Journal of the South Carolina Academy of Science 2(1):42-60 Fall 2004

A PROGRAMMING ENVIRONMENT FOR STATIC AND DYNAMIC DISTRIBUTED SYSTEMS

Seyed H. Roosta

Department of Computer Science, University of South Carolina Upstate, Spartanburg, SC 29303

ABSTRACT

The programming of distributed systems requires specific development and analysis tools. The difficulty arises because current programming languages require the programmer to specify a problem to be solved at a low level of abstraction in an imperative form. An alternative approach is to specify the problem to be solved at a high-level in a functional language. Program transformation can then be used to derive a parallel algorithm. Such algorithms can be run on parallel computers which automatically exploit the implicit parallelism in a functional language program. We present a new methodology for systematically synthesizing algorithms for various parallel architectures (static and dynamic). Our technique would be applied to produce parallel algorithms for problems as diverse as dynamic programming, tessellation of the plane, fractal image generation and Fourier transformation.

INTRODUCTION

The widespread use of parallel computers has been hampered by the difficulty of load distribution amongst a cooperating group of processors. The difficulty arises because current programming languages require the programmer to specify a problem to be solved at a low level of abstraction in an imperative form. Thus the programmer must immediately encode an architecture-specific algorithm detailing every communication and computation. This process is prone to error and complicates the reuse of applications. An alternative approach is to specify the problem to be solved at a high-level in a functional language. Program transformation can then be used to derive a parallel algorithm. Such algorithms can be run on parallel computers which automatically exploit the implicit parallelism in a functional language program. We show this by producing functional language code that explicitly expresses the computations and communications to be performed by the processors. This simplifies compilation, yields faster programs and enables parallel applications to be developed for a wide variety of parallel computer architectures. We present a new methodology for systematically synthesizing algorithms for various parallel architectures (static and dynamic). Thus we overcome one of the main problems associated with program synthesis by unfold/fold transformation (Arpaci 2001, Deminet 1982, Diniz 1999), in which it simplifies the possible program transformation at any stage in the synthesis. With an architecture specification, the synthesis is much more focused on the need to remove redundant computations by introducing interprocessor communication. Our technique would be applied to produce parallel algorithms for problems as diverse as dynamic programming, tessellation of the plane, fractal image generation and Fourier transformation.

In this paper, two new goal-seeking program transformation methodologies have been developed as the following:

1. **Transformation to Static Architectures:** Processors are modeled by functions and interprocessor communication is modeled by function decomposition.
2. **Transformation to Dynamic Architectures:** Processors are modeled by functions and message routing mechanism is modeled by set abstraction.

The methodologies enable a high-level functional specification of the application and a high-level functional abstraction of the target computer architecture to be systematically manipulated to produce an efficient parallel algorithm tailored to the target architecture.

We outline the technique and demonstrate its effectiveness by producing two parallel sorting algorithms for two different architectures (Pipeline and Message-Passing). The next section briefly describes the design approach in which machine-independent issues such as concurrency are considered early, and machine-dependent aspects of design are delayed until late in the design process. Section 3 describes models of computation for static and dynamic architectures. Section 4 provides a categorization of load distribution. Section 5 previews the two new transformation algorithms designed to solve the sorting problem in the two classes of parallel computers. Finally, in section 6 we list some concluding remarks.

Design Approach

Most programming problems have several parallel solutions (Gehani 1984, Roosta 2000, Lim 1994, Kwok 1999). The best solution may differ from that suggested by the existing sequential algorithm. The design methodology that we describe is intended to foster an exploratory approach to design in which machine-independent issues such as concurrency are considered early, and machine-dependent aspects of design are delayed until late in the design process. This methodology structures the design process as four distinct stages. In the first two stages, we focus on concurrency and scalability and seek to discover algorithms with these qualities. In the third and forth stages, attention shifts to locality and other performance-related issues. The four stages are illustrated in Figure 1 and can be summarized as follows:

- **Partitioning:** The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.
- **Communication:** The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
- **Agglomeration:** The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.
- **Mapping:** Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically (at compile time) or dynamically (at run time) by load-balancing algorithms.

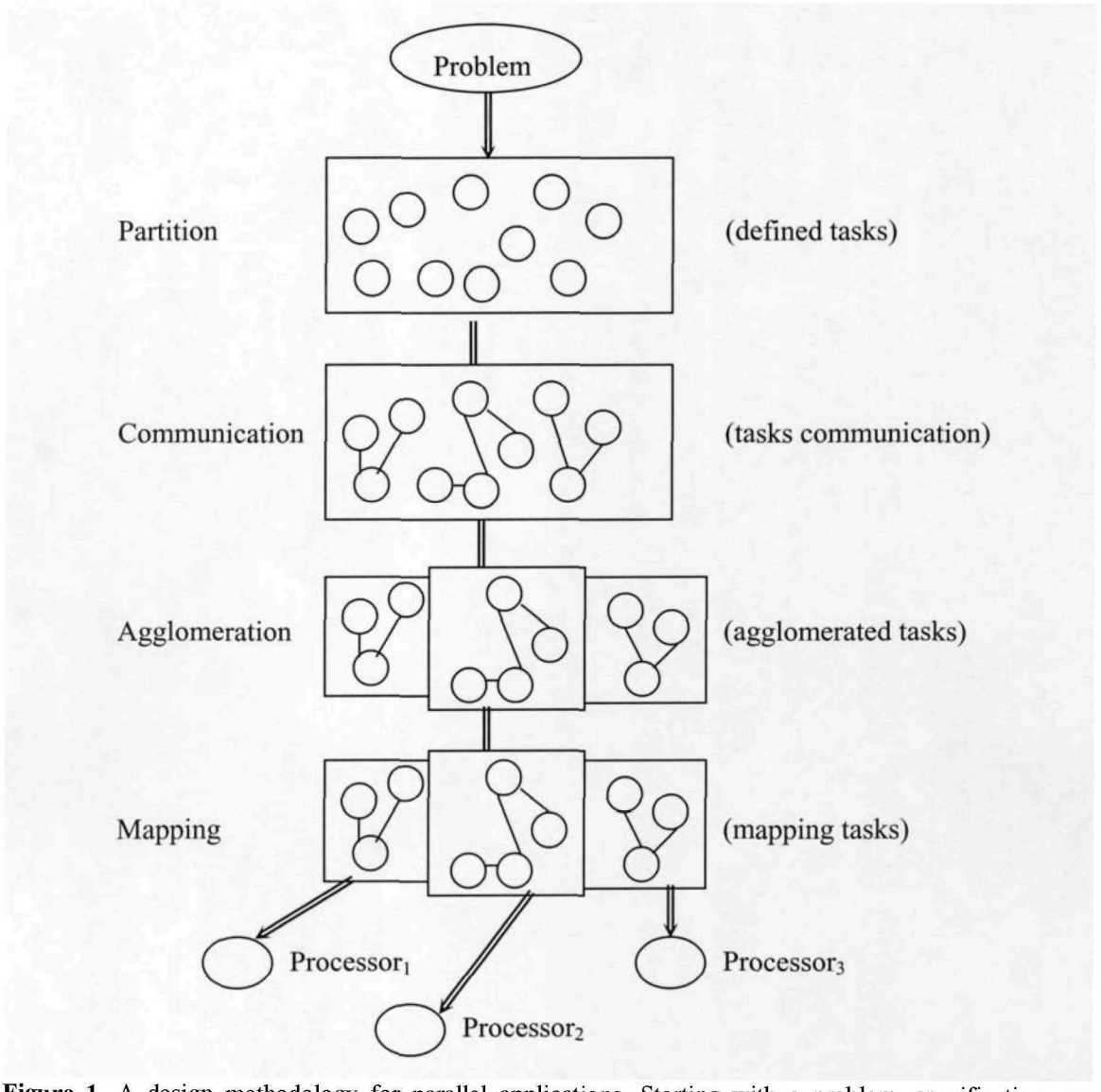

Figure 1. A design methodology for parallel applications. Starting with a problem, specification, we develop a partition, determine communication requirements, agglomerate tasks, and finally map tasks to processors.

In the final stage of the parallel algorithm design process, we specify where each task is to be executed. The mapping problem is know to be NP-Complete (Attie 2001, Bitton 1984, Deminet 1984), meaning that no computationally tractable (polynomial time) algorithm can exist for evaluating these tradeoffs in the general case. However, considerable knowledge has been gained on specialized strategies and heuristics and the classes of problem for which they are effective. This mapping problem does not arise on uniprocessor or on shared-memory computers that provide automatic task scheduling (Greenbaum 1989, Hasselbring 2000, Roosta 2001). In these computers, a set of tasks and associated communication requirements is a sufficient specification for a parallel algorithm; operating system or hardware mechanisms can be relied upon to schedule executable tasks to available processors. Unfortunately, general-purpose mapping mechanisms have yet to be developed for scalable parallel computers. In general,

mapping remains a difficult problem that must be explicitly addressed when designing parallel algorithms. Our goal in developing mapping algorithms is normally to minimize total execution time (Tzen 1993, Saks 2000). We use two strategies to achieve this goal:

Asynchronous Mapping: We place tasks that are able to execute concurrently on different processors, so as to enhance concurrency.

Synchronous Mapping: We place tasks that communicate frequently on the same processor, so as to increase locality.

Models of Parallel Computations

We present new program transformation techniques for two classes of parallel computer architectures as the following:

- **Static Architectures** have fixed interprocessor connection and these are represented straightforwardly in a functional language: processors are modeled by functions and interprocessors communication can be modeled by function composition ($f \bullet g$). The expression $(f \bullet g)x$, which means $f(g(x))$, indicates that the processor calculating f takes its input from the processor calculating $g(x)$.
- **Dynamic Message-Passing Architectures** have a message routing mechanism which routes messages of the form *Msg*(destination, contents) to the appropriate destination processor. Thus any processor may send messages to any other processor. These architectures may be represented in a functional language using functions to model the processors and set abstraction to model the message routing. The router may be physically implemented in various ways, for example, in the ALICEC machine (Burns 1989) it is implemented using a delta topology; in the Thinking Machines Connection Machine CM2 (Connection-Machine 1987) the routing is achieved by hypercube hardware and routing software.

Figures 2 and 3 show functional language representation of these two types of architectures.

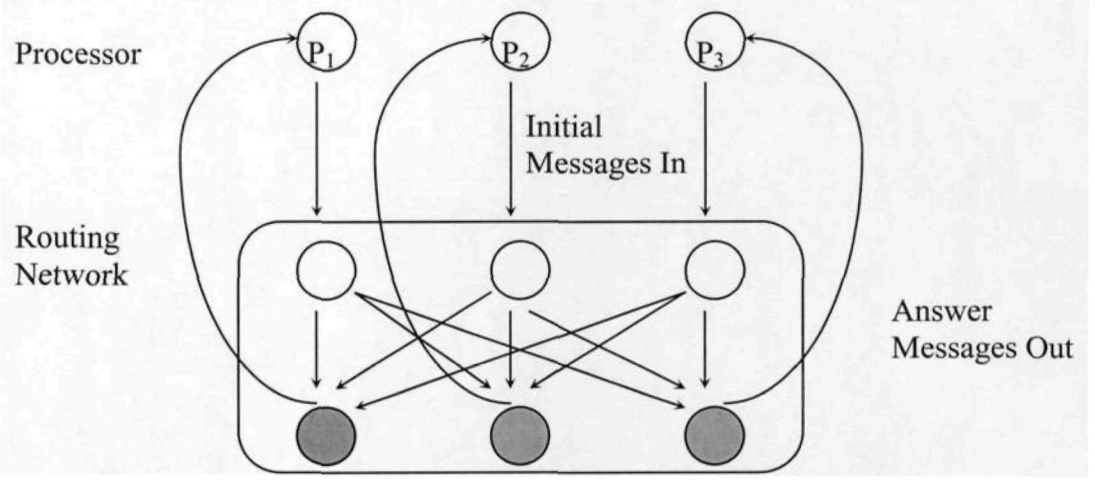

Figure 2. Functional representation of a dynamic message-passing architecture. Initial messages are determined by the problem specification.

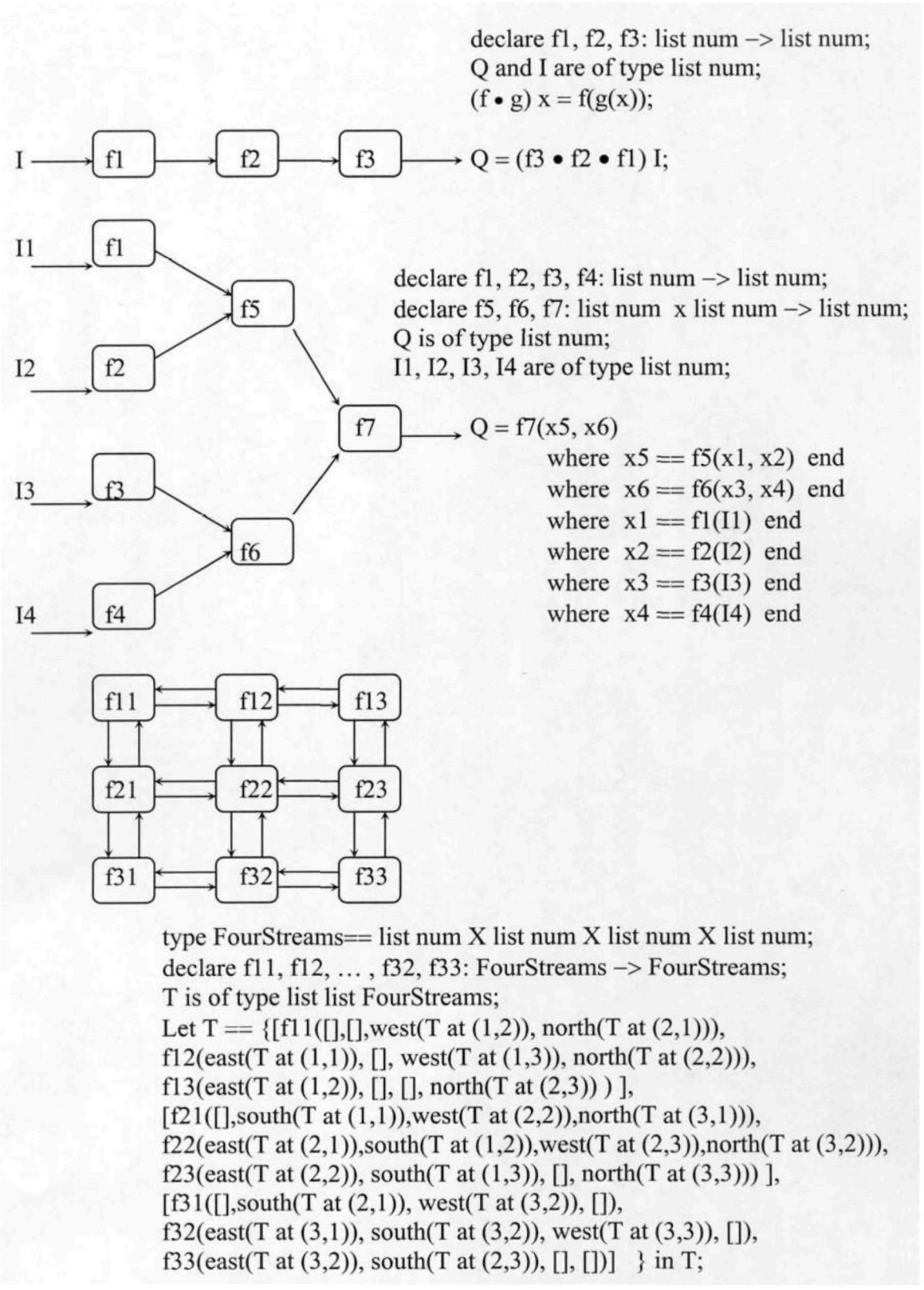

Figure 3. Functional representation of static architectures.

Load Distribution

Load distribution seeks to improve the performance of a distributed system, usually in terms of response time or resource availability, by allocating workload amongst a set of cooperating processors (Evans 1985, Francis 1998, Bilas 2001). In general, load distribution can be classified as the following:

- **Static Load Distribution** assigns tasks to processors probabilistically or deterministically, without consideration of runtime events. This approach is both simple and effective when the workload can be accurately characterized and where the scheduler is pervasive, in control of all activity, or is at least aware of a consistent background over which it makes its own distribution. Problems arise when the background load is liable to fluctuations, or there are tasks outside the control of the static load distributer.
- **Dynamic Load Distribution** is designed to overcome the problems of unknown or uncharacterisable workloads, non-pervasive scheduling and runtime variation; any situation where the availability of processors, the composition of the workload or the interaction of human beings can alter resource requirements or availability. Dynamic load distribution systems typically monitor the workload and processors for any factors that may affect the choice of the most appropriate assignment and distribute jobs accordingly. This very difference between static and dynamic forms of load distribution, is the source of the power and interest in dynamic load distribution.

The essential objective of load distribution is the division of workload amongst a cooperating group of processors. This objective may be fulfilled with varying degrees of fineness, the exact choice of which, depends on the environment and architecture of the parallel system (Hirschberg 1978, Kaplan 1994, Martel 1999). Load distribution is usually described as either load balancing or load sharing. We adopt two concepts in load distribution that are used in the strictest sense to describe the degree to which workload is distributed, and introduce a third concept to describe the middle ground (Roosta 2001).

- **Load Balancing:** Load balancing attempts to ensure that the workload on each processor is within a small degree (or balance criterion) of the workload present on every other processor in the system.
- **Load Sharing:** Load sharing attempts to ensure that the workload only be placed on idle processors, and can be viewed as a processor is either idle or busy.
- **Load Levelling:** Load levelling occupies the ground between the two extremes of load sharing and load balancing. Rather than trying to obtain a strictly even distribution of load across all processors, or simply utilizing idle processors, it seeks to avoid congestion on any one host.

In general, Load sharing, levelling and balancing define a continuum form of coarse to a fine distribution of workload, and seek to distinguish different load distribution schemes.

The objective of this research lies entirely within the domain of dynamic load balancing. For brevity, we will take the more general term of load distribution to stipulate only the dynamic form.

METHODS

In this paper, a high-level application specification is combined with a target architecture specification to produce a specification of the problem on the architecture. In the case of a static architecture, we unfold and transform the problem specification until it is in a form where its function structure is isomorphic to that of the static architecture representation. The functions for each processor on the architecture are then abstracted and compiled to machine code. For a dynamic architecture the specification of the problem is cast in terms of what answer messages are required in response to some input messages. Transformations are carried out to remove redundant calculations by introducing additional message-passing. This allows processors that require intermediate results calculated by another processor to receive them in a message rather than recalculate the contents of the message locally. The transformation ends when any need to access global data in the specification has been transformed out and an efficient algorithm has emerged. An overview of this process is illustrated in Figure 4.

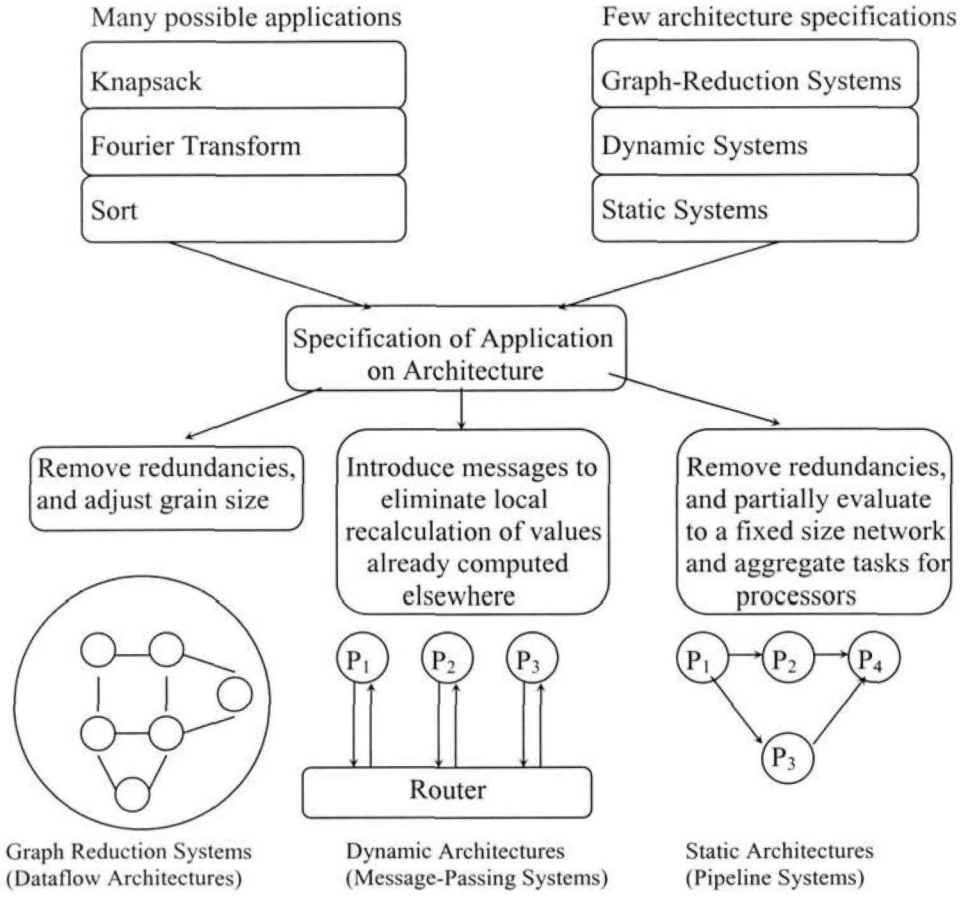

Figure 4. Methodology overview.

Sorting Problem

We have chosen to use sorting as an example and show how to use the methodology to derive parallel sorting algorithms for static (pipeline) and dynamic (message-passing) architectures. In general, a sorted list is a permutation of the original list that is in order

and preserves the relative ordering of equal elements (Akl 1985, Bitton 1984, Evans 1985, Rinard 1999, Martel 1999, Roosta 2000). Thus the position in the sorted list of elements X_j (which is in position j in the unsorted list) is 1 plus the number of elements smaller than X_j plus the number of elements equal to X_j that are to the left of it in the unsorted list. We specify sort in terms of the function Posn which returns the position of an item (the first argument) in a list (its second argument). The first item of a list is in position 1.

$$\text{Posn}(X_j, \text{sort}[X_1, \ldots, X_n]) = 1 + \#\{ X_i < X_j \mid 1 \le i \le n\} + \#\{ X_i = X_j \mid 1 \le i \le j\}$$

We intend to transform this specification into a parallel algorithm for a pipeline architecture and into another parallel algorithm for a dynamic-message-passing architecture.

Transformation To Pipeline Architecture

We intend to pipe the N items to be sorted through the pipeline and to emerge them at the other end in sorted order, smallest item first. In this case, it will take O(N) time to pipe the elements through pipe, for a near optimal algorithm we need a pipe of length O(log N). thus, we have the following architecture-specific problem specification:

$$(f_{O(\log N)} \ \ldots \bullet f_3 \bullet f_2 \bullet f_1) = \text{Sort X}.$$

Our objective is to derive the functions ($f_1, \ldots, f_{O(\log N)}$). Unfolding the specification of sort gives:

$f_{O(\log N)} \ \ldots \bullet f_3 \bullet f_2 \bullet f_1 \text{ X} = [(X_j \mid \text{Posn}(X_j, \text{Sort X}) = 1), \ \ldots, \ (X_j \mid \text{Posn}(X_j, \text{Sort X}) = N)]$.

Consider the final elements of the pipe, $f_{O(\log N)}$. The first thing it does is to produce the smallest item. As we are doing comparison based sorting the smallest item in the list is determined as the result of a comparison between two items. Before this comparison there are two elements that are contenders for smallest and afterwards there is only one. The situation before the smallest-element-determining-comparison is something like the following:

$X_i < X_j < X_k \ldots \qquad 1 \le i \le N, 1 \le j \le N, 1 \le k \le N, i \ne j \ne k \ldots$

and $X_p < X_q < X_r \ldots \qquad 1 \le p \le N, 1 \le q \le N, 1 \le r \le N, p \ne q \ne r \ldots$

In this case, the smallest element is either X_i or X_p and one comparison will determine the smallest. Suppose X_i was the smallest. The next smallest element is then either X_p or X_j. We are merging two sorted lists. Clearly the pipeline architectural specification of sort suggests that a mergesort is suitable for use with the pipeline architecture. So we now simply have to map a recursive mergesort onto a pipeline. This can be done as shown in Figure 5. Using a standard type transformation, we transform the function TreeStage, that maps the data at one tree level to the next, to a function PipeStage that maps the data at the input of one pipe stage to its output. The data at each

tree level can be represented as a list(list num); for example [[3,7], [2,8], [1,6], [4,5]] represents the output of the four vertical merges.

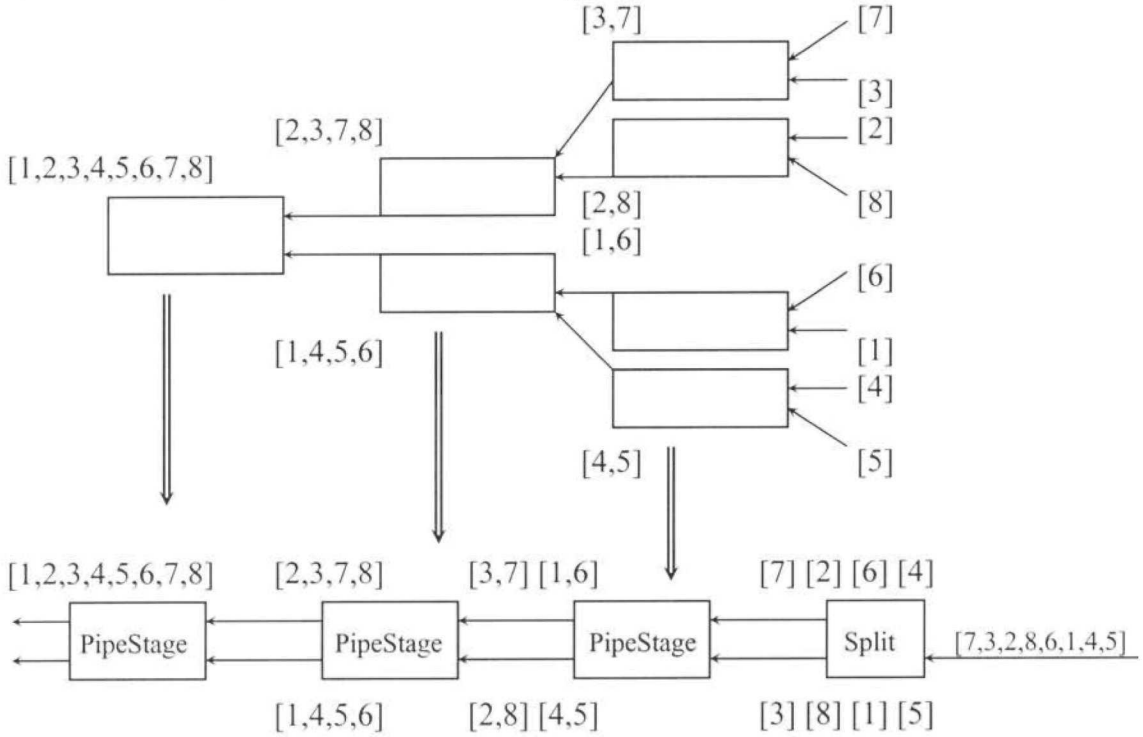

Figure 5. Mapping of tree Mergesort to Pipeline Mergesort.

The data at each pipe stage can be represented as a (list list num X list list num); for example ([[3,7], [1,6]], [[2,8], [4,5]]) represents the output of the first PipeStage function.

The structure of the tree can be represented by defining a function BuildTree that connects together layers of the tree defined using TreeStage. In the following definitions the type-variable alpha will take on the type list num and f will be instantiated to merge.

```
declare   BuildTree: (alpha× alpha–>alpha) × list alpha–>alpha;
          BuildTree(f, xs) <= BuildTree(f, TreeStage(f, xs));
          BuildTree(f, x::y::[ ]) <= f(x, y);
declare   TreeStage: (alpha× alpha–>alpha) × list alpha–>list alpha;
          TreeStage(f, xs::ys::rests) <= f(xs, ys) :: TreeStage(f, rests);
          TreeStage(f, [xs]) <= [xs];
          TreeStage(f, [ ]) <= [ ];
```

It can be easily verified that mergesort is equivalent to:

```
mergesort(xs) <= BuildTree(merge, map(lambda y => [y], xs));
```

where *lambda* represents an anonymous function and *map* is the usual higher-order function that applies a function (the first argument) to each element of a list (the second argument):

```
map(f, [ ]) <= [ ];
map(f, X::rest) <= f(x)::map(f, rest);
```

A data type transformation can be used to synthesis the function PipeStage that maps the input of one stage of the pipe to its output, as shown in Figure 6.

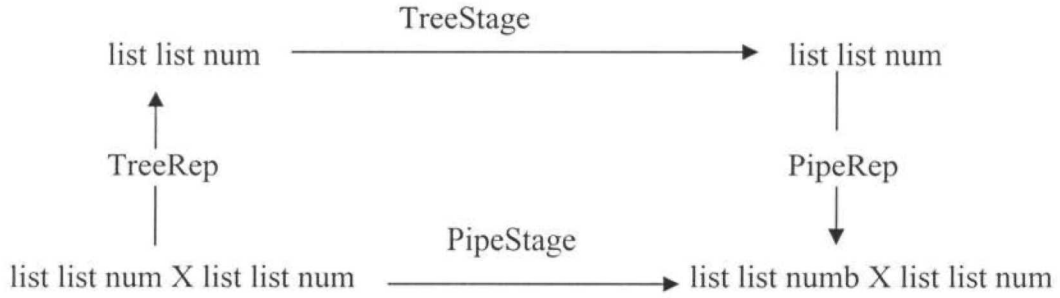

Figure 6. Data type transformation to synthesis. PipeStage: PipeStage = PipeRep(TreeStage(TreeRep)).

The function PipeRep converts data in a layer in the tree to the corresponding layer in the pipe and the function TreeRep converts data in a layer in the pipe to that in the corresponding layer in the tree. PipeStage has the same effect as TreeRep followed by TreeStage followed by PipeRep. In the transformations below PipeStage has been defined so that it takes the function to be performed on the data (i.e. merge in this case) as an extra parameter. This is to illustrate that the transformation will not only work for mergesort but for any divide and conquer algorithm that can be partially evaluated to a tree algorithm in which the amount of work at each level of the tree is constant. The use of higher order functions in this manner will allow libraries of standard transformations to be built up and thus will enable considerable computer assistance with mapping of specifications onto architectures.

```
declare    PipeStage: (alpha×alpha–>alpha)×list alpha×list alpha–>
                list alpha)×list alpha;
           PipeStage(f, xs, ys) <= PipeRep(TreeStage(f, TreeRep(xs, ys));
declare    PipeRep: list alpha–>list alpha×list alpha;
```

PipeRep is the function that converts the tree layer to the corresponding pipe layer.

```
           PipeRep(xs) <= (odds xs, evens xs);
declare    odds: list alpha –> list alpha;
           odds x::y::rest <= x::odds rest;
           odds [x] <= [x];
           odds [ ] <= [ ];
declare    evens: list alpha –> list alpha;
           evens x::y::rest <= x:: evens rest;
           evens [x] <= [x];
           evens [ ] <= [ ];
declare          TreeRep: list alpha×list alpha –> list alpha;
```

TreeRep is $PipeRep^{-1}$, i.e. the function that converts the pipe inputs to the corresponding tree inputs.

```
TreeRep(x::xs, y::ys) <= x::y::TreeRep(xs, ys);
TreeRep([ ], [ ]) <= ([ ], [ ]);
```

TreeRep is only meant to work for list of length 2^n.

Instantiation

```
PipeStage(f, x1::x2::xs, y1::y2::ys)
<= PipeRep(TreeStage(f, TreeRep(x1::x2::xs, y1::y2::ys)));
```

Unfold TreeRep

```
<= PipeRep(TreeStage(f, x1::y1::x2::y2::TreeRep(xs, ys)));
```

Unfold TreeStage

```
<= PipeRep(f(x1, y1)::f(x2, y2)::TreeStage(f, TreeRep(xs, ys)));
```

Unfold PipeRep

```
<= (f(x1, y1)::rest1, f(x2, y2)::rest2)
where     (rest1, rest2) == PipeRep(TreeStage(f, TreeRep(xs, ys)));
```

Fold PipeStage

```
<= (f(x1, y1)::rest1, f(x2, y2)::rest2)
where     (rest1, rest2) == PipeStage(f, xs, ys);
Thus
PipeStage(f, x1::x2::xs, y1::y2::ys)
<= (f(x1, y1)::rest1, f(x2, y2)::rest2)
where     (rest1, rest2) == PipeStage(f, xs, ys);
```

This is the function that each of the processors in the pipe needs to run, with f instantiated to *merge*. It takes O(N) time on O(log N) processors for a list of length N to be sorted and thus the synthesized pipeline mergesort is optimal. In this case, the synthesis did not exploit any particular properties of mergesort and is general divide-and-conquer algorithm to pipeline transformation providing the divide-and-conquer tree contains equal amounts of work at each level of the tree.

Transformation to Message-Passing Architecture

Transformation to a dynamic-message-passing architecture is achieved by reasoning about the set of messages passed between the processors (Roosta 2003). The main transformation tools are free-message-instantiation for introducing new messages and message-folding which enables a value to be used from an incoming message instead of recomputing it locally. The transformation is achieved by introducing rules which state which messages arise in response to which other messages. The first rule states what initial messages start the calculation off and what answer messages must be produced in response.

Architecture Specification

We start the sort with one record per processor and sort the records with respect to the enumeration of the processors. Thus, the smallest item is moved to processor **a** (the lowest numbered processor carrying out the sort), the next smallest item to processor **a+1**

and so on up to the largest item which is sent to processor **a+N−1**. In this respect, processor **a+j**, which has record $\mathbf{X_j}$ to begin with, wishes to calculate **Posn($\mathbf{X_j}$, Sort X)** and send its record to processor **a+Posn($\mathbf{X_j}$, Sort X)−1**. We can send a continue sort message MSG(j, CS(X_j, a, a−1+Length(X), I, X) continuing one of the N items X_j to be sorted (and other parameters to enable us to write a workable specification), to each processor a+j | 0≤j≤N−1 and in response to it, the processor must send out an answer message MSG(a+Posn(X_j, Sort[X_1, . . . , X_N]))−1, ANS(X_j) to the processor that needs to receive X_j at the end of the sort.

Rule 1. For all i ≥ 1 (i is an integer used to disambiguate messages from different recursive calls to sort)

MSG(a+j, CS(X_j, a, a−1+Length(X), i, X)) ∈ Messages 0 ≤j≤Length X−1⇒

MSG(a+ Posn(X_j, Sort X) − 1, ANS(x)) ∈ Messages

where Posn returns the position of an item in a list (numbered from 1) and Messages denotes the set of all messages that exist in the evaluation of QuickSort. This rule, which expresses a property of the messages, operationally implies that when processor a+j receives a message MSG(a+j, CS(X_j, a, b, i, X))it is responsible for ensuring that a message MSG(a+Posn(X_j, Sort X), ANS(X_j)) is produced. This is because only processor a+j is aware of the existence of messages whose destination is a+j, and thus it must make rule 1 hold. Rule 1 has a base case, which is when only a single item is being sorted. In this case j=0, a=b and Posn(X_j, Sort X) =1.

Justification: We can start Rule 1 to specialize a dynamic-message-passing architecture specification to sort a list X=[X_1, . . . , X_N] to give a list Y=[Y_1, . . . , Y_N] on processors numbered k=a to b, b=a+N−1. Processor k receives X_k initially and receives Y_k at the end of the algorithm.

DMPASort(X, a, b) = [Y_1, . . . , Y_N], where

MSG(k, ANS(Y_k)) ∈ Messages

Messages = {MSG(a+j, CS(X_{a+j}, 0, N−1)) | 0≤j≤N−1} ∪
{P_k(Filter(k, Messages), 1) | 0≤k≤N−1}

Filter(k, MS) = {m ∈ MS | m = MSG(k, __) }

P(MessagesLn, i) =

Let {MSG(a+j, CS(X_j, a, b, i, X)) } ∪ OtherMessagesLn = MessagesLn in

if (a = b)

then {MSG(k, ANS(X_j))}∪P_k (OtherMessagesLn, i+1)

else FreeMessagesOut∪{MSG(destination, ANS(X_j))}∪P_k (OtherMessagesLn, i+1)

where destination = a+Posn(X_j, Sort X) − 1.

The last parameter (i) of P_k is equal to the number of the iteration of the current call to P_k and is incremented on each new recursive call to P_k. It is used to disambiguate messages from different recursive calls. The initial program contains the free variable FreeMessagesOut in the message stream emerging from P_k; any value of FreeMessagesOut that is consistent with Rule 1 provides a correct specification for sort; for example an empty set. To prove that this specification satisfies Rule 1, messages can be instantiated to

MSG(a+j, CS(X_j, a, a−1+Length(X), 1, hX)) | 1≤j≤Length X
and the program code can be unfolded until
MSG(a + Posn(X_j, Sort X) − 1, ANS(X_j)) | 1≤j≤Length X
appears in the messages as well. In this case, the function P_k relies on access to the whole of the list to be sorted (X) and this is initially present as the last parameter of the CS message sent to processor k, but X will be removed during the transformation. Moreover, each processor individually calculates the position of its item in the final list and sends out a corresponding answer message. Clearly this is not a very efficient parallel algorithm as each processor duplicates all of the sorting work. The aim of the following section (transformation) is to remove this redundancy, replacing it by inter-processor communication whereby useful results computed by one processor are transmitted to the processor that needs them.

Transformation Justification

Consider processor a+j, in which it is trying to move its record X_j to a processor which is higher numbered that the destination processors of all items less than X_j and all items equal to X_j which were originally on a lower numbered processor than X_j. processor j can use EQN 1 (the sort specification in terms of position) to calculate the processor to which its time should be sent. There are two summations and n comparisons in EQN 1 and since the records are distributed across the processors, interprocessor communication is required to carry them out. In order to carry out the comparisons, the value of the item on processor a+j is required by all processors k, a≤k≤b. this item can be broadcast to all processors by processor a+j in O(1) time if a broadcast mechanism is available (as it is, for example, on the Connection Machine) or in O(log N) time using a tree connection of processors. Comparison of processor a+j's item with everyone elses takes O(1) time in parallel and the summations can be done in O(log N) by employing a binary tree connection of the processors. Processor a+j can then send its item to its final destination. The other processors could carry out similar calculations to those of processor a+j and then send their items to the appropriate destinations (this would carry out an enumeration sort with many duplicated calculations). Alternatively the information gathered by processor a+j can be reused by the other processors. Each processor knows whether its item is less than, equal to or greater than the item on processor a+j, X_j. Clearly all items greater than X_j must be sent to a higher numbered processor than X_j's destination and items less than X_j must be sent to a lower numbered processor than X_j's destination.

Without further calculation, the processors do not know exactly which processor to send their items to, but suppose, as an initial approximation, the items were just sent to an appropriately numbered processor (compared with the destination processor for X_j) preserving their original ordering. On processors a to a+#{ $X_i < X_j$ | $1 \le i \le N$} − 1 the original conditions of a sort would have been recreated for items greater than X_j. performing these two sorts and moving items equal to X_j into the remaining processors preserving their original ordering would clearly be the basis for a parallel quicksort with X_j as the pivot element as shown in Figure 7.

Extra message passing is introduced by instantiating the free variable FreeMessagesOut in the specification of P_k. For example, suppose the set of messages already contains the message M1= MSG(Dest1, Contents1). To introduce some message

M2= MSG(Dest2, Contents2) into the set of messages, a new rule is added which states that M1∈ messages ⇒ M2∈ messages. Processor Dest1 (which received M1) is then charged with ensuring that M2 appears. This is achieved by instantiating P_{Dest1}'s free variable FreeMessagesOut to M2∪FreeMessagesOut2. Processor Dest2 will then receive Content2 in a message. If Contents2 appears as a sub-expression in the body of P_{Dest2}, for example in a where clause, its calculation can be replaced by the contents of the message. The message is extracted from MessagesIn using a let-clause.

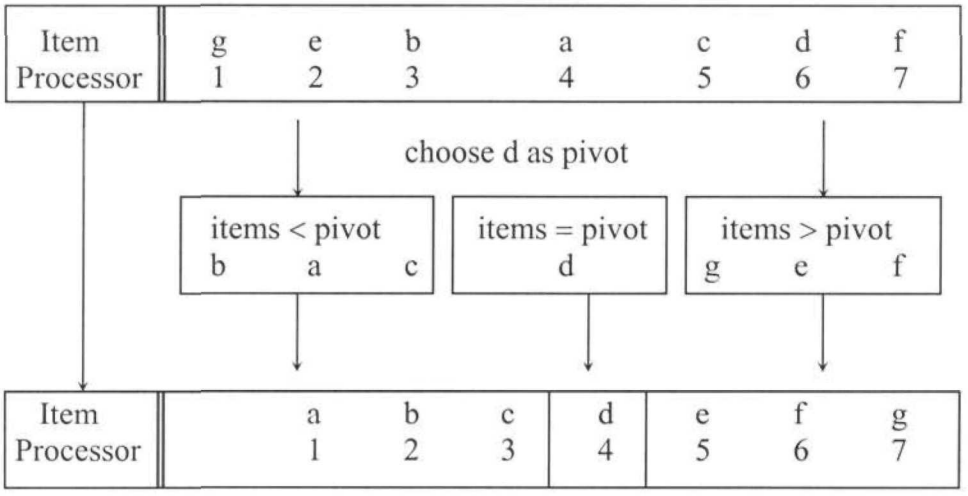

Figure 7. Parallel QuickSort.

For example, in the body of the message recipient, we can replace the expression E(x) **where** x=y+z with **let MSG(k, ValueOfxIs(x))∪Rest=MessagesIn in E(x)** providing we have introduced the message 'ValueOfxIs' by instantiating FreeMessagesOut of some other processor. In this way a novel parallel quicksort can be formally synthesized. For the full mathematical synthesis the reader is referred to another paper (Martel 99). The operation of the synthesized algorithm is as follows. Consider sorting the first seven letters of the alphabet on seven processors numbered 1 to 7 initially organized as a depth first numbered tree as shown in Figure 8.

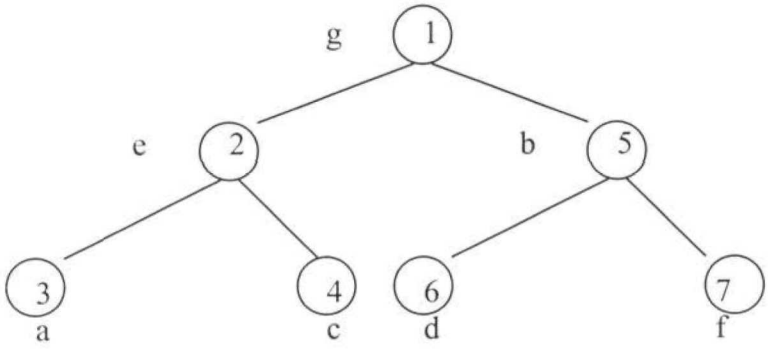

Figure 8. Depth first numbered tree with items to be sorted.

The first stage of the algorithm requires the processors to agree on a pivot element which they will all use. The scheme in Figure 9 can be used to find a pivot near the mean of the elements. Figure 9 shows how each processor (except the leaf processors) receives a triple from its children containing the best pivot so far, the sum of the elements so far, and the number of elements so far. For the purposes of summing the values of the items to be sorted, the letters of the alphabet have been assigned values as follows: a=1, b=2, c=3, d=4, e=5, f=6, and g=7.

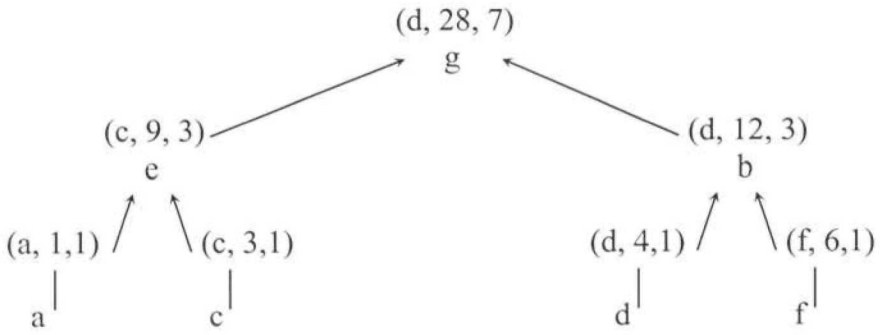

Figure 9. Calculate pivot.

The processor adjusts the sum and number of elements so far so that its own element is included and sends these to its parent together with the new best pivot so far. The new best pivot so far is the one of the three elements known to the processor that is nearest to the mean so far. For example, processor 2 which contains e chooses c as the best pivot to send to processor 1, because c is closer in value to 9/3=3 than a or e. the pivot can be broadcast down the tree in O(log N) time as illustrated in Figure 10. The processors each compare their item with the pivot and as shown in Figure 11, produce a triple:

- (1,0,0) if the item is less than the pivot.
- (0,1,0) if the item is equal to the pivot, and
- (0,0,1) if the item is greater than the pivot.

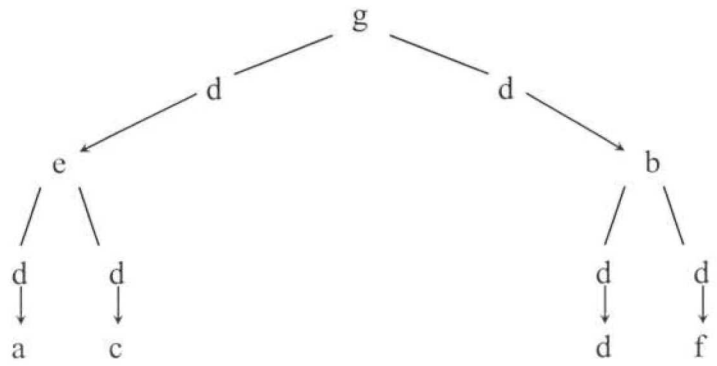

Figure 10. Broadcast Pivot.

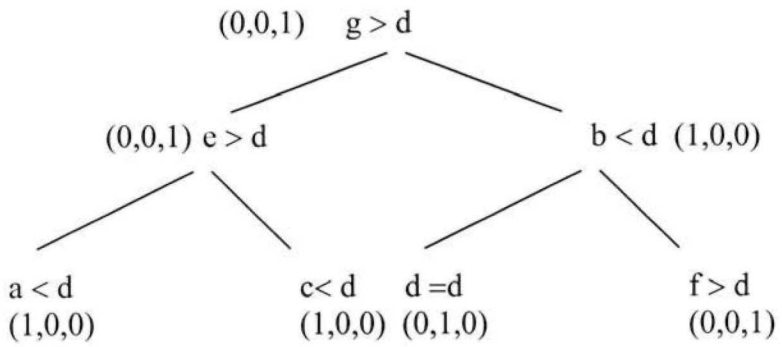

Figure 11. Comparison with Pivot (pivot is d).

We can add up the numbers of items less than, equal to, and greater than the pivot in O(log N) time as shown in Figure 12. The answer (3,1,3) indicates than three items are less than the pivot and will be sent to processors 1 to 3, that one item is equal to the pivot and will be sent to processor 4 and that three items are greater than the pivot and will be sorted on processors 5, 6, and 7.

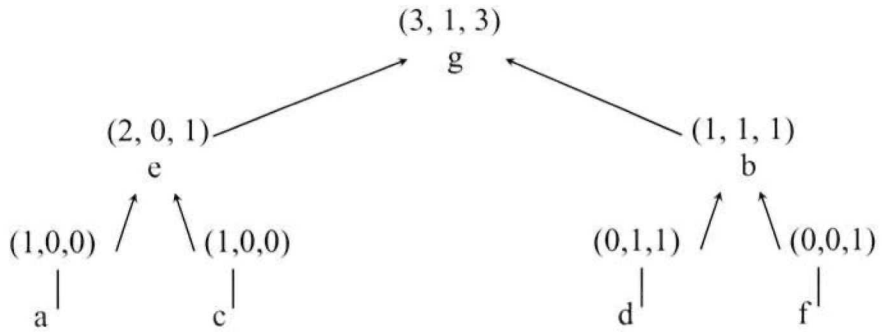

Figure 12. Add the triples.

Let #L be the number of items less than the pivot, #E be the number of items equal to the pivot and #G be the number of items greater than the pivot. Consider some processors **a+j**: if its item is less than the pivot then the destination for the item is **a** plus the number of items on processors **a** to **a+j−1** that are less than the pivot. If its item is greater than the pivot then the destination is #L+#E+(the number of items greater than the pivot on processors lower numbered than j). If its item is equal to the pivot then the destination is #L+(the number of items equal to the pivot on processors lower numbered than j). the root processor (in this case processor 1; the one with g on it) sends the g to the lowest numbered available processor sorting items greater than the pivot (processor 5). It then informs its left child that the lowest numbered available processor destination for items less than, equal to and greater than the pivot are (1,4,6) respectively. The root processor known that its left subtree contains (2,0,1) items less than, equal to and greater than the pivot respectively, and that its item is greater than the pivot and thus uses one destination processor for items greater than the pivot. It therefore informs its right child that

(1,4,6)+(2,0,1)=(3,4,6) are the destination processor numbers available to the right child. Each child uses the same technique to inform its children of the available processors as shown in Figure 13. In Figure 13 each processor (except the root) receives a triple from its parent, adds its pivot comparison triple (from Figure 11) and sends the result triple to the left child. It then adds the triple received from the left child in Figure 12 and sends the new result to the right child. It does not matter that some of the processor numbers go out of range because they won't be used. For example, (4,**5**,7) is sent to the processor which has **f** on it and it does not matter that the 5 is out of range because only the 7 will be used. The sort continues separately for the items less than and greater than the pivot as illustrated in Figure 14.

The sort ends when the number of processors in each new sort group is one. At this point processor a+i will contain the ith smallest element because the algorithm continually sends smaller items to lower numbered processors.

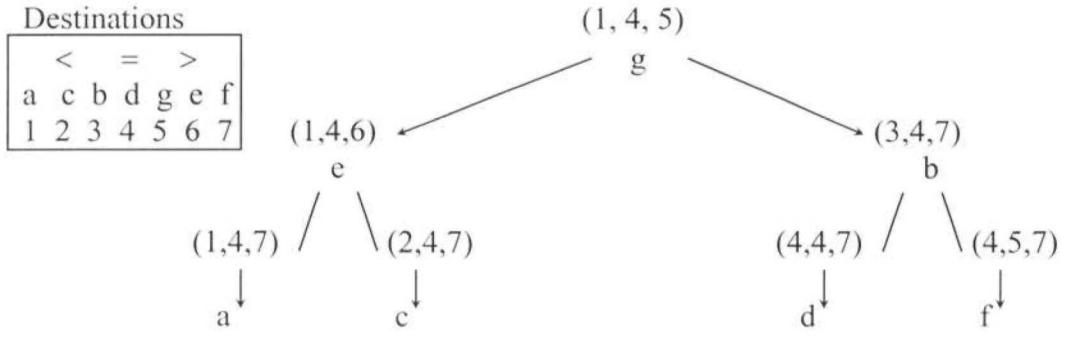

Figure 13. Determine destination messages.

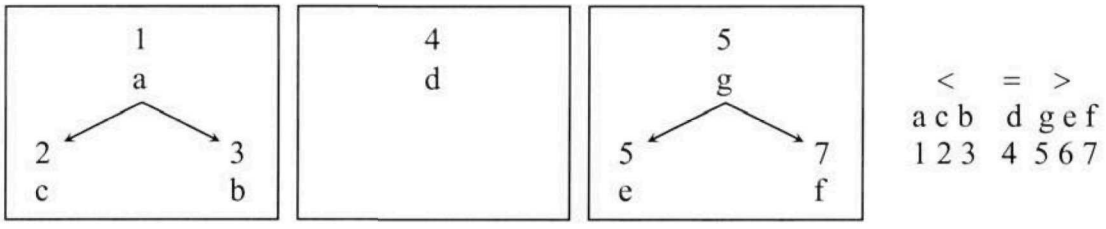

Figure 14. Continue sort using new trees.

CONCLUSIONS

We presented a new methodology for systematically synthesizing programs for various parallel architectures. The key contributions of this paper are:

- Developing a design approach that machine independent issues such as concurrency and scalability are considered early, and independent aspects such as locality and performance related issues are considered later in the design process.
- Developing a program transformation mechanism that supports static and dynamic parallel architectures.
- Developing a basic strategy for resource management on a distributed system with a solid theoretical basis and proven experimental validity.
- Developing a program transformation mechanism that can be used to derive a parallel algorithm that automatically exploits the implicit parallelism in a functional language program.

REFERENCES

Akl S. G. 1985. Parallel Sorting Algorithms, Academic Press, New York.

Arpaci-Dusseau A. C. 2001. Implicit Coscheduling:Coordinated Scheduling with Implicit Information in Distributed Systems, ACM Transactions on Computer Systems, Vol. 19, No.3, pp. 283-331

Attie P. C. and Emerson E. A. 2001. Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation. ACM Transactions on Prog. Lang. and Sys., Vol. 23, No. 2, pp. 187-242

Bilas A., Jiang D., and Singh J. P. 2001. Accelerating Shared Virtual Memory via General-Purpose Network Interface Support. ACM Transactions on Computer Systems, Vol. 19, No.1, pp. 1-35

Bitton D. and D. Dewitt 1984. A Taxonomy of Parallel Sorting Algorithms, Computing Survey, Vol. 16, No. 3.

Burns J. and Pachl J. 1989. Uniform Self-Stabilizing Ringles, ACM TOPLAS 11, 2, pp. 330-344.

Chaudhuri S., Herlihy M., Lynch N., and M. Tuttle. 2000. Tight Bounds for K-Set Agreement. Journal of the ACM, Vol. 47, No. 5, pp. 912-943.

Connection Machine model CM-2 technical summary. 1987. Thinking Machines Corporation, Technical Report Series, HA87-4.

*** defence ****Synthetic theater of war. 2002. Defence Advanced Research Projects Agency. http://stow98.spawar.navy.mil/

Deminet J. 1982. Experience with multiprocessor algorithms. IEEE Trans. Comput., C-31, 1982, pp.278-288.

Diniz P. and M. Rinard. 1999. Eliminating Synchronization overhead in Automatically Parallelized Programs using Dynamic Feedback, ACM Trans. on Comp. Sys., Vol 17, No. 2, pp. 89-132.

Dymond P. W. and Ruzzo W. L. 2000. Parallel RAMs with owned Global Memory and Deterministic Context-Free Language Recognition, Journal of the ACM, Vol. 47, No.1, pp. 16-45.

Evans D. J. and Y. Yousif. 1985. Analysis of the Performance of the Parallel Quicksort Method. BIT 25, pp. 106-112.

Francis R. S. and I. D. Mathieson. 1998. A benchmark parallel sort for shared memory multiprocessors. IEEE Trans. Comput., 37, pp. 1619-1626.

Gehani N. 1984. Broadcasting Sequential Processes. IEEE Transactions on Software Engineering SE-10, 4, pp. 343-351.

Greenbaum, A. 1989. Synchronization Costs on Multiprocessors. Parallel Computing, Vol. 10, North-Holland, PP. 3-14

Hasselbring W. 2000. Programming Languages and Systems for Prototyping Concurrent Applications. ACM Computing Surveys, Vol. 32, No.1, pp. 43-79.

Hirschberg D. S. 1978. Fast Parallel Sorting Algorithms, Commun. ACM, No. 21, pp. 657-666.

Kaplan J. and M. L. Nelson. 1994. A Comparison of Queuing, Cluster and Distributed Computing Systems. NASA Langley Research Center.

Keleher P. J. 2000. A high-Level Abstraction of Shared Accesses. ACM Transactions on Computer Systems, Vol. 18, No.1, pp. 1-36.

Kwok Y-K. and Ahmad I. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. ACM Computing Surveys, Vol. 31, No.4, pp. 406-471.

Lambright. 2002. Distributing object state for networked games using object views, Game Developer, 9(3), 30-39.

Lim B. H. and A. Agarwal. 1994.Reactive Synchronization Algorithms for Multiprocessors. In Proc. 6th Inter. Conf. on ASPLOS., ACM Press, pp. 25-37

Martel C. U. 1999. A Fast Parallel QuickSort Algorithm, Inform. Processing Letters, pp. 97-102.

Mendelson A., and Gabbay F. 2001. The Effect of Communication on Multiprocessing Systems. ACM Transactions on Computer Systems, Vol. 19, No.2, pp. 252-281

Moller-Nilsen P. and J. Staunstrup. 1997. Problem-Heap: A Paradigm for Multiprocessor Algorithms. Parallel Computers., Vol. 4, pp. 63-74

Reischuk R. 1999. A New Solution for the Byzantine Generals Problems. Information and Control 64, pp. 23-34.

Rinard M. 1999. Effective Fine-Grain Synchronization for Automatically Parallelized Programs using Optimistic Synchronization Primitives. ACM Trans. on Comp. Sys., Vol. 17, No. 4, pp. 337-371.

Roosta S. H. 2000. Parallel Processing and Parallel Algorithms: Theory and Computation. Springer-Verlag.

Roosta S. 2001. Performance Evaluation Models for Parallel Computers. ACM Transactions on Computer Systems, submitted for publication.

Roosta S. 2001. Implicit and Explicit Synchronization in Parallel Computers. ACM Computing Surveys, submitted for publication.

Roosta S. 2003. Dynamic Networking in Distributed Systems. The 3rd IEEE International Conference on Peer-to-Peer Computing, p. 74-80.

Saks M., and F. Zaharoglou. 2000. Wait-Free K-Set Agreement is impossible: The Topology of Public Knowledge. SIAM J. Comput., Vol. 29, No. 5, pp. 1449-1483.

Tzen T. H., and L. M. Ni. 1993. Trapezoid Self-Stabilizing: A Practical Scheduling Scheme for Parallel Compilers. IEEE Transactions on Parallel and Distributed Systems, 4, 1, pp. 87-98.