呼んでも君は振り向いてくれない*

— 並列論理型言語処理系 KLIC の移植経験から —

近山 隆† 藤瀬 哲朗‡
東京大学 三菱総合研究所

概要

並列論理型言語 KL1 の処理系 KLIC を種々の計算機システム上に移植した経験を報告し、それに基づいて現状の多くの計算機システム、ことにそのオペレーティングシステムの持つ問題を指摘する。

現在用いられているオペレーティングシステムの機能には、そもそも並行処理は意識していても並列処理を意識して設計されていないものが少なくない。最初から並列処理を意図しているはずの並列計算機システム用オペレーティングシステムにおいてさえ、細部においてはこの傾向をひきずっており、静的に計算量の予測が容易な算法の実装には十分な機能を有していても、動的な負荷分散や見込計算を行うような算法には適さないシステムが少なくない。

本稿では KLIC の設計にあたって移植対象システムの機能としてどのようなものを想定したか、どのような機能が実際には欠けていたか、それらをどのようにして補ってきたかについて述べる。また、このような目的のためには本来オペレーティングシステムはどのような機能を提供するのが適切であるかについて論ずる。

*You will never look back to my calls — from experiences of porting a parallel logic programming language system KLIC

†Takashi Chikayama (the University of Tokyo); chikayama@logos.t.u-tokyo.ac.jp

‡Tetsuro Fujise (Mitsubishi Research Institute, inc.); fujise@mri.co.jp

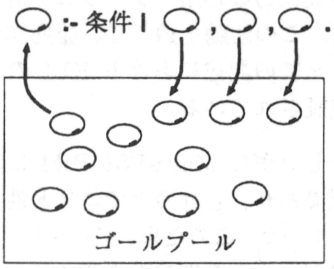

図 1: KL1 の実行モデル

1 はじめに

1.1 KL1 と KLIC

KL1 [10] は、1982 年から 11 年間に渡って行なわれた第五世代コンピュータ (FGCS) プロジェクトにおける諸研究の核となるべき『核言語』として、並行論理型言語 GHC [9] を基に同プロジェクトにおいて設計したプログラミング言語である。

KL1 の実行過程は、ゴール (引数への述語の適用) を書き換えルールを与えるプログラムに従ってサブゴールに書き換えていく過程とみなすことができる (図 1)。この書き換え (リダクション) を複数、同時並列的に行えることが、KL1 実行の並列性の源泉である。書き換えルールの適用条件判定に必要なデータが揃うまで自動的にルール適用を待ち合わせることが、データフローに基づく自動同期機構になっている。このデータフロー同期機構と論理変数の単一代入性のため、非決定的な書

法を用いた場合を除いてゴールからサブゴールへの書き換え順は実行結果を左右しないため、KL1での並列プログラム記述は通常の手続き型言語によるよりもはるかに容易になっている。

FGCSプロジェクトにおいてはKL1の実行専用に設計した並列計算機Multi-PSI [8], PIM [4]の上にKL1言語処理系を作り、その上にオペレーティングシステムPIMOS [3]や種々の実験的な応用ソフトウェア群 [6]を構築した。この活動を通じて、並列知識処理ソフトウェアの記述におけるKL1の高い生産性が実証されている。

しかしながらMulti-PSI, PIMは実験的な専用機であり、設計の上でコスト性能比は度外視していた。このため、プロジェクト終了後にプロジェクトで開発した種々のソフトウェアを普及していくことには困難が予想された。一方で市場にも大規模な構成をとれる並列計算機システムが徐々に現れてきたこともあり、1993年から2年間の第五世代コンピュータの基盤化プロジェクトにおいて、商用のUnixベースの並列計算機システム上で第五世代コンピュータプロジェクトで開発してきたソフトウェアを実行できるKL1言語処理系KLIC[2]を開発した。

KLICはKL1プログラムをいったん言語Cに翻訳してから、ホストシステムのCコンパイラで機械語に翻訳する方式をとっている (図2)。この方式をとることによって、近年高性能化が著しい最適化Cコンパイラを生かし、移植性と高効率を両立した言語処理系を実現した。KLICの逐次処理部の実行速度は、よく用いられている論理型言語向きのベンチマークプログラムにおいて、広く普及した逐次型論理型言語処理系であるSICStus Prolog [1]の生成する機械語コードと比べて約2倍を実現している (SICStus Prolog 2.1版との比較)。

KLICの並列処理系では、逐次処理用とはリンクする実行時システムが異なるものの、まったく同一のオブジェクトコードを用いる。[1]実行時システムも並列処理に関わる基本的な些少のオーバヘッド (割込み時のフラグのチェックなど) を除いては、複数プロセッサ間の通信の頻度・量に比例するような処理を除いて逐次処理用と同一である。このため、通信の少ないプログラムにおいては、最適化した逐次プログラムの処理能力にプロセッサ台数を乗じたものにごく近い性能を示す。

1.2 対象プログラムの性質

扱うデータの大きさが決まれば計算開始前に必要な計算がほぼ正確に予測できるような算法においては、予測に基づいてプロセッサへの処理の割当て (負荷分散) や処理の順序づけ (スケジューリング; 負荷分散と併せて『マッピング』と呼ぶこともある) をあらかじめ決めておくことができる。現在行なわれている科学技術計算における並列処理は、ほとんどがこのカテゴリに入るものである。このような場合は、計算に必要なデータの初期配置や計算途中でのデータの交換時期・量をあらかじめ決定し、プログラム中に指定しておくことができる。

一方、KLICが主対象としている木構造あるいはグラフ構造をなすデータに対する探索を多用する知識処理において総計算量を抑制するには、計算の進行にともなってどのような計算が必要であるかを次第に確定していく動的な算法を用いることが必須である場合が多い。このような算法にあっては、あらかじめ最適なマッピングを決定できないので、ある計算をどのプロセッサでいつ行なうかを計算の途中で判断していく**動的な負荷分散**を行なう必要がある。

また、計算の必要性が確定するまでその計算を開始しない方針では、十分な並列性が得られない場合も少なくない。たとえば探索でOR関係にあるふたつの処理は、一方で解答が

[1]後述する問題のために、一部のプラットフォーム用には並列処理専用のコードを挿入する必要が生じたが、そのオーバヘッドは小さい。

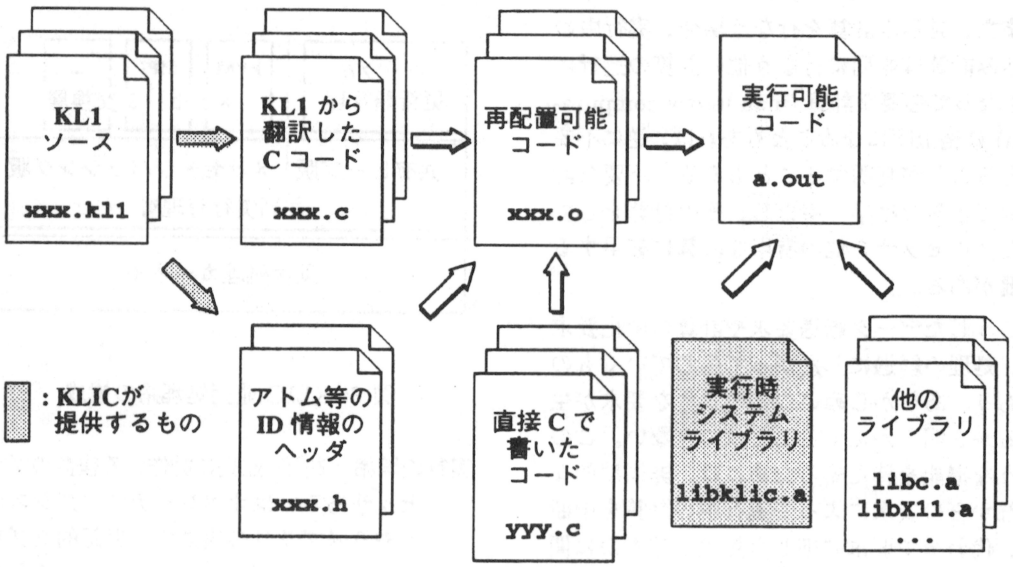

図 2: KLIC のコンパイル方式

得られないことがわかってはじめて他方の計算の必要性が確定する。同様に、AND 関係にあるふたつの処理は一方の解答が得られてはじめて他方の必要性が確定する。計算の必要性が確定はしていないが必要になる可能性が高い場合、利用可能なプロセッサがあれば必要性を見込んで計算を開始してしまう方式をとれば、より並列度の高い処理ができる。こうした処理は**見込み計算** (speculative computation) と呼ばれるが、並列処理性能を向上する代表的な手法のひとつである。[2]

1.3 言語処理系への要請

動的負荷分散を行なう場合には計算対象となるデータの配置も動的に決定する必要がある。しかも、分散する処理の中でデータのどの部分が実際に必要になるかも、計算を進めてみないとわからない場合が多い。たとえば木構造のうちのどの部分木が必要かは、計算の進行とともに木構造をたどっていってはじめて確定し、それが木構造全体からみるとごく一部であることが多い。そこで、必要であることが判明した時点でデータの転送を行なう方式をとることが望ましい。このためには、他の計算処理を実行中のプロセッサにデータ転送を依頼する機能を実現する必要がある。

また、CPU の処理速度に比して転送の遅延がかなり大きい計算機システム (イーサネットで結合したワークステーションクラスタはその典型) においては、転送待ちの間に他の処理 (往々にして優先度の低い処理) を進めることが望ましい。依頼したデータの到着時には、実行中の処理を中断し、到着したデータを必要としていた処理を再開する必要がある[3]。

[2]プロセッサアーキテクチャ分野においても見込み計算を行なうことは少なくないが、ソフトウェアで行なう見込み計算ははるかに大きなまとまりを持った計算を対象とし、その並列性もかなり大きい場合が少なくない。

[3]現在の KLIC ではデータの転送を完全に lazy に行なう方式がデフォルトになっている。この方式は転送量としては最適になるが、転送遅延が問題になる場合がある。そこでプログラムの静的解析によってできるだけ早めに必要なデータを知り、先取りして転送して転送遅延の問題を軽減しようという研究 [11] もなされている。しかし、この場合でも動的に分散する計算について動的データ配置が必要であることに変わりはない。

また、見込み計算を行なう場合、実行中の見込み計算は並列に行なう他の計算の進行にともなって必要な計算 (mandatory computation) に格上げになることもあれば、逆に不要であることが判明することもある。不要な計算はできるだけ早く中止し、その計算をしていたプロセッサを他の有用な計算に転用する必要がある。

こうしたデータ転送要求や計算の中止要求は、処理の経過にしたがって生じていくものであり、あらかじめいつどのような要求が生じるかを明らかにすることはできない。このような機能を持たせるためには、非同期的な外部からの要求によって実行中の計算を中断し、代わって要求に応じた処理を速やかに開始できるように処理系を設計する必要がある。

KLIC ではこうした機能を実現しているわけであるが、これらを互いに異なる機能を持つ並列計算機システムへの移植性を確保しつつ、一方で十分高い効率を維持できるように実現するためには、さまざまな工夫が必要であった。本稿では KLIC ではどのようにして移植性と効率性の両立をある程度でも実現したかについて報告する。その中で、商用の並列計算機システムのどのような機能不足が、より高い効率を実現するために、あるいは処理系の簡潔な構成のために障害となったかについても報告する。

2 KLIC 並列処理系の構成

KLIC の並列処理系の設計にあたっては、アーキテクチャ、オペレーティングシステム、および並列処理のためのライブラリ等を用いる場合はそのライブラリ (以下、これらを含めて**プラットフォーム**と呼ぶ) と独立な部分と、プラットフォームに依存する部分をできる限り分離し、移植性を確保する方針を採った。並列処理系を構成するにあたってプラットフォームの違いが問題になるのは、以下の諸点である。

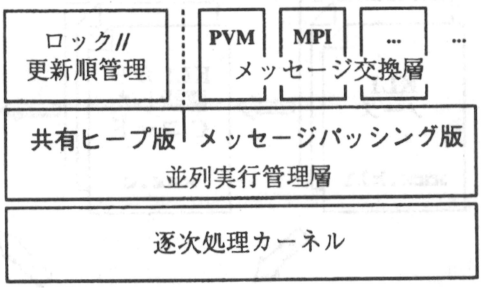

図 3: KLIC 並列処理系の構成

実行の開始・終了 並列に動作する複数のプロセッサあるいはオペレーティングシステムやライブラリの提供する仮想的なプロセッサであるプロセス (以下**ワーカ**と呼ぶ) をどのように初期化して実行を開始させるか、実行終了時にどのようにしてすべてのワーカの実行を終了させるか。これはオペレーティングシステムやライブラリによってかなり大きな違いがある。

通信・同期 プログラム実行の途中で複数のワーカ間でどのようにして通信を行い、また必要な同期を実現するか。これは用いる通信方式によって大きく異なる。

OS の仕様 同じ Unix であっても、入出力やプロセス制御などのオペレーティングシステムの仕様の細部には、個別のシステムごとに微妙な差がある。

システム構成 同じ機能を同じプログラムインタフェースで提供していても、ライブラリなどがどこに置かれているかはシステムごとに異なる。

本稿ではこのうち主として通信・同期の問題について論じる。

KLIC 処理系は並列処理のための通信・同期という観点から見ると以下のような構成要素からなっている (図 3)。

逐次処理カーネル ゴールからサブゴールへのリダクションを単位とした基本的な逐次

実行をつかさどる部分。ユーザのプログラムをモジュール単位にCの関数にコンパイルしてできるコード（以下**モジュールコード**と呼ぶ）と、モジュールコードで処理するには複雑すぎる処理（構造体同士の単一化など）を行うサブルーチン群（以下**補助ルーチン**と呼ぶ）、外部からの割り込み・プロセッサ間通信・ガーベジコレクションなどを担当するサブルーチン群（以下**割り込みルーチン**と呼ぶ）からなる。

並列実行管理層 並列実行のためのデータ管理・メモリ管理・実行管理をつかさどる部分。補助ルーチンや割り込みルーチンから呼び出すルーチン群で、これまでに共有ヒープを用いる版 [5] とメッセージパッシングを用いる版 [7] のふたつの版を用意している。モジュールコードから直接呼び出すことはないので、モジュールコードは同一でよい。

以上の二階層についてはプラットフォームに独立である[4]。これらにプラットフォーム依存部分を付け加えて、実際に並列動作する処理系が完成する。

共有メモリ機構を前提とする共有ヒープ版については、これにロックの機構や、メモリシステムの構成によっては必要となるメモリ内容の更新順管理のためのプリミティブ[5]などが追加されるだけで、プラットフォーム依存部分は大きくない。このため共有ヒープ版の移植性は高いが、高性能な共有メモリを実現するアーキテクチャに高い並列性（たとえば1,000以上）を期待することは難しい。

[4] もちろん共有ヒープ版はなんらかの形での共有メモリ機構が前提であるが。

[5] いわゆる weak consistency しか保証しないキャッシュメモリシステムでは、ふたつのメモリ語X, Yに対してあるプロセッサでX, Yの順で書き込んだものが、他のプロセッサではY, Xと逆順に書き込んだように見えることがある。この問題を避けるためにはXを書き込んだ後にキャッシュの一貫性を保証するための操作を行なう必要がある。通例特殊な1命令を実行するのみであるが、通常のCプログラム中から直接実行する手段はないので、アーキテクチャ依存の機械語命令を生成するマクロを用いている。

メッセージパッシングに基づく並列処理系の場合、データ・メモリ・実行の管理は計算を行うワーカ間でさまざまなメッセージをやりとりすることによって実現している。メッセージパッシング版では、効率を追求する上で以下のプラットフォームに依存する階層を追加する方針とした。

メッセージ交換層 並列実行管理層から呼び出され、実際にメッセージをやり取りする部分。プラットフォームによって効率的なメッセージ交換の方法は異なる。このため、この層はプラットフォームごとにさまざまな版を用意している（表1）。

新たなプラットフォーム上にメッセージパッシング版のKLICを移植するには、基本的にはこのメッセージ交換層のみを作成すればよい。この層で実現する機能は以下のとおりである。

メッセージ送信機能 任意長のメッセージを指定ワーカに送りつける機能。並列実行管理層でバッファにメッセージを構築し、メッセージ交換層にはメッセージ全体の送信を依頼する。したがって、ここではバッファ内に完成しているメッセージを送信するだけでよい。ただし、並列実行管理層は（必要ならバッファの拡張を行ないながら）いくらでも長いメッセージを作るので、送信を依頼するメッセージの最大長は決まっていない。

メッセージ受信機能 任意長のメッセージをバッファに受信する機能。バッファからの取り出しは並列実行管理層で行うので、ここでは受信したメッセージをバッファに転送するだけでよい。送信と同様、受信用のバッファも並列実行管理層が管理しており、必要に応じて自動拡張するので、ここでは無限長のバッファに対して転送するものと考えて差し支えない。

メッセージ到着通知機能 他のワーカからのメッセージの到着は、通常の計算処理中

表 1: メッセージパッシング版 KLIC

標準的ライブラリなどを利用	
TCP:	ワークステーションクラスタなど
PVM:	VPP300 (富士通), SR2201 (日立), AP3000 (富士通), DEC7000 (DEC)
MPI:	Cenju 3 (日本電気), MPICH 稼働システム
共有メモリを利用	
	Sparc/Solaris 並列機 (Sun), PowerChallenge (SGI Cray), Pentinum Pro/Linux (4CPU)
システム独自の機能を利用†	
	AP1000+ (富士通), NCUBE (NCUBE), CM5 (TMC)

†最近の OS 版では動作未確認

にその処理とは同期せずに起こる。一方、メッセージの受信には他の処理と同期を必要とするものが大部分なので、メッセージを受信しての処理は逐次処理カーネルの処理と同期的に行うのが適当である。したがって、メッセージ到着時にはメッセージ交換層からメッセージが到着しその処理が必要である旨を逐次カーネルに通知するのみとし、並列実行管理層経由でメッセージ交換層を同期的に呼び出す方針を採っている。

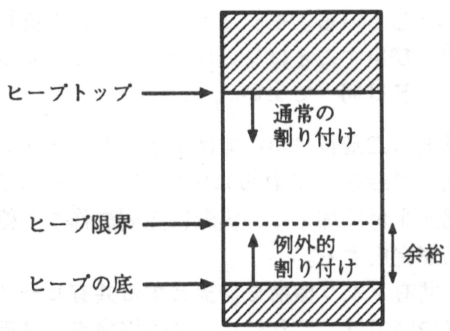

図 4: ヒープ領域の構成

メッセージ到着通知機能の実現のため、KLIC ではメッセージ交換層がメッセージ到着時にメモリ中のフラグを設定し、逐次処理カーネルは実行途中にこのフラグを適宜ポーリングすることによって受信処理の呼び出しタイミングを取る方針としている。

このポーリング処理はヒープ領域のオーバフローチェックと一体になっている。メモリ割り付けが進行してヒープをあるところまで使用したらガーベジコレクションを行なうわけであるが、逐次カーネルではこのオーバフローチェックのためにヒープの限界を示す大域変数を設けている (図 4)[6]。メッセージ交

[6]ヒープを双方向から割り当てているのは、実行時システム中の補助ルーチンを呼ぶ際に、ヒープポインタを渡さずに済ませるためである。普通はメモリ割り付けを必要としないが、例外的な場合だけ割り付けを行なうことがあるような実行時サブルーチンが少なくないのであるが、C 言語ではサブルーチンから効率良く返せる値はひとつだけなので、常にヒープポインタの受渡しをするのは望ましくない。一方、ヒープポインタを大域変数に置いたのではロード/ストアのコストが大きい。

そこで、通常のヒープポインタはレジスタ上にキャッシュできる局所変数とし、補助ルーチン中での例外的な割り付けはヒープの逆側から行なうこととした。常にヒープ限界と実際のヒープの底 (共に大域変数に置く) との間にある程度の余裕を持たせておけば、オーバフ

換層では、メッセージ到着時にメッセージ到着自体を示すフラグを設定すると共に、このヒープ限界を表す変数を小さな値にしてしまう。こうすることによって、次のチェック時に偽のヒープオーバフローが生じるが、その処理ルーチンの中でメッセージ到着を示すフラグをチェックし、フラグが設定されていたらそのための処理をする。

この方式でのポーリングは、もともと必要なガーベジコレクションのためのチェックと一体化しているため、ほとんどオーバヘッドなしに[7]短い検出遅延を実現できる。

3 移植上の問題点と解決法

3.1 想定したメッセージ交換機構

KLIC 処理系を設計するにあたっては、対象とするプラットフォームが概略以下のような機能を提供できるものと想定していた。

- 任意のプロセッサ宛てのメッセージパケットの送信。
- メッセージ到着の割り込みによる通知。メッセージ長に制限がある場合は、メッセージを分割した結果のパケットの到着の通知。
- なんらかの理由で即座にメッセージを送れなかったとき、あるいは、メッセージが長過ぎて全体を送り切れずパケットに分割して送ったときに、送出処理が進行して次の送信が可能になったことの割り込みによる通知。

後二者の通知機能として想定していたのは、SIGIO シグナル (BSD 系 UNIX の場合; SYSV 系では SIGPOLL) のような機能である。すなわち、あらかじめ指定しておいた受信チャネル (IP ソケットなど) に新たなデータが到着し読み出し可能になると (あるいは、送信チャネルのバッファに追加送信のための書き込みが可能になると)、シグナルが生じて登録しておいたハンドラに制御が移るというものである。シグナルハンドラは逐次カーネルの処理とは同期しないので、ハンドラの中で直接メッセージ送受信処理を行うことはできないが、フラグをあげる処理はできる。逐次カーネルとの同期にはデッカーのアルゴリズムと同様の 2 メモリ語を用いた処理が必要であるが、オーバヘッドは小さい。

しかしながら、実際に種々のプラットフォームへの移植を行ってみると、商用のプラットフォームではこれらの機能のすべてを提供するものはほとんどないこと、ことにメッセージ到着や次パケットが送信可能になったことの割り込みによる通知が得られないものが多いことがわかってきた。

3.2 割り込みによる通知の必要性

前述の通り、KLIC では計算をしている最中に他のプロセッサからデータ転送の要求、転送要求の結果などのメッセージが送られてくる場合があり、それに対応して処理しなければならない。このためにはなんらかの割り込みが生起することが望ましい。上述したように、割り込み処理と通常の計算処理との同期は、大域変数中のフラグを用いてかなり効率的に行なえる。

ところが、多くのプラットフォームではこのような割り込みの機能が提供されていない。たとえば、フリーソフトウェアの並列処理用通信ライブラリである PVM や MPI は、通信の受け手がメッセージの到着を予期し、自ら受信体制に入って待ち合わせることを想定した構成になっている。PVM には通信とは別に他ワーカに対してシグナルを送る機能が提供されているが、メッセージを送信した後にシグナルを送ってもシグナルが先着する可能

ローチェックの間にこの余裕以内の割り付けしかしないようにすれば正しく動作する。

[7]限界値を示す大域変数の値を、レジスタ上などにキャッシュしておくことができないこと (C 言語では volatile 属性の指定) だけがオーバヘッドである。実際にオーバフローが起きたのかは、別に保存しておいた真のヒープ限界値と比較して判断するので、余分なガーベジコレクションを行なうことはない。

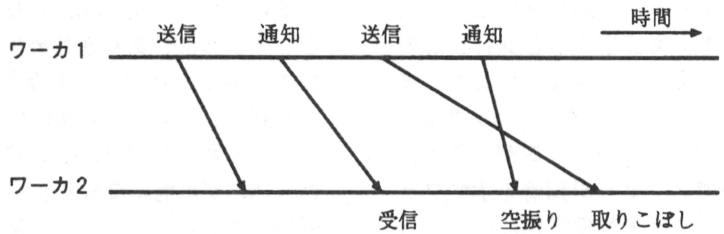

図 5: 通知の追い越しによるメッセージの取りこぼし

性や、複数送ったシグナルが一度だけ到着する可能性がある[8]ので、シグナルの到着だけをメッセージ受信の契機とするのではメッセージの取りこぼしが生じる (図5)。MPI にはこのようなシグナル機構はまったく用意されていない。種々の並列計算機システムの独自の通信プリミティブの多くも、このような機構を備えていないものが多い。

また、多くの通信機構においては一括して送信できるデータ量に上限があるが、KLIC はいくらでも大きな構造体を取り扱うことができるため、いくらでも長いメッセージを送りたくなることがある。このような場合はメッセージを複数のパケットに分割して送る必要がある。しかし、ひとつのパケットの送信後に他の処理をしていると次のパケットの送信が可能になったことを検知できないようでは、他の処理をせずに次のパケットが送信可能になるのを待ち合わせている必要がある。これでは空いている CPU を利用して他の有用な処理を進めることができない[9]。したがって、これも割り込みとして通知されることが望ましい。

この送信可能の割り込みによる通知に至っては、ほとんどのプラットフォームで提供されていなかった。

[8]複数のシグナルがひとつになるのは PVM の問題ではなく、UNIX のシグナル機構の仕様である。

[9]単に送信可能になるのを待っているだけでは、互いに通信し合うプロセッサ同士がこのような状態に入ってデッドロックに陥る可能性があるので、受信処理は行ないながら待ち合わせる必要がある。

3.3 対処法

想定した割り込み生起の機能を持たないプラットフォームへの移植にあたっては、以下のようなさまざまな方策を採った。

- 他ワーカの空間への書き込み
- タイマの利用
- リダクション数に基づくポーリング
- ヒープの人為的縮小

以下にやや詳しく述べる。

3.3.1 他ワーカの空間への書き込み

メッセージ到着シグナルのハンドラでは、結局メモリ中の特定語にあるフラグを立てるだけである。そこで、他ワーカのメモリ空間中の語を直接的に更新する機能が利用できる場合は、このフラグを書き換えてメッセージ送信を通知することも考えられる。

この方法は他ワーカのメモリを直接書き換える機構 (PUT) を持つ並列計算機システム上の版や、共有メモリ並列計算機上で (共有ヒープ方式ではなく) メッセージパッシング方式を用いる版で用いている。

この場合、書き込みは受信の契機を知らせるために用いるだけで、メッセージ自体は別の手段を用いて送ればよいので、スループットの低さも書き込みの遅延の大きさもあまり問題にならない[10]。

[10]とはいっても、1語の更新のためにも1ページ分の

この方式はあまり欠点がないが、このような機能を提供しているプラットフォームは数少ない。なお、メモリ書き込みによる通知がメッセージ送信を追い越さない保証は必要であり、この保証が得られない場合はシグナルがメッセージを追い越す場合と同様の対処が必要になる。

なお、これとよく似た実装方式として、メッセージ到着を待ち合わせる処理専用のスレッドを設け、このスレッドが共有する大域変数を経由で主スレッドにメッセージ到着を通知する方式も考えられる。しかしながら、十分広く普及し安定した実装のあるスレッドライブラリ仕様が見当たらなかったので、KLICでは現在のところスレッドを用いる方式は提供していない。

3.3.2 タイマの利用

メッセージの到着検出にある程度の遅れを覚悟すれば、タイマを用いて定期的なシグナル割り込み (SIGALRM) を起こさせ、その際にメッセージ到着の有無を調べるようにする方法である。シグナル処理のオーバヘッドはかなり大きいのが普通なので、割り込み頻度をあまり高く設定することは望ましくなく、到着検出の遅れとの調整が難しい欠点がある。

シグナルの送り付けは可能でシグナルの追い越しの危険はあるが、追い越さない場合が多い、というプラットフォームもある。そのような場合には、メッセージ送信後にはシグナルも送るようにし、運良く追い越しがなかった場合 (それが大部分であることを期待するわけだが) にはシグナルを契機として受信処理を開始させるようにする。これだけでは追い越し時のメッセージ取りこぼしが生じるので、別にタイマ駆動のポーリングも行なう。この方法では、シグナルの追い越しの頻度が低ければ、タイマ割り込みの頻度を下げてもレスポンス悪化は処理容量の上では大きな問題にならない[11]。

転送が生じるような仮想共有メモリシステムでは、非効率になり過ぎるであろう。

[11] ただし、最大遅延はタイマ割り込みの間隔に依存するので、実時間応用には向かない。

しかしながら、並列プラットフォームには各プロセッサ上で動作するタイマを持たない[12]ものが少なくなく、必ずしもこの方法が利用できるとは限らない。通信ライブラリの規格あるいはその実装によっては、ユーザプログラムでタイマを用いるとライブラリ自身の動作が保証されないものもある。

3.3.3 リダクション数に基づくポーリング

上述の方法がいずれも利用できない場合には、何らかの方法で積極的にポーリングするしかない。しかしメッセージ到着の有無を知るためにはシステムコールを必要とする場合が多く、高い頻度でのポーリングは大きなオーバヘッドを伴う。

KLICでは無限ループする可能性のあるループの最小単位は、親ゴールを子ゴールに書き換えるリダクションである。これは数十機械語命令程度の大きさであり、これを単位としたポーリングでは実行速度を耐えられない程度まで低下させる可能性が高い。そこで、リダクションの回数を数えるカウンタを設け、ある程度回数のリダクション数ごとにポーリングを行う方式を用意した。

この場合の実行時間上のオーバヘッドは、リダクションごとのカウンタのデクリメントと、ゼロになったかどうかのチェックである。命令セットによって異なるが、多くの場合これはメモリ参照、即値の減算、条件分岐、メモリ格納の4命令程度を要する。KLICで1リダクションに要する命令数が数十程度であることを考えれば、この程度のオーバヘッドでも無視できない程度に大きい。カウンタをレジスタ上に保持すれば[13]半分の2命令程度になるが、その分だけ通常の実行に利用できるレジスタ数が減ることになる。

[12] ハードウェアとして持たない、OSが機能を提供していない、提供していてもバグがあって使い物にならない、などさまざまなケースがある。

[13] 実際にはKLICはCで記述しているので、レジスタ上に保持させたい場合は局所変数にキャッシュするように記述し、これをレジスタを用いるかどうかはCコンパイラの判断に任せている。

3.3.4 ヒープの人為的縮小

リダクション数を数えるには独立したカウンタの管理が必要で、完全に余分なオーバヘッドになる。そこで、ヒープ限界を実際よりも小さい値に設定することによって、偽のヒープオーバフローを起こす方法が考えられる。実際にオーバフローが起きたのかポーリングのための疑似オーバフローかは、真のヒープ限界と比較することによって判定できる。疑似オーバフローによるポーリングの後には、ヒープ限界をふたたび現在のヒープトップより少し上に設定し直すことによって、ある程度の時間の後にまたポーリングを行なうようにすることができる。

この方式はヒープ消費がほぼ一定の速さで進むことを仮定しているが、ヒープを全く消費しない無限ループもありうる。これは無意味な無限ループとは限らない。たとえば静的プログラム解析によるコンパイル時ガーベジコレクションによって、有用な計算においてもかなり長時間に渡りまったくヒープを消費しないような最適化が可能な場合も考えられる。このような将来の最適化の可能性と矛盾するような方式は不適当である。幸いヒープを消費しないリダクションはコンパイラで容易に検出できるので、そのような場合はヒープ限界を人為的に押し下げるようなコードを生成することで対応できる[14]。

リダクション数があまり大きくない誤差の範囲で計算時間に比例するのに比べ、ヒープの消費量は計算時間と良く比例しない場合が多い。このため、一定に近いポーリング間隔を実現しにくいのはこの方式の問題点である。逆にいうと、この方式でヒープ消費量の少ないプログラム部分でも十分短い間隔でポーリングするよう設定すると、ヒープ消費量の多いプログラム部分ではポーリング間隔が短かくなり過ぎ、大きな性能低下を招く可能性がある。

以上、さまざまな『逃げ手』をあげてきたが、どの方式が採用可能か、どの方式が性能上適しているかはプラットフォームの特性に大きく依存する。そこで、KLIC では複数の方式を用意し、プラットフォームごとに選択できるようにしている。

[14] この方式のアイディアの一部は中島浩氏に負っている。

3.4 他の問題点と解決法

ワーカ間の通信・同期以外にも、プラットフォームによってさまざまな違いがあり、移植が容易でなかった点が少なからずある。そのうちのいくつかについて、問題点と KLIC で採用した方式について述べる。

3.4.1 結果を待ち合わせない入出力

インターネットソケットとの結合を行なうシステムコール connect について、完了を待ち合わせずに他の処理を行なう機能 (O_NONBLOCK または O_DELAY) を提供していなかったり、完了を待ち合わせないことはできるが完了時に割り込みで通知する機能を提供していなかったりするプラットフォームが少なくない。これではたとえば地球の反対側 (もっと極端には冥王星あたりを飛んでいる人工惑星) との間にソケット結合を行なおうとすると、かなり長い間他の計算を止めなければならないことになる。

単純なファイルの入出力についても同様の問題がある。相手がネットワークストリームなどでない場合は計算機に直結した遅延の少ないディスクのみを想定しているらしく、遠隔地のファイルシステムを NFS などでマウントしている場合でも、待ち合わせなしの入出力ができないシステムが多い。

適切な機能が提供されていないものについては、この問題を解決する方法はないといえる。スレッドを用いることが唯一の手段であろうが、前述のように標準といえるスレッドライブラリはなく、また、ワーカ間の通信の場合と異なって単にタイミングを知らせるだけでなく実際の入出力を別スレッドに依頼する必要が生じるので、かなり複雑な処理になってしまう。このため、現在のところ KLIC では

特別の対処をしておらず、入出力の際に実行がブロックしてしまうことをあきらめている。

この問題点は並列処理特有のものではなく、逐次プログラムでも問題になるはずのものである。しかしながら、たとえ結果を待ち合わせない入出力機構が提供されていても、逐次処理用のプログラム言語で単一スレッドとして記述する場合には、応用プログラム側で自分で割り込み処理などを記述せねばならず、その繁雑さを考えれば容易にあきらめがつくのであろう。KLIC はこうした繁雑な処理を言語処理系側で引き受け、応用プログラムでは簡明な記述で済ませられるだけに残念なところである。

3.4.2 並列プログラムの立ち上げ方

並列プログラムをどう立ち上げるのかは、プラットフォームによりさまざまである。並列計算機システムの中には、複数のプロセッサ上で動作するプロセスをフロントエンドプロセッサなどから同時に立ち上げる必要があり、その後には他プロセッサ上のプロセスを作ることができないプラットフォームもある。プログラムの実行中に他プロセッサで動作するプロセスを作ることができる場合でも、プログラムの先頭からの起動しかできず、途中までの計算結果 (たとえばコマンドラインの解析結果) を渡すことができないようなプラットフォームもある。

このあたりは個別のプログラムの記述を考える限りは些細なことといってよいであろうが、KLIC のようなミドルウェアとなる汎用システムを移植性高く実現しようとする場合は、大きな問題になる。現在 KLIC はプラットフォームごとに異なる立ち上げコードを用意することによってこの問題を解決しているが、新たなシステムへの移植の際にかなりの作業量が必要になっている。

4 おわりに

並列論理型言語 KL1 の処理系 KLIC の構成方針と、それを種々の計算機システム上に移植した経験について述べた。その中で、現状の多くの計算機システム、ことにそのオペレーティングシステムが、動的負荷分散や見込み計算を伴う並列処理に必要とする十分な機能を備えていないことを指摘した。

一般に、現在広く使われているオペレーティングシステムの多くは、計算機上で複数の処理を並行して行なう処理、ことに非同期的な処理をサポートする機能が不十分であるように見受けられる。これは多くのオペレーティングシステムが逐次処理、あるいは、互いに独立な逐次処理を複数並行に実行する、という利用形態を前提に設計されているためではないかと考えられる。

多くのオペレーティングシステムにおいては、本質的な構成方針に大きな問題があるわけではない。本稿に述べてきたように、目的とする機能を実現するのには基本的には UNIX のシグナル機構で十分なのである。問題は、オペレーティングシステムの提供する個別のサービスについて、シグナル機構を用いて提供できるはずの非同期処理のための機能をきちんと提供していないことにある。

オペレーティングシステムの提供する個々のサービスすべてについて、非同期的処理をサポートするような機能拡充を行なうのはそう簡単ではないのだろう。しかし、今後ネットワーク上に分散した計算機システム群の利用が多くなっていくであろうし、見込み計算などの非同期的な処理を必要とする並列処理を行なう場合も多くなっていくであろうから、KLIC に限らずこうした並行処理を必要とするソフトウェアはますます増えていくであろう。そうなると、我々が KLIC の移植時に行なったようなさまざまな解決手段を、個別のソフトウェアについてその都度講じていくことは労力の無駄である[15]。

[15] KLIC 自体、こうした労力の無駄を省くためのミドルウェアと位置付けることもできるが。

今後、あらゆるオペレーティングシステムが非同期的処理をスムースに行なえるように拡充していくこと、そしてそうした拡充が「当然のこと」となり、たとえば待ち合わせなしの入出力ができないようなオペレーティングシステムは欠陥品であると認識されるようになることを望む。

なお、KLIC はフリーソフトウェアであり、誰でも以下から自由に転送し利用可能である。

http://www.icot.or.jp/AITEC/COLUMN/KLIC/klic-J.html
ftp://ftp.logos.t.u-tokyo.ac.jp/klic/

配布にはシステムのマニュアル (和文・英文) が含まれている。また、上述の AITEC のページからは講習会テキストも参照できる。

参考文献

[1] M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Brootz, H. Nilsson, and T. Sjöland. *SICStus Prolog User's Manual*, 1993.

[2] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of KL1. In M. Hermenegildo and J. Penjam eds., *Proceedings of PLILP'94*, pp. 25–39, Berlin, 1994. Springer-Verlag.

[3] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operating system (PIMOS). In *Proceedings of FGCS'88*, pp. 230–251, Tokyo, Japan, 1988.

[4] A. Goto, M. Sato, K. Nakajima, K. Taki, and A. Matsumoto. Overview of the parallel inference machine architecture (PIM). In *Proceedings of FGCS'88*, Tokyo, Japan, 1988.

[5] M. Morita, N. Ichiyoshi, and T. Chikayama. A shared-memory parallel execution scheme of KLIC. ICOT technical report, ICOT, 1994.

[6] K. Nitta, et al. Experimental parallel inference software. In *Proceedings of FGCS'92*, pp. 166–190, Tokyo, Japan, 1992.

[7] K. Rokusawa, A. Nakase, and T. Chikayama. Distributed memory implementation of KLIC. ICOT technical report, ICOT, 1994.

[8] Y. Takeda, H. Nakashima, K. Masuda, T. Chikayama, and K. Taki. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *Proceedings of FGCS'88*, Tokyo, Japan, 1988. Also in New Generation Computing 7–2, 3 (1990), pp. 179–195.

[9] K. Ueda. Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. ICOT Technical Report TR-208, ICOT, 1986.

[10] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, December 1990.

[11] 大野, 伊川, 森, 中島, 富田. 静的解析による並列論理型言語 KL1 の通信最適化. In *Proceedings of JSPP'95*, pp. 169–176, 1995.