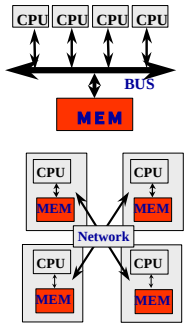
OpenMP

佐藤

並列処理の利点

- 計算能力が増える。
  - １つのＣＰＵよりも多数のＣＰＵ。
- メモリの読み出し能力（バンド幅）が増える。
  - それぞれのＣＰＵがここのメモリを読み出すことができる。
- ディスク等、入出力のバンド幅が増える。
  - それぞれのＣＰＵが並列にディスクを読み出すことができる。
- キャッシュメモリが効果的に利用できる。
  - 単一のプロセッサではキャッシュに載らないデータでも、処理単位が小さくなることによって、キャッシュを効果的に使うことができる。
- 低コスト
  - マイクロプロセッサをつかえば。

⇒ クラスタ技術



クラスタコンピューティング

- クラスタシステム：既存のワークステーションやＰＣを（既存の）ネットワークで結合して，並列計算を行うシステム
- 第１世代：既存のＬＡＮで並列計算するシステム
  - Poorman's supercomputer
  - 遊休のワークステーションを利用
  - お茶大 Sun IPC cluster （計算センタのワークステーションを利用）
  - イーサネット
- 第２世代：クラスタ専用の計算機システム
  - etlwiz Alpha cluster, 東大富士川研クラスタ
  - 100 BASE-TX SWITCH，ATM
  - beowulf class クラスタとも呼ばれる．
- 第３世代：高速のネットワークによるクラスタ
  - 高並列計算機（ＭＰＰ）なみの性能
  - RWCP PC cluster
  - Myrinet，Gigabit Ethernet, Fiber Channel, DEC Memory Channel, IBM SP2 network
- その他
  - ＳＭＰクラスタ （UCB CLUMPS, RWC COMPaS)

クラスタコンピューティングを支える技術

- ハード，ソフトのコモデティ化，高性能化，標準化
- ハードウエア
  - プロセッサテクノロジロードマップの恩恵
    - 急激な高性能化，価格性能比の向上
  - ネットワークの高性能化
    - ethernet : 10Mbps から 100Mbps そしてGigabit either
    - Myrinetなどの SAN Network
  - 高性能 I/Oインタフェースの標準化
    - ＰＣＩなど
- ソフトウエア
  - 並列通信ライブラリの発展・標準化
    - PVM, P4, TCGMSG，MPI，MPI2
  - 標準ライブラリ上に並列ソフトウエアが開発できる．
  - フリーなオペレーテングシステムの普及

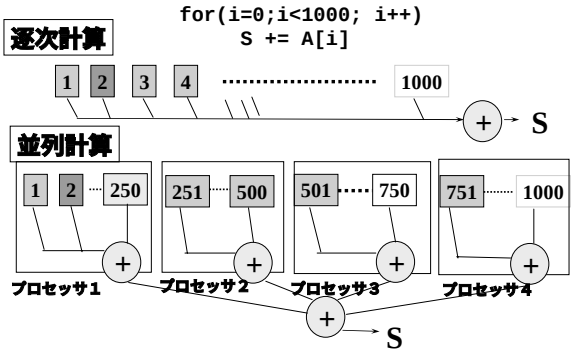
並列プログラミング

- メッセージ通信 (Message Passing)
  - 分散メモリシステム（共有メモリでも、可）
  - プログラミングが面倒、難しい
  - プログラマがデータの移動を制御
  - プロセッサ数に対してスケーラブル
- 共有メモリ (shared memory)
  - 共有メモリシステム（DSMシステムon分散メモリ）
  - プログラミングしやすい（逐次プログラムから）
  - システムがデータの移動を行ってくれる
  - プロセッサ数に対してスケーラブルではないことが多い。

並列プログラミング

- メッセージ通信プログラミング
  - MPI,PVM
- 共有メモリプログラミング
  - マルチスレッドプログラミング
    - pthread, solaris thread, NT thread
  - OpenMP
    - 指示文によるannotation
    - thread制御など共有メモリ向け
  - HPF
    - 指示文によるannotation,
    - distributionなど分散メモリ向け
- 自動並列化
  - 逐次プログラムをコンパイラで並列化
    - コンパイラによる解析には制限がある。指示文によるhint
- Fancy parallel programming languages

簡単な例



POSIXスレッドによるプログラミング

- スレッドの生成

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
      pthread_join();
```

PARAMCS

```
For(t=1; t<n_thd;t++)
  CREATE(thd_main);
thd_main(0)
WAIT_FOR_END(n_thd-1);
```

POSIXスレッドによるプログラミング

- ループの担当部分の分割
- 足し合わせの同期

```
int s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c,b,e,i,ss;
  c=1000/n_thd;
  b=c*id;
  e=s+c;
  ss=0;
  for(i=b; i<e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```

OpenMPによるプログラミング

これだけで、OK!

```
#pragma omp parallel for reduction(+:s)
   for(i=0; i<1000;i++) s+= a[i];
```

もくじ

OpenMPとは

- 共有メモリマルチプロセッサの並列プログラミングのためのプログラミングモデル
  - ベース言語(Fortran/C/C++)をdirective（指示文）で並列プログラミングできるように拡張
- 米国コンパイラ関係のISVを中心に仕様を決定
  - Oct. 1997 Fortran ver.1.0 API
  - Oct. 1998 C/C++ ver.1.0 API
  - (1999 F90 API?)
- URL
  - http://www.openmp.org/

背景

- 共有メモリマルチプロセッサシステムの普及
  - SGI Cray Origin
    - ASCI Blue Mountain System
  - SUN Enterprise
  - PC-based SMPシステム
- 共有メモリマルチプロセッサシステムの並列化指示文の共通化の必要性
  - 各社で並列化指示文が異なり、移植性がない。
    - SGI Power Fortran/C
    - SUN Impact
    - KAI/KAP
- OpenMPの指示文は並列実行モデルへのＡＰＩを提供
  - 従来の指示文は並列化コンパイラのためのヒントを与えるもの

科学技術計算とOpenMP

- 科学技術計算が主なターゲット
  - 並列性が高い
  - コードの5%が95%の実行時間を占める(?)
    - 5%を簡単に並列化する
- 共有メモリマルチプロセッサシステムがターゲット
  - small-scale(～16プロセッサ）からmedium-scale (～64プロセッサ）を対象
  - 従来はマルチスレッドプログラミング
    - pthreadはOS-oriented, general-purpose
- 共有メモリモデルは逐次からの移行が簡単
  - 簡単に、少しずつ並列化ができる。
    - （でも、デバックはむずかしいかも）

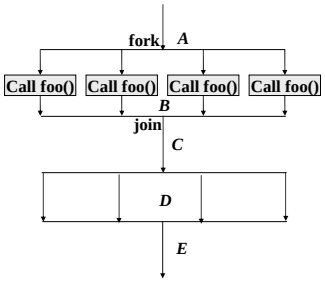
OpenMPのAPI

- 新しい言語ではない！
  - コンパイラ指示文（directives/pragma）、ライブラリ、環境変数によりベース言語を拡張
  - ベース言語：Fortran77, f90, C, C++
    - Fortran： !$OMPから始まる指示行
    - C: #pragma omp のpragma指示行
- 自動並列化ではない！
  - 並列実行･同期をプログラマが明示
- 指示文を無視することにより、逐次で実行可
  - incrementalに並列化
  - プログラム開発、デバックの面から実用的
  - 逐次版と並列版を同じソースで管理ができる

OpenMPの実行モデル

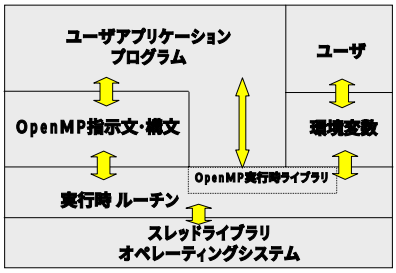
- 逐次実行から始まる
- Fork-joinモデル
- parallel region
  - 関数呼び出しも並列実行

```
… A …
#pragma omp parallel
{
   foo(); /* ..B... */
}
… C ….
#pragma omp parallel
{
… D …
}
… E …
```



OpenMPのアーキテクチャ

- OpenMPのAPI
  - 指示文･構文
  - 実行時ライブラリ
  - 環境変数



OpenMPの指示文フォーマット

- Fortran
  - OMP,COMP,*$OMPのsentinelから始まる行

  `!$OMP directive_name [clause, clause, …]`

  - *directive_name*: 指示子名
  - *clause*: 指示節、データ属性や並列ループのスケジューリング，同期オプションなどを指定する。
- C/C++
  - #pragma omp から始まるpragma行

  `#pragma omp directive_name [clause, clause, …]`

  - 構文要素として扱われるので注意
    - 例えば、`#pragma omp parallel` は後続のブロック文に作用する。

Parallel Region

- 複数のスレッド(team)によって、並列実行される部分
  - Parallel構文で指定
  - 同じParallel regionを実行するスレッドをteamと呼ぶ
  - region内をteam内のスレッドで重複実行
    - 関数呼び出しも重複実行

Fortran:

```
!$OMP PARALLEL
    …
    … parallel region
    …
!$OMP END PARALLEL
```

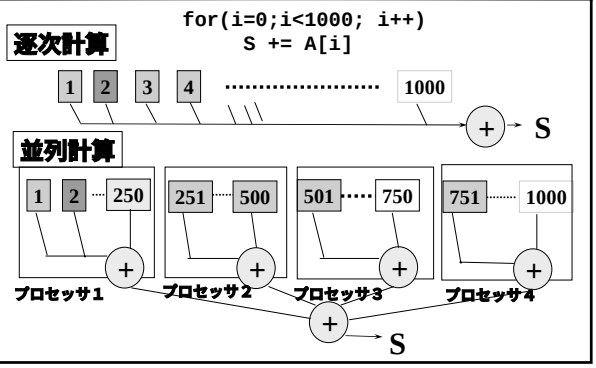
C:

```
#pragma omp parallel
{
    …
    … Parallel region…
    …
}
```

Parallel region (contd.)

- スレッド ID
  - 実行時ライブラリ関数 omp_get_thread_num()で得る。
  - IDは、Team内の０から始まる番号
  - マスタスレッド ID=0
  - IDを使って違うデータにアクセス。
- スレッド数
  - 実行時ライブラリ関数 omp_set_num_threads(*nthreads*)で設定
  - 環境変数 OMP_NUM_THREADS
- 同期
  - parallel regionの最後でjoin
  - 指示文による同期
    - critical, atomic, barrier
  - 実行時ライブラリを用いる
    - ロック関数

簡単な例



簡単な例

OpenMPによるマルチスレッドプログラミング

```
#pragma omp parallel
{ int c,b,e,i,ss;
  c=1000/omp_get_num_threads();
  b=c*omp_get_thread_num();e=s+c;ss=0;
  for(i=b; i<e; i++) ss += a[i];
#pragma omp atomic
  s += ss;
}
```

OpenMPによるデータ並列プログラミング

```
#pragma omp parallel for reduction(+:s)
   for(i=0; i<1000;i++) s+= a[i];
```

OpenMPの使い方

- 並列化指示として:
  - 並列ループを明示 (data-parallel)
  - タスク並列部分を明示 (task-parallel)
  - 自動並列化ではない → ユーザがtuning
- スレッドライブラリとして:
  - SPMDのプログラミングモデル
  - omp_get_thread_num()でスレッドIDを得る
  - SPLASH 2のPARMACS Macroの代わり
- 自動並列化コンパイラのbackendとして:
  - 自動並列化コンパイラがOpenMPのソースを出力
    - e.g. Polaris Compiler

マルチスレッドプログラミングとOpenMP

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
      pthread_join();
```

PARAMCS

```
For(t=1; t<n_thd;t++)
  CREATE(thd_main);
thd_main(0)
WAIT_FOR_END(n_thd-1);
```

OpenMP

```
  omp_set_num_threads(n_thd);
#pragma omp parallel
{
  thd_main(omp_get_thread_num());
}
```

Work sharing構文

- Team内のスレッドで分担して実行する部分を指定
  - parallel region内で用いる
  - `for` 構文
    - イタレーションを分担して実行
    - データ並列
  - `sections`構文
    - 各セクションを分担して実行
    - タスク並列
  - `single`構文
    - 一つのスレッドのみが実行
  - parallel 構文と組み合わせた記法
    - `parallel for` 構文
    - `parallel sections`構文

For構文

- Forループ（DOループ）のイタレーションを並列実行
- 指示文の直後のforループは*canonical shape*でなくてはならない

```
#pragma omp for [clause…]
 for(var=lb; var logical-op ub; incr-expr)
   body
```

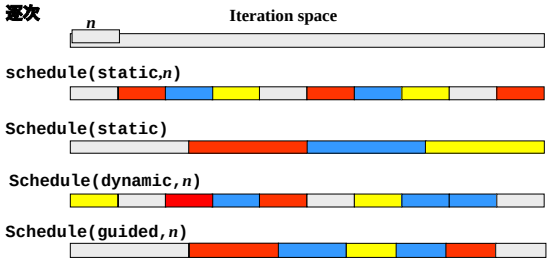
  - *var*は整数型のループ変数（強制的にprivate）
  - *incr-expr*
    - ++*var*, *var*++, --*var*, *var*--, *var*+=*incr*, *var*-=*incr*
  - *logical-op*
    - <、<=、>、>=
  - ループの外の飛び出しはなし、`break`もなし
  - *clause*で並列ループのスケジューリング、データ属性を指定

並列ループのスケジューリング

- For構文の指示節 `schedule(`*kind*[,*chunk_size*]`)` で指定
  - `schedule(static,`*chunk_size*`)`
    - *chunk_size*のイタレーションを静的に`round-roubin`でスレッドに割り当てる
    - 指定なし：プロセッサに等分割
    - *chunk_size*=1：cyclic分割
  - `schedule(dynamic,`*chunk_size*`)`
    - *chunk_size*のイタレーションを動的に割り当てる
    - 指定なし：*chunk_size*=1
  - `schedule(guided,`*chunk_size*`)`
    - 残りのイタレーションをプロセッサで動的に分割
    - *chunk_size*は最小の分割を指定
  - `schedule(runtime)`
    - 環境変数 `OMP_SCHEDULE`で指定
  - 指定なし：implementation依存

並列ループのスケジューリング

- プロセッサ数４の場合



例

疎行列ベクトル積ルーチン

```
Matvec(double a[],int row_start,int col_idx[],
 double x[],double y[],int n)
{
   int i,j,start,end; double t;
#pragma omp parallel for private(j,t,start,end)
   for(i=0; i<n;i++){
      start=row_start[i];
      end=row_start[i+1];
      t = 0.0;
      for(j=start;j<end;j++)
         t += a[j]*x[col_idx[j]];
      y[i]=t;
   }
}
```

Sections構文とsingle構文

- Sectionを各スレッドで並列実行

```
#pragma omp sections
{
#pragma omp section
 { … section1… }
#pragma omp section
 { … section2… }
}
```

- 一つのスレッドのみで実行

```
#pragma omp single
{
   … statements …
}
```

スレッドの同期操作

- Work sharing構文は、nowait指示節を指定しない限り、最後でバリア同期が行われる
- バリア同期
  - barrier 指示文
- Critical section
  - critical 構文
- Atomic 更新
  - atomic 構文

Barrier 指示文

- バリア同期を行う
  - チーム内のスレッドが同期点に達するまで、待つ
  - それまでのメモリ書き込みもflushする
  - 並列リージョンの終わり、work sharing構文でnowait指示節が指定されない限り、暗黙的にバリア同期が行われる。

```
#pragma omp barrier
```

Atomic構文

- メモリの更新をAtomicに行う。

```
#pragma omp atomic
        statement
```

  - 直後の文が、以下の形の更新でなくてはならない。
    - x *binop*= *expr*
    - x++,++x, x--, --xx
  - xのアドレス計算、*expr*の評価は並列に行われる。
  - Atomicなメモリ書き換えのハードウエアがあるときには高速化ができる。

Critical構文

- 排他的に実行されるCritical sectionを指定

```
#pragma omp critical[(name)]
{
      statements
}
```

  - 大域的な名前をつけることができる
    - 同じ名前のcritical sectionは排他的に実行される
    - 名前のない場合、他の名前のないcritical sectionと排他的に実行
  - conditional waitはなし
    - 逐次プログラムからの並列化を前提（？）

Master構文とordered構文

- master 構文

```
#pragma omp master
      block  statements
```

  - マスタスレッドだけで実行
  - 同期はなし
- ordered構文

```
#pragma omp ordered
      block  statements
```

  - for構文のdynamic extentにおいて、逐次と同様な順序で実行
  - for構文で、ordered指示節による指定が必要

Data scope属性指定

- parallel構文、work sharing構文で指示節で指定
- shared(*var_list*)
  - 構文内で指定された変数がスレッド間で共有される
- private(*var_list*)
  - 構文内で指定された変数がprivate
- firstprivate(*var_list*)
  - privateと同様であるが、直前の値で初期化される
- lastprivate(*var_list*)
  - privateと同様であるが、構文が終了時に逐次実行された場合の最後の値を反映する
- reduction(*op*:*var_list*)
  - reductionアクセスをすることを指定、スカラ変数のみ
  - 実行中はprivate、構文終了後に反映

Threadprivate 指示文

- スレッドごとに固有のfile-scopeの変数を指定

  `#pragma omp threadprivate(`*var_list*`)`

  - 変数宣言部に記述する
  - スレッド数が変わらない限り、parallel region間で変数の値がpersistentであることが保証される
  - parallel構文の`copyin(`*var_list*`)`指示節の指定により、マスタスレッドの値で初期化をすることができる。

Data scope属性指定とwork sharing構文

- Parallel 構文
  - `private,firstprivate,shared,reduction,copyin`
  - `default(shared|none)`
    - データ環境のdefaultを指定する。Noneを指定するとすべての変数に対して指定なくてはならない。
- for構文
  - `private,firstprivate,lastprivate,reduction`
- sections構文
  - `private,firstprivate,lastprivate,reduction`
- single構文
  - `private,firstprivate`

Orphan directiveとextent

- Static extent
  - lexicalに並列構文に含まれる部分
- dynamic extent
  - 実行時に並列に実行される部分
  - 並列構文内で呼び出される関数を含む
- orphan directive
  - Static extent以外のdynamic extentに現れる指示文
  - dynamic extentにない場合は無視される
- dynamic extentでのdata scope属性
  - auto変数は、private
  - 大域変数は、shared

例 (orphan directive)

```
main(){
 …
 for(it=0;it<NITER;I++){
  resid=cgsol(…)
  printf(…,resid);
 }
}
cgsol(…){
 …
#pragma omp parallel for
 for(i=0;i<cols;i+)
   p[i]=r[i]=x[i];
 for(it=0;it<NITCG;I++){
  matvec(…);
 …
#pragma omp parallel for
  for(I=0;I<cols;I++)
   z[I]+=alpha*p[I];
…
 }
}
```

```
main(){
 …
#pragma omp parallel
 for(it=0;it<NITER;I++){
  resid=cgsol(…)
#pragma omp master
   printf(…,resid);
 }
}
cgsol(…){
 …
#pragma omp for
 for(i=0;i<cols;i+)
   p[i]=r[i]=x[i];
 for(it=0;it<NITCG;I++){
  matvec(…);
 …
#pragma omp for
   for(I=0;I<cols;I++)
    z[I]+=alpha*p[I];
 …
 }
}
```

Directive binding

- `for`, `sections`, `single`,`master`, `barrier` directiveは、dynamic extentにbindされる
  - dynamic extent以外にあるのは、逐次実行
- work sharing構文は、nestできない。
  - master, criticalの中で使うことができない
- nested parallelism
  - parallel directiveはnestしてもよい
  - Nested parallelismがenableの時、parallelに実行
  - Disableの時には、一つのthreadのチームで実行（逐次）

Nested parallelism

- Nested parallelismについてのコメント
  - in FAQ ``What about nested parallelism?''
    - Nested parallelism is permitted by the OpenMP specification. Supporting nested parallelism effectively can be difficult, and we expect most vendors will start out by executing nested parallel constructs on a single thread.
  - In ``OpenMP Fortran Interpretations Version 1.0''
    - In Note that an OpenMP-compliant implementation is permitted to serialize a nested parallel region.
- 逐次実行での実行を保証する必要がある。
  - Nested parallelismのserialize
    - section構文のserialize
    - 並列ループのserialize
  - 逐次プログラムからの並列化を前提としている（?）

OpenMPのmemory consistencyモデル

- OpenMPの共有メモリモデルはweak consistency
  - 以下の場合に一貫性を保証すればよい。
    - Parallel regionの終了時
    - volatile変数の更新
    - バリア同期（nowaitのないwork sharing構文の終了時）
    - flush 指示文
- flush 指示文

```
#pragma omp flush[(var_list)]
```

  - 指定された変数のconsistencyを保証する。
  - 変数を指定されていない場合にはすべての共有変数

実行時ライブラリ

- omp_get_num_threads, omp_set_num_threads
  - teamのスレッド数を取得、変更
- omp_get_thread_num
  - スレッドidを取得
- omp_get_max_threads
  - 最大のスレッド数を返す
- omp_get_num_procs
  - プロセッサ数を返す
- omp_set_dynamic, omp_get_dynamic
  - スレッド数を動的に変更するかどうか
- omp_set_nested, omp_get_nested
  - parallel regionのnest実行が可能かどうかの指定
- lock関数
  - omp_lock_t
  - omp_nest_lock_t

環境変数

- OMP_NUM_THREADS
  - Parallel regionを実行するスレッド数を指定
- OMP_SCHEDULE
  - schedule(runtime)のスケジュール方法の指定
- OMP_DYNAMIC
  - スレッド数を動的に変えていいかどうかの指定
  - SGI origin では対応
- OMP_NESTED
  - nested parallelismが有効かの指定
  - nestされたparallel regionは逐次実行でも可

OpenMPの利点・欠点

- 利点
  - incrementalに並列化ができる。
  - 逐次実行版と並列実行版を同じソースで管理できる
  - ユーザ指示通りに並列化できる
  - スレッドプログラミングに比べて、並列性が構造的に記述されている。
    - Work sharing 構文、orphan directive
    - コンパイラで解析が可能
- 欠点
  - 並列化可能性はユーザがチェックする必要あり
  - data mappingが記述できない。
    - Iteration mappingとの整合性
    - localityが失われる可能性
  - 配列に対してreduction演算の指定ができない
  - コンパイラが必要
    - pragmaによる記述

まとめ

- OpenMP --- 共有メモリモデル向けの実行モデル＆ＡＰＩ
  - 逐次のベース言語(Fortran,C/C++)を拡張
  - fork-joinモデル
  - 並列性の構造的な記述
  - 逐次プログラムからのincrementalな並列化をサポート
- 動向
  - Fortran版のinterpretation がpublishされた
  - SC'99において、Fortran版のsecond versionが検討される予定
  - 研究課題
    - 最適化（同期除去、localityの最適化）
    - ＳＭＰクラスタ（MPI,HPFとの統合）
    - 分散メモリへの対応
    - 自動並列化コンパイラとの統合

Commercial products

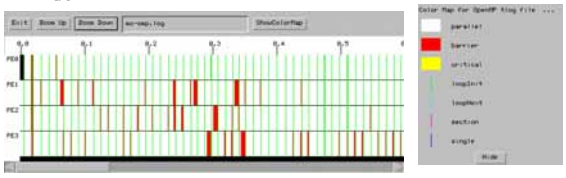
- KAI
  - Guide compiler(Fortran,C,C++)
    - Digital UNIX/NT alpha, HPUX,IBM AIX,Intel Solaris/NT,SGI,SUN Solaris
- PGI
- SGI
  - MIPSpro 7.2.1 (Fortran,C)
  - Gray UNICOS
- SUN
- COMPaQ/Digital Fortran
- IBM
- 日立 SR8000(?)
- NEC SX-4(?)

Performance tuning tools

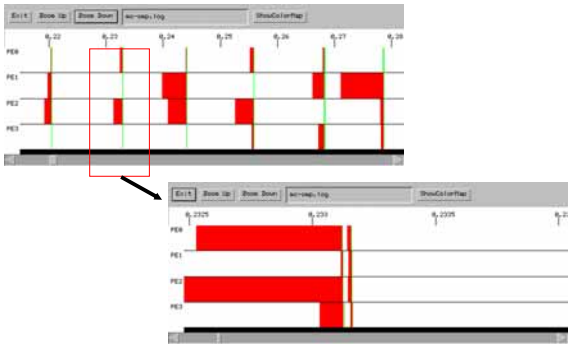
- 共有メモリプログラムはプログラミングが簡単だが、performance tuningが難しい
- 性能デバックツール
  - KAI Assure/Guide view
  - TAU (OGI)
- 自動並列化コンパイラによる並列化
  - Polaris コンパイラ

Omni tlogview

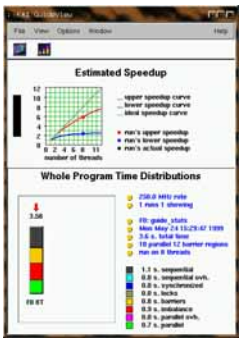
- Omni OpenMPコンパイラのdirectiveイベントのトレース(tlog file)を表示するツール
  - 時間軸上にbarrierなどのイベントを表示する
  - 並列プログラムの粒度などの見ることができる。
  - どこで、時間のロスがあるか。
  - 例：モンテカルロシミュレーション



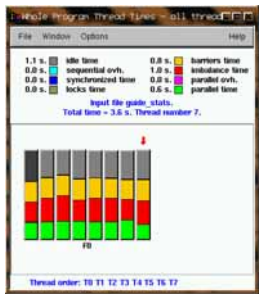
Omni tlogview



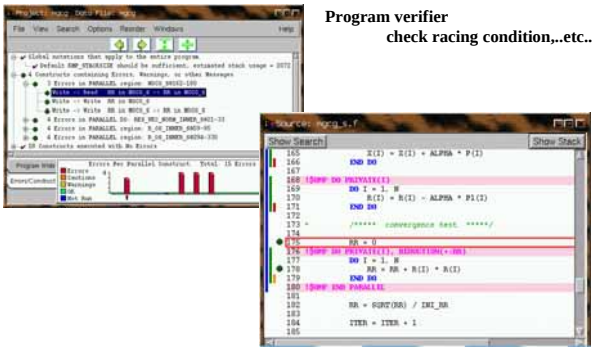
KAI Guide Tools



Performance Viewer



KAI Assure Tools



Program verifier
check racing condition,..etc..

関連研究

- OpenMP-NOW(Rice)
  - ネットワーク上の複数WSでソフトウエア分散共有メモリTradeMark上でOpenMPプログラムを並列実行
- OpenMP+MPI(ASCI)
  - 階層メモリを有効に利用する並列化手法
- OpenMP+HPF(Vienna Univ.)
  - 2種類の並列化指示を同時に指定するプログラミングの提案。
- ベンチマーク
  - SPEC HPGでは、OpenMP(共有メモリ向け）とMPI（分散メモリ向け）の２つの方法でベンチマークを標準化する方針
  - ＳＭＰクラスタには還在バージョンも。
- ＳＭＰクラスタ向けOpenMP(RWCP)

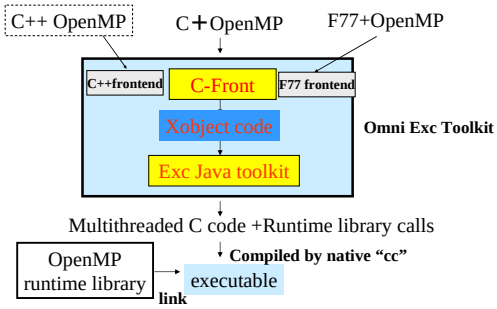
Omni OpenMPコンパイラ

- 概要
  - SMP版 (Solaris Thread or POSIX Threads)
  - Solaris 5.6 (SPARC,x86), linux 2.2.5 (x86 SMP)
  - 現在は、CとFortran77をサポート
  - コンパイラ開発環境の一部
    - C-front: C言語parser
    - exc-tools-java: Javaによるコード変換ツールキット
  - download
    - http://www.hpcc.jp/Omni/

RWC Omni OpenMP Compiler

- A translator from an OpenMP program to the multithreaded C program with the runtime library calls.
- Omni Exc toolkit
  - Toolkit for compiler research
  - C-front : OpenMP C parser to generate Xobject code
  - Xobject code: AST (Abstract Syntax Tree) and data type informations
  - Exc java toolkit : Java class libraries to analyze and transform Xobject code.
  - OpenMP transformation and optimization are written in Java using Exc java toolkit.
- Omni OpenMP compiler for SMP
  - Solaris Thread or POSIX Threads. (Stack/Threads at U. of Tokyo)
  - Solaris 5.6 (SPARC,x86), linux 2.2.5 (x86 SMP), (O2K pthread)
  - C and Fortran77. F90 is under development.

Overview of Omni OpenMP Compiler



OpenMPプログラムの変換例(1)

・入力プログラム

```
S1
#pragma omp parallel for
  for(i=0; i<N; i++){
    x[i]=…;
    …
  }
 S2
```

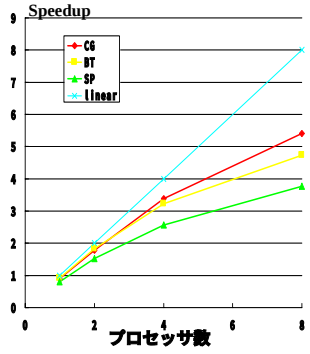
OpenMPプログラムの変換例(2)

```
void __ompc_func_1(void ** __ompc_args){
  auto int *_pp_n;
  _pp_n=(int *)*(__ompc_args+0);          引数設定
  { auto int _p_i, _p_i_0, _p_i_1, _p_i_2;
    _p_i_0=…;
    _ompc_static_sched(&_p_i_0, ...);     ループの担当範囲
    for(_p_i=…)...                        Parallelの本体
  }
}

S1
{ auto void * __ompc_argv[1];
  *(__ompc_argv+0)=(void *)&n;                    引数設定
  __ompc_do_parallel(__ompc_func_1,__ompc_argv);スレッド生成
}
S2
```

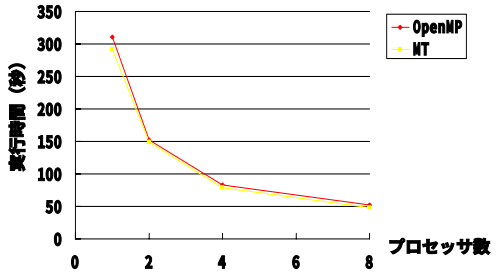
ケーススタディ（1）

- RWCP Omni OpenMP Compiler/C
- ベンチマーク
  - NPB1 CG,BT,SP (Class A) in C
  - orphan directive により overheadを減少
- プラットフォーム
  - SUN S1000(8CPU)



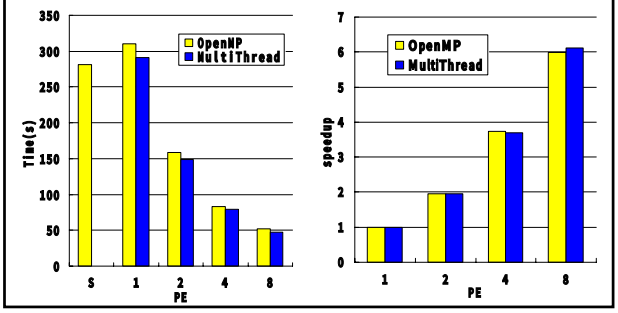
性能(RWC OpenMPコンパイラ)

- NPB1 CG(Class A) in OpenMP and Multithread(solaris thread)



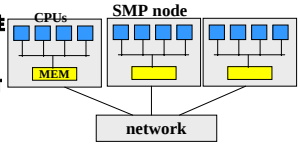
ケーススタディ（2）

- NPB1 CG (Class A) in OpenMP and Multithread (solaris thread)



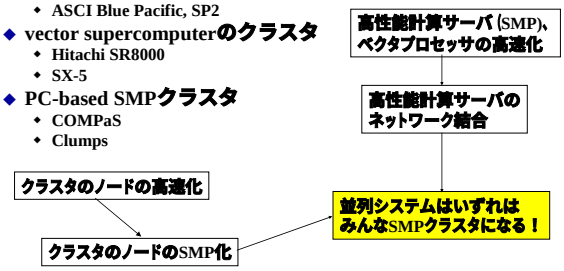
SMPクラスタ

- クラスタテクノロジ
  - PC、マイクロプロセッサの高性能化と普及、低価格化
  - ネットワークの高速化
  - MPPに匹敵する高性能計算の可能性
- SMPクラスタ
  - SMPがクラスタのノードとして使われつつある
    - SMPの低価格化
    - コンパクト、管理が容易
    - ネットワークが少なくてすむ
  - COMPaS: a PC-based SMP Cluster
    - SMPクラスタ利用技術の研究
      - プログラミング
      - 性能モデル



SMPクラスタ

- Middle scale Serverのクラスタ
  - ASCI Blue Mountain, O2K
  - ASCI Blue Pacific, SP2
- vector supercomputerのクラスタ
  - Hitachi SR8000
  - SX-5
- PC-based SMPクラスタ
  - COMPaS
  - Clumps



SMP クラスタでのプログラミング

- すべて共有メモリプログラミング
  - hardware shared memory + DSM
  - DSMのコストが高い？
  - スレッドプログラミングは簡単ではない
- すべてメッセージ通信プログラミング
  - MPI/shmem＋MPI
  - わざわざ、共有メモリ上でメッセージ通信をする?
  - メッセージ通信プログラミングは煩雑
- 共有・分散融合（Hybrid）プログラミング
  - ノード内は、スレッド＋共有メモリ
  - ノード間は、メッセージ通信またはリモートメモリ通信
  - 局所性を利用可。性能は得ることができる。
  - プログラミングコストが２倍！データ並列にはある程度適用できる
- MPI+OpenMP
  - 共有メモリプログラミングを容易にする。

MPIとOpenMPの混在プログラミング

- はじめに、MPIのプログラムを作る
- 並列にできるループを並列実行指示文を入れる
- 例：Cyclic Shiftによる並列行列積のプログラム

```
for(iter=0;iter<N_PE; iter++){
#pragma omp parallel for private(j,k,t) firstprivate(blkN)
    for(i=0; i<blkN;i++){
       t=0;
       for(j=0;j<blkN; j++) t+=A[k][i]*B[j][k];
       C[j][i]=t;
    }
    ……
     r=MPI_Sendrecv(…,B,BB,…);
    … update matrix, B <- BB, …
}
```

MPIとOpenMPの混在プログラミング

- MPI+OpenMP
  - はじめに、MPIのプログラムを作る
  - 並列にできるループを並列実行指示文を入れる
    - 並列部分はSMP上で並列に実行される。
  - 例：Cyclic Shiftによる並列行列積のプログラム
- OpenMP+MPI
  - OpenMPによるマルチスレッドプログラム
  - single構文・master構文・critical構文内で、メッセージ通信を行う。
    - thread-SafeなMPIが必要
    - いくつかの点で、動作の定義が不明な点がある
      - マルチスレッド環境でのMPI
      - OpenMPのthreadprivate変数の定義？
- SMP内でデータを共用することができるときに効果がある。
  - かならずしもそうならないことがある（メモリバス容量の問題？）

まとめ

- OpenMP --- 共有メモリモデル向けの実行モデル＆ＡＰＩ
  - 逐次のベース言語(Fortran,C/C++)を拡張
  - fork-joinモデル
  - 逐次プログラムからのincrementalな並列化をサポート
- SMPクラスタのプログラム
  - MPIのプログラムに並列に実行されるループをOpenMPによって並列化。
- ベンチマーク
  - SPEC HPGでは、OpenMP(共有メモリ向け）とMPI（分散メモリ向け）の２つの方法でベンチマークを標準化する方針
  - ＳＭＰクラスタには現在バージョンも。
- ＳＭＰクラスタ向けOpenMP(RWCP)
  - OpenMPそのままで、SMPクラスタ、が理想的

OpenMP for SMP Cluster

- OpenMP on SDSM
  - OpenMP(SIF) on TreadMarks (at Rice Univ.)
  - Omni OpenMP Compiler for SCASH (RWCP and TITECH)
  - This approach cannot exploit application-specific data access pattern.
- “Compiler-directed” SDSM
  - The compiler generates memory coherence check codes to keep memory consistency (e.g. Shasta SDSM).
  - The compiler analyzes the memory access pattern to optimize communication between nodes.
  - OpenMP structured parallelism description enables more high-level optimization
    - The data-parallel computation in work sharing directives can be compiled into efficient and explicit communication by compiler analysis.

おわりに

- OpenMPは、共有メモリ向けの並列プログラミングモデル
  - 現実的なアプローチ
  - 逐次プログラミングからの移行が容易
- これからの並列プラットフォームはSMPクラスタ
  - OpenMP+MPIが短期的な解
  - SMPクラスタ向けOpenMPコンパイラ
  - HPF？