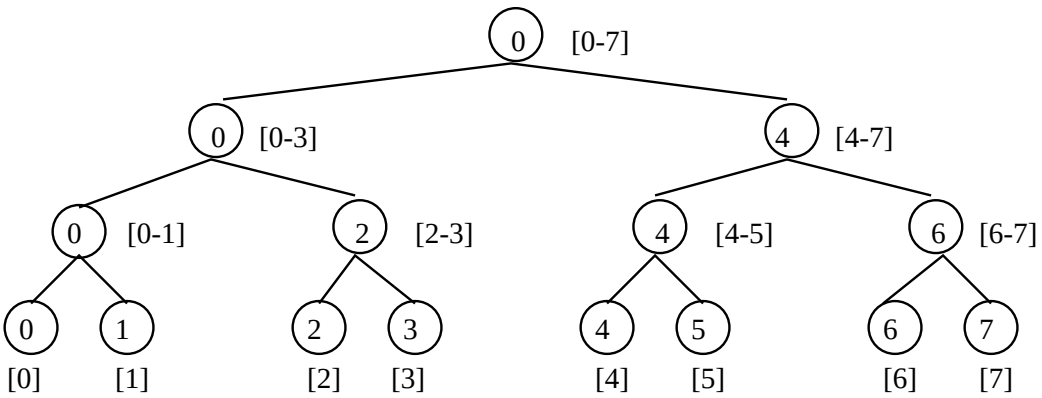

Figure 4: A Simple Form of Team Dividing

Note that in the processor-team model, the parallelism is exploited by rearranging the processors. This is different than the traditional fork-join model where processors are acquired as they are needed. In addition, since the sub-teams are independent and controlled by different processors, the amount of contention and synchronization is reduced. As we discuss in subsequent sections, this team-dividing strategy is a crucial key in achieving the goals of high utilization and efficiency.

[2]Similar strategies here have been suggested by others including Brooks [Bro89] and Hendren [Hen90]. We give a comparison of our techniques and others in section 5.

Also note that the parameter for array `a[]` corresponds to an address and therefore the corresponding argument in a function calling `sum` must be declared as `shared`. For example, if the main program has a call `sum(x,0,N-1,&sumx)`, then both `x` and `sumx` should be declared as `shared` variables.

3 Self-scheduling Mechanisms for SELSYN-C

In the previous section we have outlined our SELSYN-C parallel programming language. However, as we discussed in the introduction, the definition of a parallel language is not enough - we also need a way of effectively mapping the high-level parallel programs onto a real parallel machine. In order to solve this problem we have developed the SELSYN scheduling mechanism that is based on processor teams and a new idea of processor team cooperation. Our SELSYN-C compiler automatically inserts all of the necessary synchronization needed to support this scheduling mechanism, and thus the name of our language, **SEL**f-**SYN**chronizing **C**.

In this section we outline the motivation for our solution, introduce the notion of processor teams and a dynamic strategy for dividing teams based on programmer-specified weights, and then we discuss our solution for supporting processor team cooperation.

3.1 Motivation

One possible implementation of parallel function calls is to use the traditional fork-join model in which a parallel function call of the form `f1() // f2()` is handled by forking the call `f2()` to a new processor and then joining again after the parallel function call. This solution, however, faces two problems on most shared memory multiprocessors, such as BBN Butterfly. One problem is that the acquisition of a processor is a very expensive proposition [BAD87]. If a heavy cost must be paid to acquire a processor, the granularity of the work in a fork-join block must be very large if the code is to run efficiently. The other major problem is that there is no performance advantage in having more threads that are ready to execute than available processors. Instead, this situation represents a performance liability, since the additional threads imply increased scheduler overhead. In order to overcome these problems, we introduce a processor-team mechanism which has the ability to control the parallelism and the assignment of processors to executable threads.

3.2 The Processor-Team Model

Instead of incurring the large penalty of acquiring processors as the program executes, we have designed a mechanism that reduces processor acquisition costs through the use of processor-teams. When a program starts to execute it is assigned a team of processors that will be available to handle new tasks as they become available through the

problem. In this case the problem of summing all entries `a[l..r]` is divided into two smaller problems, summing the entries `a[l..midpoint]` and summing the entries `a[midpoint+1..r]`. Since the summing of the two halfs are independent of each other, the two recursive calls to `sum` are performed in parallel.

```
/* sum all entries a[l .. r], put in result */
sum(a,l,r,result)
int a[],l,r,*result;

{ shared int sum_left, sum_right;

  /* if only one entry left, then that is the sum */
  if (l == r)
    *result = a[l];
  else
    { /* sum left half and right half in parallel,
                        result is sum of left and right */
      int midpoint;
      midpoint = (l + r) / 2;
      sum(a,l,midpoint, &sum_left) // sum(a,midpoint+1,r,&sum_right);
      *result = sum_left + sum_right;
    }
}
```

Figure 3: An example parallel function call.

As illustrated in the previous example, the operator '//' indicates a parallel function call. In general, there are the following two forms for parallel function calls:

$$p1(arg_1, arg_2, \ldots, arg_n)//p2(arg_1, arg_2, \ldots, arg_m)$$

$$p1(arg_1, arg_2, \ldots, arg_n)@weight1//p2(arg_1, arg_2, \ldots, arg_m)@weight2$$

The second type of parallel function call allows the programmer to add some information about the relative weights of the two procedure calls. These weights can be fixed constants, or they can be other expressions that are evaluated at run-time to estimate the relative importance(weight) of the two function calls $p1(arg_1, arg_2, \ldots, arg_n)$ and $p2(arg_1, arg_2, \ldots, arg_m)$. The programmer may supply a wide variety of such weight estimates. For example, the weight might be a function of the size of the input to each calls, or the weight might be a heuristic that is guiding a search. We give a more concrete example of using weights in section 3.3.

We can also use the `sum` example to illustrate the various variable classifications. Note that it is only those variables whose address is used in parallel functions calls that must be declared as shared, all others are declared as in ordinary C programs. In our example the variables `sum_left` and `sum_right` were declared as shared variables local to the function `sum`. These two variables have a lifetime associated with the lifetime of a particular call to `sum`, and during that lifetime they can be accessed by any processor to which the parallel task may be assigned.

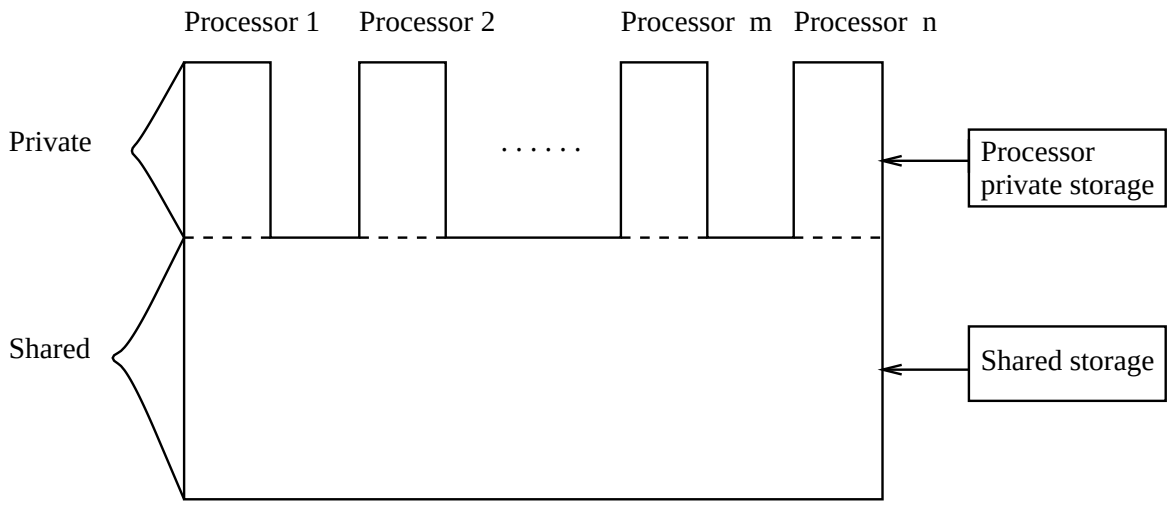

Figure 2: An abstract view of Processor-Private and Shared Memory

Some declarations of shared variables are,

```
shared int i;
shared float j[10];
shared int *pointer;
```

which defines one integer `i`, an array of 10 floats and an integer pointer.

2.1.2 Dynamically-Allocated Shared Data

In addition to statically declared variables, one needs to make a distinction between dynamically-allocated memory that is processor-private and dynamically-allocated memory that is globally-shared and thus accessible to all processors. We have supported this distinction by providing two families of allocation functions. The ordinary functions such as `malloc()` allocate processor-private memory, while the function `shared_malloc()` allocates globally-shared memory. Just as in the case of declared variables, pointers to processor-private memory may not be used as arguments to parallel function calls, while pointers to globally-shared memory may be freely used.

2.2 Parallel Function Calls: Issuing Parallel Tasks

In the SELSYN system, we support fork/join parallelism be introducing a parallel operation, which can be used by the programmer to specify that two function calls (tasks) may be executed in parallel. A typical recursive function that uses a parallel function call is given in Figure 3. This function, `sum`, is a typical divide and conquer type

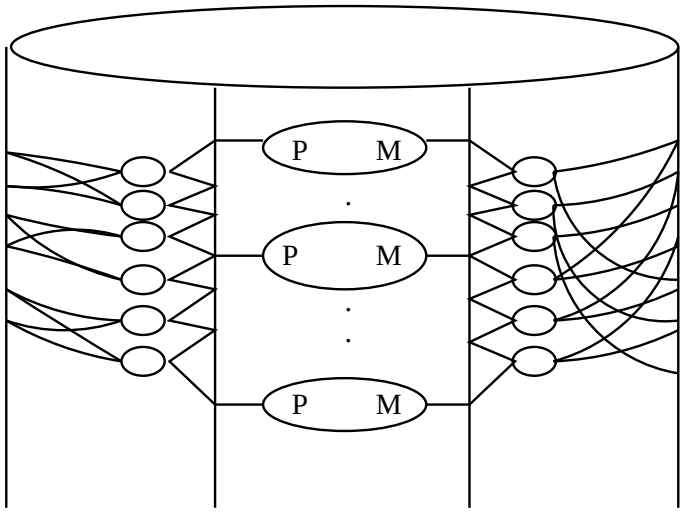

Figure 1: Architecture of the BBN parallel processor

We can abstractly view this memory hierarchy as shown in Figure 2. Note that each processor has an address space for its own processor-private storage, while the shared storage provides an address space accessible to all processors.

In order to model the distinction between processor-private and globally-shared memory in our language, we have included the concept of `shared` variables and `shared` dynamically-allocated memory. We discuss these two concepts in the following subsections.

2.1.1 Declaring Shared Variables in SELSYN-C

In SELSYN-C, all of the conventional variable declarations in C fall into the processor-private variable class. All of processor-private variables are permitted to be used as in ordinary C programs, except for the restriction that their address (pointers to the variable) should not be passed as arguments to parallel function calls.[1]

In addition to processor-private variables, SELSYN-C also supports shared variables. Shared variables are declared by adding a new leading keyword, 'shared', before the conventional declaration. This keyword indicates that the variables will be stored in shared storage address space as presented in the abstract model in Figure 2. These variables are accessible to any processor, and thus their addresses **may** be accessed by any function including parallel functions.

[1] We introduce parallel function calls in section 2.2. The general idea behind this restriction is that a parallel function call may lead to the function being executed on a different processor. Thus, all addresses that are passed to parallel functions should be accessible to all processors.

illustrative experimental results which demonstrate the effectiveness of our self-synchronizing mechanism. Finally, in section 5 we compare our approach with related approaches and in section 6 we give our conclusions.

2 SELSYN-C, A Parallel C Language

As outlined in the introduction, one of our major goals was to design a language that is simple to use and familiar to a wide range of programmers. With this goal in mind, we decided to focus on designing simple extensions to the programming language C. These extensions were selected to be easy to use, but at the same time provide the necessary high-level information necessary for an effective translation to a real parallel processor. To keep these extensions portable for different target machines, our compiler performs a source-to-source transformation. Currently we have implemented the SELSYN system on BBN Butterfly GP-1000. With this implementation the programmer specifies his or her parallel program using SELSYN-C, and our compiler produces an output program in a C dialect that is specific to the BBN Butterfly. This output program is then linked with our run-time environment to produce a program that can be executed directly on the BBN Butterfly GP-1000.

SELSYN-C supports two major extensions to C: (1) the distinction between processor-private and globally-shared data, and (2) the introduction of weighted parallel function calls. In the following subsections we discuss these two extensions in more detail.

2.1 Processor-Private versus Globally-Shared Memory

In parallel systems, it is often important to distinguish between memory local to one processor and memory accessible to all processors. For example, consider our target machine, the BBN Butterfly GP-1000. As illustrated in Figure 1, the Butterfly is a loosely coupled system in which each processor is paired with its own memory module. All processor cards are identical and independent and are connected together by a network interconnection system, called the *Butterfly Switch*, which handles the data transfers from one memory module to another.

With an architecture like that of the BBN Butterfly, a processor may access a memory location associated with another processor, but at a penalty associated with the time for communication through the switch. Thus, we can abstractly define two levels of memory as follows:

Processor-private memory: As the name suggests, data in processor private memory can be accessed only from the processor associated with that memory. In general, this is more efficient than accessing globally-shared memory.

Globally-shared memory: Data in globally-shared memory is accessible from all processors. Accesses to globally-shared memory are usually associated with a penalty for communication costs, and are therefore less efficient than accesses to processor-private memory.

Abstract

In this paper we report our design and implementation of a new self-scheduling parallel programming language, SELSYN-C. As parallel processors become more accessible to a broad range of programmers, the development of simple to use and effective programming languages becomes increasingly important. Our approach to the challenge of parallel programming language design and implementation is two-fold: (1) the design of simple extensions to C that are both easy to use, and useful for effective compilation, and (2) the design of efficient and effective scheduling strategies that can be automatically supported by a compiler and associated run-time environment.

We outline our approach by presenting: (1) our motivation, (2) an overview of the extensions to C that form the SELSYN-C programming language, and (3) the development of a new scheduling strategy that can be used to effectively compile SELSYN-C programs for a real parallel processor, the BBN GP-1000. We illustrate the effectiveness of our language and scheduling approach with real examples that have been processed by our SELSYN-C compiler.

1 Introduction and Motivation

The rapid advance of technology and architecture design is making possible rapid advances in the development of new and cheaper parallel processors. As these parallel processors become more accessible and more popular, the great challenge will be to provide software environments that make parallel processors usable by a wide variety of programmers. A critical step in this development is the design of parallel programming languages that are simple to use. Programs written in such a language should provide straight-forward mechanisms for expressing parallelism, and effective mechanisms for efficiently exploiting this parallelism on real parallel processors. That is, the language should allow the programmer to express parallelism at the application level, and not force the programmer to worry about the complicated details of resource management and synchronization of tasks for a particular parallel architecture. In addition, a good parallel programming language should support the development of portable programs.

In this paper we present our new parallel programming language, SELSYN-C, an extension of the traditional C programming language that we have designed and implemented. We have selected the programming language C as the basis of our work because it is familiar to a wide range of programmers, and thus it provides a large user base for our new language. Users of our system need only master a small set of new constructs in order to develop parallel C programs, or to convert their existing C programs to parallel C programs. The two major contributions of this work are: (1) the development of simple-to-use extensions to C, and (2) the development of a compiler and associated run-time system that supports a new kind of self-synchronizing mechanism. As demonstrated by some examples, this approach provides a simple way to specify parallel programs, and an effective way to produce efficient programs that run on a shared memory parallel processor.

The remainder of this paper is structured as follows. In section 2 we give an overview of the language extensions that are supported by SELSYN-C. In section 3 we discuss our new mechanisms that support the efficient self-synchronization of the parallel tasks that can be specified by a SELSYN-C program. In section 4 we provide some

Contents

1 Introduction and Motivation **1**

2 SELSYN-C, A Parallel C Language **2**

2.1 Processor-Private versus Globally-Shared Memory 2

2.1.1 Declaring Shared Variables in SELSYN-C 3

2.1.2 Dynamically-Allocated Shared Data 4

2.2 Parallel Function Calls: Issuing Parallel Tasks 4

3 Self-scheduling Mechanisms for SELSYN-C **6**

3.1 Motivation 6

3.2 The Processor-Team Model 6

3.3 A Weighted Team-Division Strategy 8

3.4 Team-Cooperation 9

3.4.1 Summary of the SELSYN Task Scheduling Mechanism 11

4 Experimental Results **11**

4.1 Example 1: Parallel Quicksort 12

4.2 Example 2: Parallel Search 14

4.3 Summary of Experiments 16

5 Related Work **16**

6 Conclusions **17**

A A Brief Description of the SELSYN Implementation **20**

McGill University
School of Computer Science

SELSYN-C: A Self-Synchronizing Parallel Programming Language

Weiren Ding
*Laurie J. Hendren**

ACAPS Technical Memo 36
January 24, 1992

Advanced Computer Architecture and Program Structures Group

3480 University St • Montreal • Canada • H3A 2A7

*The work supported in part by FCAR, NSERC, and the McGill Faculty of Graduate Studies and Research.