First International Workshop on High-Level Programming Models and Supportive Environments, April 1996.

An Efficient Implementation of Nested Data Parallelism for Irregular Divide-and-Conquer Algorithms

Jonathan C. Hardwick
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
jch@cs.cmu.edu

Abstract

This paper presents work in progress on a new method of implementing irregular divide-and-conquer algorithms in a nested data-parallel language model on distributed-memory multiprocessors. The main features discussed are the recursive subdivision of asynchronous processor groups to match the change from data-parallel to control-parallel behavior over the lifetime of an algorithm, switching from parallel code to serial code when the group size is one (with the opportunity to use a more efficient serial algorithm), and a simple manager-based run-time load-balancing system. Sample algorithms translated from the high-level nested data-parallel language NESL *into C and MPI using this method are significantly faster than the current* NESL *system, and show the potential for further speedup.*

1. Introduction

Nested data-parallel languages are a recent solution to the problem of easily and efficiently expressing parallel algorithms that operate on irregular data structures [4]. These irregular data structures, such as unstructured sparse matrices, graphs, and trees, are becoming increasingly prevalent in computationally intensive problems [16].

Existing parallel languages are generally either data-parallel or control-parallel in nature. The data-parallel (or collection-oriented [27]) model has a single thread of control, and is easy to understand and use. Parallelism is achieved by applying functions in parallel across a set of values, although the functions themselves must be serial in nature and are typically predefined. The data-parallel model is easy to parallelize, and is a good match to computations on regular data structures such as dense matrices and regular meshes. Unfortunately, it cannot efficiently express irregular algorithms (that is, algorithms operating on irregular data structures). By contrast, the control-parallel (or task-parallel) model achieves parallelism by using multiple threads of control, each tackling part of the problem. This allows efficient implementation of irregular algorithms, but the multiple threads of control make it correspondingly more difficult to understand and use.

The nested data-parallel model is an extension of the standard (or flat) data-parallel model, adding the ability to nest flat data-parallel structures and to apply arbitrary parallel function calls in parallel across such structures [4]. It combines the data-parallel model's parallelizability and ease of programming with the control-parallel model's run-time efficiency for irregular algorithms [10].

Although the performance of a nested data-parallel language has been demonstrated to be competitive with that of other high-level parallel languages for irregular algorithms on vector and SIMD machines [8], performance is relatively poor on the current generation of distributed-memory multiprocessors. These efficiency problems are due mainly to the reliance of current nested data-parallel language implementations on a library of low-level vector functions, which results in unnecessary interprocessor communication and a loss of data locality. This paper describes a new method of implementing nested data-parallel algorithms that uses asynchronous processor groups to reduce communication, specialized single-processor code to reduce parallel overhead, and a run-time load-balancing system to cope with dynamic data distributions. The method is initially targeted at irregular divide-and-conquer algorithms, some examples of which are listed in Table 1. On two sample algorithms (quicksort and a two-dimensional convex hull algorithm) the code produced is significantly faster than the equivalent algorithm expressed in the current NESL system (a high-level sequence-based nested data-parallel language [7]), and shows the potential for further speedup.

The rest of this paper is arranged as follows. Section 2 uses a simple irregular divide-and-conquer algorithm to il-

Algorithm	Reference
Barnes-Hut n-body	[9]
Delaunay triangulation	[2]
Geometric graph separators	[26]
Two-dimensional convex hull	[7]

Table 1. Examples of irregular divide-and-conquer algorithms.

lustrate the theoretical performance advantages of the nested data-parallel model, and the practical difficulties of writing an efficient implementation. Section 3 contains a brief summary of the nested data-parallel language NESL, and a discussion of how the structure of the current implementation limits its performance on distributed-memory multiprocessors. Section 4 presents work on a new implementation layer for NESL, and Section 5 describes the implementation of two algorithms that have been prototyped using the new system. Section 6 contains some initial performance figures, and compares them to the performance of both the current NESL implementation and hand-optimized versions of the algorithms. Finally, Section 7 discusses related work, Section 8 summarizes the paper and Section 9 outlines plans for future work.

2. Using nested data parallelism for divide-and-conquer algorithms

Divide-and-conquer algorithms provide a good illustration of the strengths of the nested data-parallel model. Figure 1 shows pseudocode for the quicksort algorithm (taken from [1]), which will be used as an example for the rest of this section. Quicksort was chosen because it is a well-known and very simple divide-and-conquer algorithm that also illustrates the problems of irregular data distribution.

Although it was originally written to describe a serial algorithm, this pseudocode contains both data-parallel and control-parallel operations. Comparing the elements of S to the pivot a and selecting the elements for the new subsequences S_1, S_2 and S_3 are data-parallel operations, while recursing on S_1 and S_2 is a good match to the control-parallel model, since we can implement it by performing two recursive function calls in parallel.

The restrictions of their language models mean that a simple data-parallel quicksort cannot exploit the control parallelism that is available in this algorithm, and a simple control-parallel quicksort cannot exploit the data parallelism that is available. By contrast, a simple nested data-parallel quicksort can exploit both sources of parallelism, using data-parallel comparison and sequence-creation operations, and

procedure QUICKSORT(S):
if S contains at most one element **then return** S
else
  begin
    choose an element a randomly from S;
    let S_1, S_2 and S_3 be the sequences of elements in S
      less than, equal to, and greater than a, respectively;
    return (QUICKSORT(S_1) followed by S_2 followed
      by QUICKSORT(S_3))
  end

Figure 1. Quicksort, a simple irregular divide-and-conquer algorithm (taken from [1]).

implementing the parallel recursion as a single parallel function call across a *nested sequence* containing the two subsequences S_1 and S_2.

Figure 2 shows the benefits of this nested data parallelism in terms of the stylized parallelism profile (parallelism versus time) of a divide-and-conquer algorithm expressed in the three different language models. The flat data-parallel algorithm achieves parallelism proportional to the length of the sequences being operated on, which is greatest at the beginning of the algorithm. The control-parallel algorithm achieves parallelism proportional to the number of possible threads, which is greatest at the end of the algorithm. The nested data-parallel algorithm is able to exploit both sources of parallelism, and achieves parallelism proportional to the original input length throughout the algorithm.

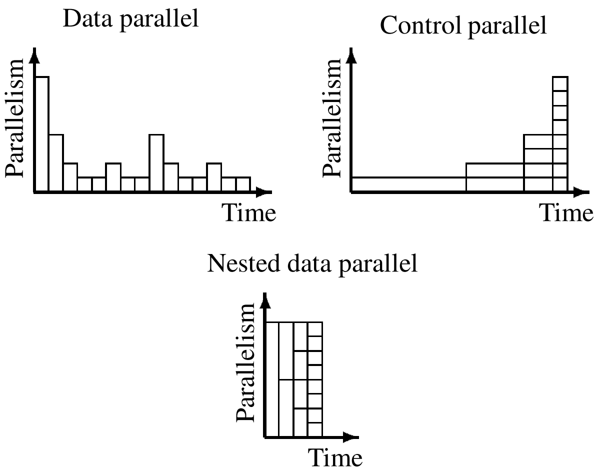

Figure 2. Parallelism profile of a divide-and-conquer algorithm expressed in three parallel language models.

Of course, in practice we must also worry about the efficiency of our implementation of nested data parallelism. For example, since quicksort is a data-dependent algorithm, we cannot know in advance how the data will be divided. A parallel implementation must therefore perform some form of load balancing at run time. A second concern is that the use of nested data parallelism should not introduce significant additional overheads beyond those of pure data-parallel or control-parallel implementations. For current distributed-memory multiprocessors, this generally means minimizing the number of accesses to local memory and to the interconnection network. The next section discusses a current implementation of nested data parallelism, and outlines some of its performance problems.

3. NESL: a nested data-parallel language

NESL was the first language to fully support nested data parallelism. It is a high-level, strongly-typed, applicative, nested data-parallel language, with sequences as the primitive data structure, and a syntax based on that of ML. NESL was designed to allow the concise description of programs in a nested data-parallel form, and provides complexity guarantees for each of its primitives.

Figure 3 shows the NESL code for quicksort. The structure is similar to that of the pseudocode in Figure 1, although the following syntax may help in understanding the code: the operator # returns the length of a sequence, the function `rand(n)` returns a random number between 0 and $n-1$, square brackets are used to access elements of a sequence, ++ is used to append two sequences, and curly braces are used to signify parallel. In particular, {`e in S | e < a`} is read as "in parallel, select all elements `e` in `S` for which `e` is less than `a`", and {`Quicksort(v) : v in [S1, S3]`} is read as "in parallel, perform `Quicksort(v)` for all elements `v` in `S1` and `S3`".

```
function Quicksort (S) =
  if (#S <= 1) then S
  else
    let a  = S[rand(#S)];
        S1 = {e in S | e < a};
        S2 = {e in S | e == a};
        S3 = {e in S | e > a};
        R  = {Quicksort(v) : v in [S1, S3]}
    in R[0] ++ S2 ++ R[1];
```

Figure 3. NESL code for quicksort.

The current NESL system consists of three layers. The first is a compiler that translates the NESL program into VCODE, an intermediate vector language [5]. The data structures that are represented as nested sequences in NESL are "flattened" into segmented vectors in VCODE. The second layer is a portable VCODE interpreter. Note that since the interpretive overhead of each VCODE instruction is amortized over the length of the vectors on which it operates, interpretation does not impose a significant performance penalty for large problem sizes [8]. The final layer is CVL, a run-time library that is the only part of the system that needs to be rewritten for a new machine. CVL provides an abstract segmented vector machine model that is independent of the underlying architecture, and a variety of data-parallel functions that operate on the vectors [6].

This three-layer model is a good match for the vector and SIMD machines originally targeted by NESL. Although their compilers could not directly generate efficient code for segmented operations expressed in a language such as C, it was easy to implement the operations in CVL in terms of the native data-parallel primitives provided by each machine. However, it is difficult to write an efficient implementation of CVL (and hence NESL) for the current generation of distributed-memory multiprocessors [17]. There are at least three reasons for this.

First, to enforce the guarantees of complexity that apply to each NESL primitive, CVL must perform a load-balancing step whenever a vector changes length. In the quicksort example, the machine representations of the three intermediate sequences `S1`, `S2` and `S3` are each independently load-balanced across the machine when they are created. Since each load-balancing step results in interprocessor communication, this can be a significant cost for algorithms with frequently-changing data structures.

Second, CVL is data parallel in operation, implicitly synchronizing all processors. Note that this is not required by NESL, but was chosen to ease the task of implementing CVL.

Finally, the structure of CVL as a library of vector functions results in poor data locality, because each CVL function is typically implemented as a loop over one or more large vectors. For example, the three intermediate sequences in quicksort are created using three separate loops, rather than the optimal single fused loop, tripling the number of data cache misses in that section of the code.

The rest of this paper describes work on a new implementation layer for nested data-parallel languages such as NESL that can be used to solve these performance problems. This layer is initially targeted at irregular divide-and-conquer algorithms, which offer the greatest potential for performance gains and represent a wide range of significant computational problems.

4. A new implementation layer

The new implementation layer is designed to avoid the problems of implicit load balancing and synchronized data-parallel operation present in CVL. There are also some opportunities for improvements in per-processor data locality.

The central idea is to match the aggregate behavior of processors to the run-time behavior of the divide-and-conquer algorithm being implemented. That is, data-parallel behavior at the beginning of the algorithm, control-parallel behavior at the end of the algorithm, and combined data-parallel and control-parallel behavior in between the two extremes.

Given a divide-and-conquer algorithm, we can implement this idea using a message-passing system such as MPI [20]. At the beginning of the algorithm, its data structures are spread in a block-cyclic fashion across the machine. The processors are in a single group, executing flat data-parallel operations. At the first recursive step, the initial group of processors subdivides into independent subgroups, with one subgroup per parallel function call, and the algorithm's data structures are redistributed amongst the subgroups in a global communication step. Note that the subgroups need not have equal numbers of processors. Indeed, for an algorithm such as quicksort, where a poor choice of pivot can lead to greatly differing subset sizes, we will want to choose the ratio of the number of processors in each subgroup to approximate the ratio of the sizes of the subsets of data that the groups will hold.

Each subgroup then behaves as a smaller version of the original group, operating in a data-parallel manner on its subset of the data, but running asynchronously with respect to other groups. At each successive recursive step, more and more groups are formed, with fewer and fewer processors in each group. When a processor has recursed down to a group of size one, it switches to a serial version of the parallel algorithm. This serial code only has to operate on local data, and hence will normally have lower constant factors than the parallel version of the code. Rather than relying on a serial translation of the parallel algorithm, the user can even supply an entirely new serial algorithm, which may result in a lower complexity as well as lower constant factors. After a processor's recursion terminates, it returns from serial to parallel code, and the groups of processors reform in reverse order as results are passed back up the call tree of the user's algorithm.

For a binary divide-and-conquer algorithm running on p processors with a problem of size n, and $\log n$ recursive calls of the algorithm, the expected case is for each processor to execute $\log p$ of the calls in parallel code. Assuming $n > p$, the remaining $\log n - \log p$ calls will be executed in serial code, with lower constant factors (and possibly lower algorithmic complexities, if the programmer has supplied a different serial algorithm).

4.1 Load balancing

Although there is some approximate load balancing in the choice of group sizes to match data subset sizes, this is not enough to cope with irregular and data-dependent algorithms. The prototype therefore adds a simple run-time load-balancing system, using one processor as a centralized manager. The manager mediates between processors that have finished their serial phase and those that have work remaining. Whenever a processor finishes a serial phase, it informs the manager. If all processors have finished, the manager signals the return to parallel code; otherwise, the manager adds the idle processor to a queue. Whenever a processor is about to recurse on more than a predefined amount of data in a serial phase, it asks the manager for help. If no idle processor is available, the manager denies the request, and the original processor must continue processing on its own. Otherwise, the manager removes an idle processor from the queue, and tells it which processor to help and the amount of data to expect. The helper allocates a buffer to receive the data, and informs the original processor. The original processor then ships the arguments of the recursive call to the helper, continues with its other recursive calls, and waits for the helper to return the result of the first call. Finally, the helper again tells the manager that it is available, and is added to the idle queue.

There are several important points to note about this load-balancing system. First, all communication is point-to-point and blocking, with message sizes known in advance by the receiver. This can be used by MPI implementations to reduce buffering and improve transfer rates. Second, the load-balancing strategy is fixed, with no data structures other than the manager's idle queue. Third, load balancing only takes place when processors are running serial code. No attempt is made to load-balance the multi-processor groups that are running parallel code. Finally, the requirement for a dedicated processor to act as the manager is a side effect of relying on the current generation of portable message-passing systems, which use explicit send-receive function pairs. In this model, we cannot interrupt a processor to tell it to offload or accept some work, and it is impractical to add a function call to the inner loop of each algorithm in order to poll for messages.

The performance of this load balancing system will be shown in Section 6, and the final three points raised above will be returned to in Section 9.

5. Prototype system

The current prototype relies on the user's algorithm being translated into ANSI C and linked against a run-time library containing generic communication routines and the load-balancing system. This translation is currently done by hand. Simple element-wise operations (e.g., comparing the pivot in quicksort to every element of the input sequence) are expanded into per-processor loops across the data of the sequence. Vector-scalar operations (e.g., extracting a sequence element to act as the pivot) are mapped to broadcast or send operations. Sequence manipulation operations involve both per-processor loops and inter-processor communication. For example, selecting out the elements of quicksort's input sequence to form a subsequence involves a local count of the number of elements to select, an exchange of this information amongst the processors in the current group, and finally an all-to-all communication step within the group to create the new subsequence. This is all wrapped up into a single *pack* function which is part of the run-time library. Finally, nested data parallel function calls are implemented using the group subdivision method explained in Section 4. The structure of the C code for this step of quicksort is shown in Figure 4.

```
task_group group, new_group;
vec_double S, S1, S3, S1_or_S3, R0, R1;

/* One group gets S1, the other gets S3 */
new_group = split_groups (group, S1, S3,
                          &S1_or_S3);
/* Now we're in two separate groups */
if (new_group.nproc == 1) {
  /* If group size is 1, run serial code */
  R = quicksort_serial (S1_or_S3);
  idle_loop ();           /* Load balancing */
} else {
  /* Else recurse with new vector and group */
  R = quicksort_parallel (S1_or_S3, new_group);
}
/* Groups rejoin, putting results in R0 & R1 */
extract_2_vec_double (S, &R_0, &R_1);
```

Figure 4. Quicksort group subdivision in C.

6. Results

This section presents initial benchmark results for two irregular nested data-parallel algorithms. The two algorithms are quicksort and quickhull, an efficient divide-and-conquer algorithm for finding the 2D convex hull of a set of points. The NESL code for the recursive step of quickhull is shown in Figure 5. The function `hsplit` returns all (x, y) coordinates in the sequence `points` that are to one side of the line formed by the points `p1` and `p2`. For further details, see [7].

All experiments were performed on an Intel Paragon running OSF R1.2, using `icc -O3` and MPICH 1.0.11. The NESL system used an MPI-based version of CVL [17]. For quicksort, sequences of pseudo-random 64-bit floating-point numbers were used as input. For quickhull, uniform distributions of points inside the unit square were projected onto (x, x^2+y^2), resulting in convex hulls of approximately $\sqrt{n}$ points for an input sequence of size n. For all benchmarks, the average time over 50 runs is shown, with a different input sequence being generated for each run. The data

```
function hsplit (points, p1, p2) =
let cross = {cross_product (p, (p1, p2)):
             p in points};
    packed = {p in points; c in cross
              | plusp(c)};
in if (#packed < 2) then [p1] ++ packed
   else
     let pmax = points[max_index (cross)];
     in flatten({hsplit(packed,p1,p2):
                  p1 in [p1,pmax];
                  p2 in [pmax,p2]});
```

Figure 5. NESL code for recursive step of quickhull.

size at which a processor requested help was fixed at 20% of the expected data per processor. The problem sizes used (up to 2M elements for quicksort and up to 4M elements for quickhull, both on 32 processors) are the maximum achievable by the current NESL system without swapping. Full performance data is given in Appendix A.

The new implementation layer is significantly faster than the current NESL system. The load-balanced quicksort is 15–33 times faster than NESL, and the load-balanced quickhull is 8–18 times faster. In both cases, higher ratios occur at smaller problem sizes, where the overhead of the VCODE interpreter becomes significant. Additionally, higher ratios are achieved on larger numbers of processors, where scalability of inter-processor communication becomes important. Since CVL performs more communication, due to its implicit load balancing and its inability to switch to serial code, it starts experiencing scalability problems sooner.

The remainder of this section will concentrate on the effects of load balancing and the potential benefits of using an optimizing compiler. Since both of these give relatively small additional speedups which would be lost on a large scale, the graphs do not include NESL performance data.

To illustrate the effects of load balancing, Figure 6 and Figure 7 show the behavior over time of each of eight processors in a sample quicksort run, with and without load balancing. The figures were produced using the `upshot` visualization tool, and are to scale. The same sequence of 0.5M elements is being sorted.

Figure 6 shows quicksort's behavior without load balancing. Initially, there are alternating phases of data-parallel computation within a group ("Parallel"), and group subdivision causing data redistribution ("Group"). The asynchronous behavior of different groups can be seen, with processor two rapidly forming a singleton group and switching to serial code ("Serial") at about the 0.5-second mark, while processors six and seven are members of four successively smaller groups before switching to serial code. However, by about the 1.0-second mark, all processors have recursed

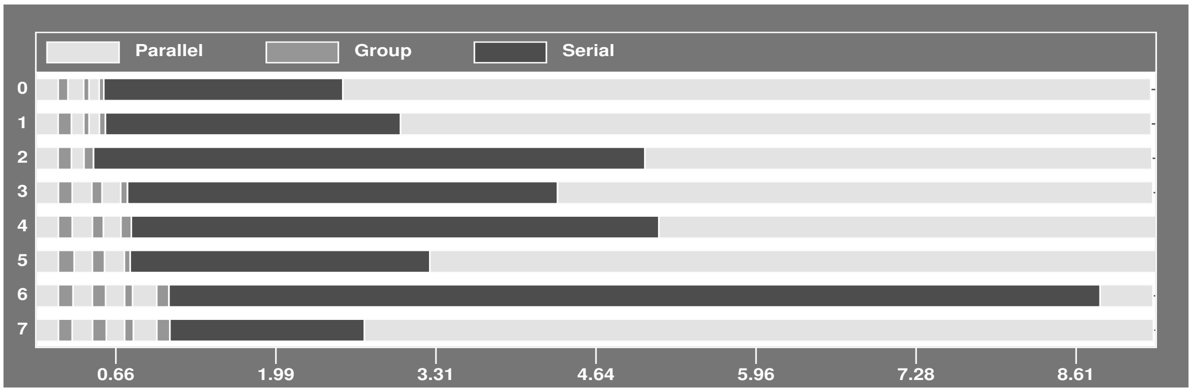

Figure 6. Behavior of eight processors running parallel quicksort without load balancing. Phases are parallel computation ("Parallel"), group subdivision ("Group"), and serial computation ("Serial").

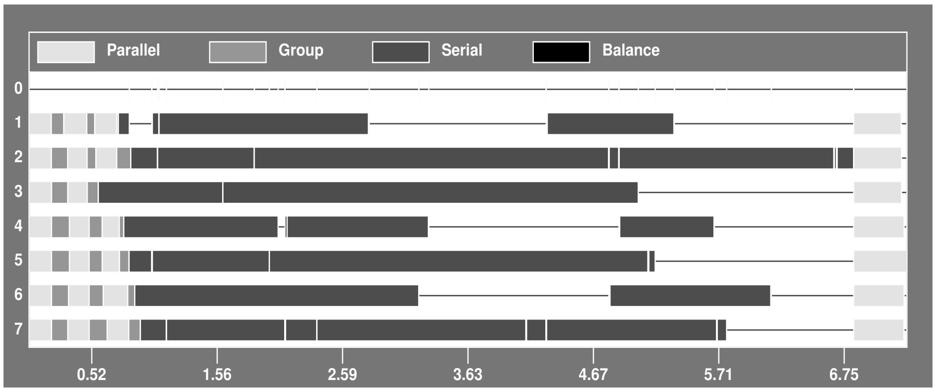

Figure 7. Behavior of seven processors plus dedicated manager running parallel quicksort on the same data set, with simple load balancing. Additional phase is balancing ("Balance").

down to singleton groups and are running serial code. Now we see the effects of an irregular and data-dependent algorithm; although half of the processors finish their serial work and have returned to parallel mode by the 3.3-second mark, processor six does not finish its serial work until almost 9 seconds have elapsed. The parallel data coalescing phase, where sequences are appended and passed back up the call tree, then takes less than half a second to complete.

Figure 7 shows quicksort's behavior with load balancing. Processor zero is now a dedicated manager, and is almost entirely idle. The other processors initially behave as before, with alternating phases of data-parallel computation and group subdivision until all processors are executing serial code. Processor one finishes its serial work almost immediately. At around the 1.0-second mark, processor six recurses on an input sequence large enough to make it ask the manager for help. Since processor one is available, the manager assigns it to receive some of processor six's work. As the algorithm proceeds, more processors become available and more load balancing takes place, although there are extensive gaps, representing idle processors.

For the rest of the benchmarks, both algorithms were tested before and after adding load balancing (labelled ''base implementation" and "plus load balancing" in the figures). The load-balanced variants were then hand-tuned, resulting in the three subsequence loops in quicksort being fused into one, and the `cross` and `packed` loops in quickhull being fused together. These variants are labelled "plus fused loops" in the graphs, and represent what could be achieved with a compiler capable of symbolic analysis.

6.1 Quicksort benchmarks

Figure 8 shows the performance of the three versions of quicksort on a 32-processor Paragon, for different problem sizes. The importance of load balancing is obvious, with the load-balanced version being 1.19–1.50 times faster than the base implementation. Fused loops give an additional speedup of 1.13–1.23 times. To illustrate the performance gains possible by using a tuned serial algorithm instead of the standard serial translation of parallel code, a fourth version uses the system `qsort` function when it is in a group of size one ("Using system qsort"). This is 1.27–1.87 times faster than the fused-loop version, with the difference being greatest at large problem sizes.

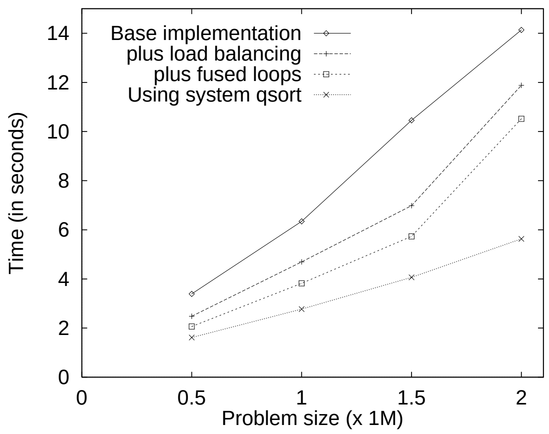

Figure 8. Performance of three versions of quicksort on a 32-processor Paragon.

Figure 9 shows the effect on quicksort of keeping the problem size fixed while varying the number of processors. The trends are similar to those in Figure 8, with load balancing improving performance by 1.21–1.40 times, and fused loops by 1.15-1.30 times. Note that fused loops make more of a difference for small machine sizes, where communication is minimized and the impact of memory traffic on overall performance is more significant. Using the system qsort function is 1.13–1.83 times faster than fused loops, and the difference is again greatest at small machine sizes.

6.2 Quickhull benchmarks

When the quickhull algorithm of Figure 5 is given a uniform input distribution, it is actually very regular in its data division. All processors finish at about the same time, and the load-balancing system becomes an active handicap, since it requires both a dedicated processor and extra inter-processor communication. The quickhull results can therefore be considered a worst-case test for the load-balancing system. Figure 10 shows the performance of the three versions of quickhull running on a 32-processor Paragon, for different problem sizes. The speedup due to load balancing varies between 0.93 (that is, slowing the system down) and 1.02. The addition of fused loops improves performance by an additional factor of 1.09–1.10, reflecting the reduced opportunity for optimization of the quickhull algorithm.

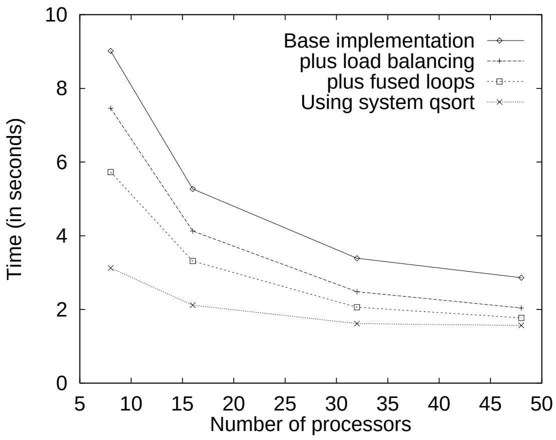

Figure 9. Performance of three versions of quicksort for a problem size of 0.5M elements.

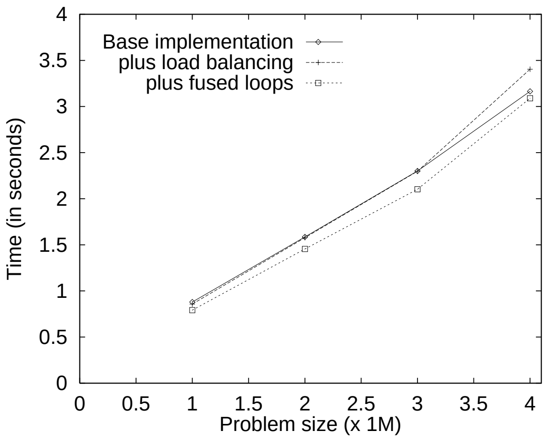

Figure 10. Performance of three versions of quickhull on a 32-processor Paragon, with a uniform input distribution.

Figure 11 shows the effect on quickhull of keeping the problem size fixed while varying the number of processors. Again, the trends are similar, with load-balancing improving performance by 0.83–1.04 times, and fused loops by 1.07–1.14 times.

7. Related work

One of the main ideas behind this work was independently arrived at by Barnard [3], who has implemented a parallel multi-level recursive spectral bisection algorithm on the Cray T3D using *recursive asynchronous task teams*, which are analogous to the processor groups described earlier. However, the work described here goes further in three areas. First, it includes a dynamic load-balancing system

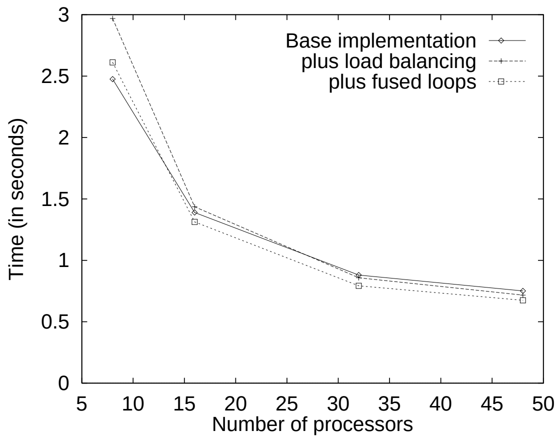

Figure 11. Performance of three versions of quickhull for a problem size of 1M elements, with a uniform input distribution.

for irregular algorithms. Second, it switches to serial code where possible, avoiding the overhead of parallel code. Finally, it is built on top on C and MPI and hence is widely portable, rather than being tied to the shared-memory architecture of a particular machine.

The PCP system also supports a variant of asynchronous task teams, although under explicit programmer control [11]. PCP is a data-parallel extension of C that is designed to exploit nested parallelism. The programmer can defer choice of team sizes until run time, allowing for some approximate load balancing. Again, there is no dynamic load balancing, and no use of serial code.

Previous work on improving the performance of NESL has involved the compilation of VCODE into multi-threaded C for a shared-memory multiprocessor [14]. The compiler used extensive symbolic loop analysis and program graph clustering to improve locality and reduce synchronization, but retained a flat, implicitly load-balanced model.

There are now several other nested data-parallel languages besides NESL. Proteus is a high-level architecture-independent programming language designed for rapid application prototyping [21]. V extends nested data parallelism to the imperative programming model of C [13], and the language developed by Sheffler and Chatterjee adds nested constructs to C++ [25], using a flattening technique to transform nested data-parallel code into ordinary C++. However, all three of these languages use CVL as a base implementation layer, and are therefore limited by its performance problems on current distributed-memory multiprocessors. Adl, a functional language similar to NESL, uses a multi-threaded implementation instead of CVL, but has so far only been implemented on the CM-5 [15].

Compiling serial and parallel versions of the same code is also done by the Illinois Concert System compiler for two fine-grained object-oriented languages (IC++ and Concurrent Aggregates), using a multi-level execution model [24]. Although the compiler is aimed at a thread-based language model rather than a nested data-parallel model, it tackles many of the same problems, namely run-time adaptation to changing data layouts, use of sequential code to improve efficiency, and minimizing the overhead of parallel code. Additionally, Chakrabarti and others have analyzed the theoretical benefits of mixed data and control parallelism [12]. They conclude that best results are obtained when communication is slow or when there is a large number of processors, and that a single switch between data and control parallelism can achieve most of the benefits of a more general model.

Most work on parallel divide-and-conquer algorithms has concentrated on regular algorithms, such as Fast Fourier Transforms, sorting networks, and prefix sums. Divacon is a formal model for such algorithms and was implemented on the CM-2 [23]. More recently, the *powerlist* has been proposed as the basic data structure of a similar model, exposing the parallelism and recursion in an algorithm [22]. However, neither model is designed to deal with the data-dependent computations typical of irregular algorithms.

Load-balancing is one of the most heavily-studied areas of parallel computation. The interested reader is referred to a survey such as [19].

8. Summary and conclusions

The promise of nested data-parallel languages has been hampered by the inefficiency of their current implementation on distributed-memory multiprocessors. This paper has described work in progress on a new implementation method targeted at irregular divide-and-conquer algorithms that uses dynamic processor groups to match the run-time behavior of algorithms, separate parallel and serial versions of code to reduce constant factors where possible, and a load-balancing run-time system. Sample algorithms expressed in this way have shown significant speedups over the current implementation method, and further improvements are possible.

9. Future work

Besides implementing a full compiler for the system described here, other short-term goals include benchmarking on other architectures, and quantifying the performance effects of the different parts of the system.

There are several interesting longer-term research issues involving the load-balancing system. The current simplistic approach can clearly be improved by using information on the number of idle processors and the relative processor loads to control the system's behavior. This should reduce the idle time exhibited by processors in Figure 6. Additionally, giving the system more information about an algorithm's complexity, either as a user-supplied hint or by run-time monitoring, would allow it to load-balance more effectively (the current system estimates load according to data size rather than expected run time, and hence is optimal only for $O(n)$ algorithms). The system could be extended to operate in the parallel phase of the algorithm, dealing with multi-processor groups as well as single processors [18]. Finally, the use of threads or active messages would allow the load-balancing system to be distributed across the machine, rather than tying up one processor.

10. Acknowledgements

I would like to thank my adviser, Guy Blelloch, for his guidance and suggestions.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] F. Aurenhammer. Voronoi diagrams: a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23:345–405, 1991.

[3] S. T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proceedings of Supercomputing '95*, Nov. 1995.

[4] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[5] G. E. Blelloch and S. Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of Symposium on The Frontiers of Massively Parallel Computation*, pages 471–480, Oct. 1990.

[6] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, M. Reid-Miller, J. Sipelstein, and M. Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[7] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, and M. Zagha. NESL user's manual (for NESL version 3.1). Technical Report CMU-CS-95-169, School of Computer Science, Carnegie Mellon University, July 1995.

[8] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.

[9] G. E. Blelloch and G. Narlikar. A comparison of two n-body algorithms. In *Proceedings of DIMACS International Algorithm Implementation Challenge*, Oct. 1994.

[10] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, Feb. 1990.

[11] E. D. Brooks III, B. C. Gorda, and K. H. Warren. The parallel C preprocessor. *Scientific Programming*, 1(1):79–89, 1992.

[12] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, July 1995.

[13] M. M. T. Chakravarty, F. W. Schroer, and M. Simons. V—nested parallelism in C. In *Proceedings of Workshop on Massively Parallel Programming Models*, Oct. 1995.

[14] S. Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, Oct. 1991.

[15] D. Engelhardt and A. Wendelborn. A partitioning-independent paradigm for nested data parallelism. In *Proceedings of International Conference on Parallel Architectures and Compiler Technology*, June 1995.

[16] G. C. Fox. The architecture of problems and portable parallel software systems. Technical Report SCCS-134, Syracuse Center for Computational Science, Syracuse University, 1991.

[17] J. C. Hardwick. Porting a vector library: a comparison of MPI, Paris, CMMD and PVM. In *Proceedings of Scalable Parallel Libraries Conference*, pages 68–77, Starkville, Mississippi, Oct. 1994.

[18] J. C. Hardwick. A portable toolbox for nested data-parallel algorithms on distributed-memory multiprocessors. PhD proposal, School of Computer Science, Carnegie Mellon University, Apr. 1995.

[19] C. N. M. Karavassili. Description of the adaptive resource management problem, cost functions and performance objectives. Technical Report P8144, ESPRIT III, Dec. 1994.

[20] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications and High Performance Computing*, 8(3/4), 1994.

[21] P. H. Mills, L. S. Nyland, J. F. Prins, J. H. Reif, and R. A. Wagner. Prototyping parallel and distributed programs in Proteus. In *Proceedings of Symposium on Parallel and Distributed Processing*, Dec. 1991.

[22] J. Misra. Powerlist: a structure for parallel recursion. In *A Classical Mind: Essays in Honor of C.A.R. Hoare*. Prentice-Hall, Jan. 1994.

[23] Z. G. Mou and P. Hudak. An algebraic model of divide-and-conquer and its parallelism. *Journal of Supercomputing*, 2(3):257–278, Nov. 1988.

[24] J. Pleyvak, V. Karamcheti, X. Zhang, and A. A. Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Proceedings of Supercomputing '95*, Dec. 1995.

[25] T. J. Sheffler and S. Chatterjee. An object-oriented approach to nested data parallelism. In *Proceedings of Symposium on The Frontiers of Massively Parallel Computation*, Feb. 1995.

[26] H. D. Simon and S.-H. Teng. How good is recursive bisection? Technical Report RNR-93-012, NASA Ames Research Center, 1993.

[27] J. Sipelstein and G. E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, Apr. 1991.

A. Performance results

Data for Figure 8: time in seconds for quicksort on 32 processors, varying the problem size.

Algorithm version	Problem size 0.5M	1.0M	1.5M	2.0M
NESL	66.5	102.9	140.8	178.1
New implementation	3.39	6.34	10.46	14.13
Plus load balancing	2.48	4.69	6.98	11.88
Plus fused loops	2.06	3.82	5.73	10.52
Using system qsort	1.62	2.77	4.06	5.63

Data for Figure 9: time in seconds for quicksort of 0.5M elements, varying the number of processors.

Algorithm version	Number of processors 8	16	32	48
NESL	113.6	77.1	66.5	68.3
New implementation	9.02	5.27	3.39	2.86
Plus load balancing	7.46	4.12	2.48	2.04
Plus fused loops	5.73	3.31	2.06	1.77
Using system qsort	3.13	2.11	1.62	1.57

Data for Figure 10: time in seconds for quickhull on 32 processors, varying the problem size.

Algorithm version	Problem size 1.0M	2.0M	3.0M	4.0M
NESL	11.6	17.4	22.0	26.2
New implementation	0.88	1.59	2.30	3.16
Plus load balancing	0.86	1.58	2.30	3.40
Plus fused loops	0.79	1.45	2.10	3.09

Data for Figure 11: Time in seconds for quickhull of 1.0M elements, varying the number of processors.

Algorithm version	Number of processors 8	16	32	48
NESL	19.8	13.9	11.6	13.2
New implementation	2.47	1.39	0.88	0.75
Plus load balancing	2.97	1.44	0.86	0.72
Plus fused loops	2.61	1.31	0.79	0.67