Hybrid Dynamic Load Balancing for Distributed-Memory Multicomputers*

Raymond Chowkwanyun and Kai Hwang
Department Electrical Engineering – Systems
University of Southern California
Los Angeles, CA 90089-0781

Abstract

The hybrid load balancing method recently developed by the authors gives applications the properties of portability and scalability. Experimental results are shown for three Gabriel benchmark programs: Boyer, Browse, and Traverse representing the AI paradigms of unification, pattern matching, and searching. For a 16-processor Intel iPSC system, speedups between 8 and 12 were achieved compared with sequential Lisp execution on a single iPSC processor.

1 Hybrid Load Balancing Method

Dynamic load balancing is used as a heuristic to solve the NP complete mapping problem, i.e. the problem of mapping subprocesses to processors in a multicomputer system with distributed memory. The processors in a multicomputer are connected by a network with each processor consisting of a CPU, local memory, and a switch connecting to the network.

Load balancing can be characterized as either sender-initiated [1] or receiver-initiated [2] depending on the way in which process migration is initialized. Process migration always goes from the busy processor to the lightly-loaded processor. Thus the busy processor is the sender while the idle processor assumes the role of receiver. The hybrid load balancing method proposed by the authors [3] copes with rapidly changing run-time conditions by switching individual processors between sender- and receiver-initiated modes.

We measure the internal load of a processor as the length of the ready queue. The external load of a processor is the load information broadcast by a processor to its neighbours.

Serial Lisp programs can be executed in parallel under the hybrid system after the insertion of the operating system (OS) directives SUSPEND and RUN. The example below shows the parallel version of the serial code \( \text{f (g) (h)} \). First \( \text{f} \) is SUSPENDed with the 2 indicating that \( \text{f} \) takes two arguments. RUNning \( \text{g} \) and \( \text{h} \) causes them to be added to the ready queue. Parallel execution occurs if one of them is migrated to another processor. The arguments to RUN of 1 and 2 indicate that \( \text{g} \) and \( \text{h} \) are the first and second arguments of the SUSPENDed \( \text{f} \). A complete discussion of RUN and SUSPEND is given in [4].

\[
\begin{align*}
\text{(SUSPEND 'f 2)} \\
\text{(RUN 'g 1)} \\
\text{(RUN 'h 2)}
\end{align*}
\]

A. Receiver-Initiated Mode

The receiver-initiated mode is based on a drafting protocol proposed by Ni and associates [2]. The idle processors initiate load balancing by requesting work from the busy processors. The external load is obtained by banding the range of the internal load into three bands: light-load, normal-load, and heavy-load according to a threshold and saturation level denoted by \( R \) and \( S \).

A heavy-load processor has internal load greater than \( S \) and is a candidate for migrating processes. A normal-load processor has internal load between \( R \) and \( S \) and does not participate in load balancing. A light-load processor has internal load less than \( R \) and requests work from heavy-load processors using a drafting protocol. The work request is called a draft. The heavy-load processor rejects the draft if its load state has changed to normal-load or light-load and broadcasts its new status to its neighbourhood. All eligible heavy-load processors return their internal load to the drafting processor. The drafting processor will then request a process from the processor with the highest internal load among those responding and a process is then migrated to the drafting processor.

*This research was supported by NSF grant No. DMC-84-21022
B. Sender-Initiated Mode

The sender-initiated mode is based on a gradient method of load balancing by Lin and Keller [1]. External load indicates the number of hops to the nearest idle processor. A processor is considered lightly loaded if its internal load drops below a static systemwide threshold, $T$. If its load exceeds $T$, it takes the minimum external load of its neighbours and increments it by one. The external load is bounded by the diameter of the network to prevent circulating processes. The equation below summarizes the definition of external load.

$$\text{External Load} =$$

\[
\begin{cases}
0 & \text{if Internal Load} \ < \ T \\
\min(\Delta, 1 + \min(\text{External load of neighbours})) & \text{otherwise}
\end{cases}
\]

where $\Delta = \text{network diameter}$

The external load of a processor $P_i$ is zero if its internal load is less than $T$, signalling the rest of the system that $P_i$ is available to take on work. The propagation of external load values provides an implicit routing to the nearest lightly loaded processor.

C. The Hybrid Method

In a very heavily loaded system, the receiver-initiated mode has the advantage of minimising network traffic because the drafting protocol only allows process migration from idle to busy processors. However in a lightly loaded system, the sender-initiated mode has the advantage of immediately beginning process migration as soon as a processor enters a heavily loaded state. There is no necessity to wait for lightly loaded processors to send drafts and to engage in a lengthy protocol before migration begins. We therefore propose a hybrid method using the receiver-initiated mode when system load becomes excessive and using the sender-initiated mode when system load is light.

The hybrid method requires a distributed mechanism to manage the switching between sender-initiated and receiver-initiated modes of operation. The use of distributed control cannot guarantee that the entire system is in either one or the other mode exclusively. Instead, each processor will operate in either sender-initiated or receiver-initiated mode depending on its local environment. All processors are initialized to sender-initiated mode.

When more than the threshold number of a processor's neighbours become heavily loaded, the system becomes congested and the processor switches to receiver-initiated mode. We refer to this threshold as the hybrid threshold, $H$, to distinguish it from the threshold value, $T$, that defines low and heavy load. If the number of heavily loaded neighbours falls below the hybrid threshold, the processor switches back to sender-initiated mode. Processors can service messages and processes received from processors operating under the other mode, providing a smooth interface between the sender-initiated and receiver-initiated parts of the system.

2 Hybrid Load Balancer Software

A macro dataflow execution model forms the basis for implementing the hybrid load balancer. Figure 1 shows the ready queue and suspend heap for a single processor. The components of the hybrid load balancer are shown as boxes with italic labels. The decision maker pops the next process off the ready queue and decide whether to evaluate it remotely or locally. The decision depends on whether the processor is in sender or receiver-initiated mode which is also controlled by the decision maker.

In our architecture, a complete copy of all applications programs is maintained on all processors. Processes are represented by process control blocks (PCB) and it is these PCB's which are migrated rather than the applications code. A complete copying policy represents a tradeoff of memory for lower communications latency. The structure of run and suspend PCB's representing runnable and suspended functions is shown below. The parent ID field is common to both and is shown as a separate display. Only run PCB's are migrated in the hybrid architecture. Suspend PCB's always remain on the processor on which they were created.

<table>
<thead>
<tr>
<th>Run PCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable Form</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Suspend PCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parent ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Processor</td>
</tr>
</tbody>
</table>

When a function spawns child functions, the parent ID points back to the parent process allowing results of remote evaluation to be returned to the appropriate suspended process. PCB's destined for remote
Figure 1: A macro dataflow execution model.

Figure 2: Components of a software load balancer.
evaluation are sent to the *migrator* and are routed to a remote processor for evaluation. The returned values are picked up by the *mailman* and forwarded to *awake* which matches up the value with its parent in the suspend heap.

When all arguments are present the parent is "fired" in macro-dataflow style and returned to the ready queue. If the *decision-maker* decides on local evaluation then the run PCB is sent to the *local dispatcher* and evaluated. This may cause additional functions to be suspended and added to the suspend heap and spawned functions which are added to the ready queue. Results are either sent to the local *awake* or the remote *awake* on which the parent process resides.

Figure 2 shows the architecture of the software design. The identical architecture is duplicated on every processor and there are no client/server relationships between processors. The hybrid load balancer is therefore a true distributed operating system with no central master. This gives the hybrid load balancer the desirable property of *scalability* meaning that even as the number of processors in the system is increased the hybrid load balancer will continue to perform. It has no master component to become a bottleneck that eventually chokes the system.

In Fig. 2, functions are shown in plain boxes with italic labels while data structures are shown in heavy boxes with plain labels. The basic *kernel* consists of a tight loop consisting of the *decision-maker*, *local dispatcher*, and *mailman*. The kernel is entered through an *initializer* which configures the hybrid load balancer according to information contained in a file of topologically dependent functions. In this way we are able to isolate the hybrid load balancer from the topology of the underlying hardware allowing it to be ported to multi-computers of any topology.

The architecture is centered around three data classes: the run PCB, the suspend PCB, and a class of 7 message types. Each data class is contained in a separate *package* which defines that data class and all functions that access that class. Only names declared within the package as *exported* can be accessed by objects outside the package. For example, the run PCB's are stored in the ready queue which obeys a first-in, first-out discipline. All ready queue access functions have been concentrated in the *local-dispatcher* package. Any access to the ready queue must be made using the functions defined in the *local-dispatcher* package. Functions outside the *local-dispatcher* package cannot access the queue directly. The table below summarizes the data classes together with their packages.

<table>
<thead>
<tr>
<th>Package</th>
<th>Data Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Run PCB</td>
</tr>
<tr>
<td>-dispatcher</td>
<td>Suspend PCB</td>
</tr>
<tr>
<td>Awake</td>
<td>Message</td>
</tr>
<tr>
<td>Mailman</td>
<td>Result of evaluation</td>
</tr>
<tr>
<td></td>
<td>Sender-initiated mode load information</td>
</tr>
<tr>
<td></td>
<td>Receiver-initiated mode load information</td>
</tr>
<tr>
<td></td>
<td>Receiver-initiated mode draft</td>
</tr>
<tr>
<td></td>
<td>Receiver-initiated mode response-to-draft</td>
</tr>
<tr>
<td></td>
<td>Receiver-initiated mode final-call</td>
</tr>
<tr>
<td></td>
<td>Kill kernel</td>
</tr>
</tbody>
</table>

The run PCB and suspended PCB data classes have been discussed and [4] has an example of how PCB's can be used to represent programs. All data types in the *mailman* are messages and are passed back and forth between the processors.

Structures of type *result of evaluation* are used to return the results of function evaluation back to the parent process. Instances of this type are produced by *local-dispatcher* and consumed by *awake*. The structure of a *result* is shown below:

<table>
<thead>
<tr>
<th>Result Type</th>
<th>Parent Address</th>
<th>Port ID</th>
<th>Return Value</th>
</tr>
</thead>
</table>

The *parent address* and *port ID* allow *awake* to place the return value in the correct argument port of the correct suspended process.

*Load information* messages for sender-initiated mode and receiver-initiated mode carry the external load information defined in Section 1. The structure of this type is shown below. The *processor id* field identifies the processor whose load information is being broadcast to the neighbouring processors.

<table>
<thead>
<tr>
<th>Load Information Type</th>
<th>Processor ID</th>
<th>Load Value</th>
</tr>
</thead>
</table>

The three receiver-initiated mode types *draft*, *response-to-draft* and *request-process* are used to implement the
drafting protocol defined in Section 1. A new drafting protocol is initiated when the decision maker signals the draft protocol module. This module generates and handles all messages related to the protocol. The structures of the draft, response-to-draft, and request-process messages are shown below. When sending a draft or a request for process it is only necessary to identify the sender of the message. The response-to-draft requires two fields, one to identify the sender and another to hold the internal load information.

### Draft and Request-Process Types

<table>
<thead>
<tr>
<th>Type</th>
<th>ID</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draft</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request-Proc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The kill-kernel message type is used to terminate the kernel when the application has finished executing. The kernel knows when to terminate by the following mechanism. The parent address 0 is reserved for use as the parent of the root process of the application. Therefore, when awake receives a result destined for a parent with address 0 we know that processing has completed. An instance of the kill-kernel type is broadcast to all processors and the kernel shuts down. The kill-kernel consists of an empty dummy field.

## 3 Concurrent Lisp Benchmarking Results

In this section, we present results of running the Gabriel [5] benchmarks Boyer, Browse, and Traverse on an Intel iPSC multicomputer. In these experiments, we used speedup as the performance measure which is calculated by first timing the execution of the serial benchmark program on a single processor of the multicomputer. We then timed the execution of the benchmark using multiple processors to obtain the parallel execution time. The speedup is then obtained by dividing the serial timing by the parallel timing.

### A. The Boyer Benchmark Results

Figure 3 shows the speedups for the Boyer benchmark. The mix of primitive Lisp functions in Boyer is representative of unification. The goal of Boyer is to prove a given logical term correct. Two levels of granularity are exploited. At the coarse level, the compound logical term is broken up into simpler terms which can be unified in parallel. At medium granularity, a simple term is unified in parallel with multiple axioms in the rule base.

With 16 processors, the hypercube performs best, followed by the ring and tree. At this machine size, the hypercube has greater connectivity with each processor having four neighbours whereas in a tree a processor may have between one and three neighbours and in a ring each processor has only two neighbours. The hypercube does not have this advantage for smaller machine sizes and so performance values are similar.

The sender-initiated method can lead to load migration between busy processors when the system is heavily loaded. The penalty for this over-migration becomes greater as the network diameter increases. Therefore we see a more severe flattening off of the performance curve for the pure sender-initiated method for the ring and tree topologies than for the hypercube.

### B. The Browse Benchmark Results

Figure 4 shows the benchmark results for Browse. Browse provides an instruction mix representative of the pattern matching inner loop operation of expert systems. Parallelism is exploited at two levels of granularity. The coarse level involves multitasking 1,200 pattern matching functions. Medium granularity exploits parallelism within each pattern match.

The hybrid method produces the best performance across all topologies and machine sizes. However we noticed a severe flattening of the performance curve for the ring and tree topologies. We attribute this flattening to the uneven pattern of process generation in this program. The hypercube is better able to redistribute the load with its superior connectivity.

### C. The Traverse Benchmark Results

Traverse is graph traversal program representative of the AI paradigm of searching. The serial version was parallelized so that 250 graphs are traversed in parallel. In addition, all processors cooperate on the traversal of any given single graph. As revealed in Fig. 5, Traverse shows the least speedup of all the benchmarks. We attribute this to the possibility of duplicate work occurring during the execution of this program because multiple processors traverse the same graphs in parallel. The processors exchange information about which graph nodes have been visited but multiple processors can still visit the same graph node.

The iPSC running the hybrid system is shown to be scalable for the four benchmarks. The speedup lines in Figs. 3a, 4a, and 5a are linearly scalable with speedup performance in the range 8 to 12. There is sufficient parallelism in the programs, such as 1,200 for Browse, that a hypercube system with 128 processors should have a projected speedup range from 75 to 98.
Figure 3: Results for the Benchmark Boyer
Figure 4: Results for the Benchmark Browse.
Figure 5: Results for the Benchmark Traverse

(a) iPSC hypercube.

(b) Ring topology simulated on the iPSC.

(c) Tree topology simulated on the iPSC.
In all cases the best performance is obtained under the hybrid method of load balancing. The low performance of the receiver-initiated method is attributed to the high overhead incurred during system initialization. Processing begins at a single root processor and must propagate throughout the system. Under the receiver-initiated method, process migration must wait for load information to propagate first since receivers must have knowledge about busy processors before they can begin drafting. During initialization, only the root processor is doing useful work. The receiver-initiated method should be reserved for the most heavily loaded stages of the computation during which its conservative approach can best help to reduce overhead.

The sender-initiated method works well during the initial stages of computation when load is light and the primary need is to propagate processes. During the later stages of computation this method produces excessive process migration as all the processors become heavily loaded and attempt to migrate processes to each other. The ability of the hybrid method to switch to the less expensive receiver-initiated method accounts for its better performance.

4 Conclusions

The hybrid load balancer allows all the processors in a multicomputer to cooperate together on the solution of a single problem. The major features are its dual modes of operation that allow it to adapt to changing system loads and a completely distributed mechanism for deciding which mode should be used. The hybrid load balancer enables Common Lisp programs to be run in parallel with appreciable speedups as demonstrated in this paper. In [4], we also showed that dynamic load balancing is superior to static load balancing for programs with unpredictable run-time characteristics. Furthermore the hybrid method is relatively insensitive to the setting of internal parameters and performs well without need for extensive tuning.

We have focussed on Lisp as applied to AI rather than numerical problems because AI programs generally exhibit more run-time unpredictability and therefore are in greater need of dynamic resource management. However, there is nothing in the design of the hybrid load balancer that restricts it to AI. The hybrid load balancer is also suitable for numerical applications with similarly unpredictable run-time characteristics. Functional languages such as FP and Hope could be candidates for execution under the hybrid load balancer. Modification of the hybrid load balancer to support concurrent Prolog is also suggested for future research.

References


